International Journal of Enhanced Research in Management & Computer Applications
ISSN: 2319-7471, Vol. 5 Issue 5, May-2016, Impact Factor: 1.544

Model Smells In Uml Class Diagrams

Parul[1], Brahmaleen Kaur Sidhu[2]

[1]M. Tech Research Scholar at Punjabi University Patiala
[2]Assistant Professor at Punjabi University Patiala

1. INTRODUCTION

Models are the primary artifacts in software development processes following the model-based paradigm. Especially in Model Driven Software Development (MDSD), models become primary artifacts where quality assurance of the overall software product considerably relies on the quality assurance of involved software models. In this report the approach is focused towards the importance of Model Driven Architecture (MDA), Model Driven Engineering (MDE), and Model Driven Development (MDD) in Model Driven Software Development (MDSD).Then focus towards the quality of software models, considering model quality assurance processes that concentrate on the syntactical dimension of model quality, introducing the concept of model smells, using typical quality assurance techniques such as model metrics and anti-patterns, a prototype Eclipse plug-in providing specification and detection of smells in models based on the Eclipse Modeling Framework.

Model-driven software development (MDSD) is an alternative to round-trip engineering. Round-trip engineering is the concept of being able to make any kind of change to a model as well as to the code generated from that model. In MDSD the model is more abstract than the code generated from it. It is generally impossible to keep the model consistent automatically after a manual change of the generated code, therefore a precise definition that states which parts are generated and which are implemented manually is necessary.

Over the last few years, as tools and technologies have evolved, another option has evolved to define a software-solution's architecture: Model-Driven Development (MDD). MDD gives architects the ability to define and communicate a solution while creating artifacts that become part of the overall solution.

Another important approach under MDSD is MDE (Model Driven Engineering), which is a software development methodology which focuses on creating and exploiting domain models (that is, abstract representations of the knowledge and activities that govern a particular application domain), rather than on the computing (i.e. algorithmic) concepts.

The MDE (Model Driven Engineering) approach is meant to increase productivity by maximizing compatibility between systems (via reuse of standardized models), simplifying the process of design (via models of recurring design patterns in the application domain), and promoting communication between individuals and teams working on the system (via a standardization of the terminology and the best practices used in the application domain). A modeling paradigm for MDE is considered effective if its models make sense from the point of view of a user that is familiar with the domain, and if they can serve as a basis for implementing systems. The models are developed through extensive communication among product managers, designers, developers and users of the application domain. As the models approach completion, they enable the development of software and systems.

Some of the better known MDE initiatives are:

- The first tools to support MDE were the Computer-Aided Software Engineering (CASE) tools.
- To overcome the shortcomings of case tools, the US government developed Unified Modeling Language (UML)
- Rational Rose, a product for UML implementation, was done by Rational Corporation (Booch,).
- The Object Management Group (OMG) initiative model-driven architecture (MDA), which is a registered trademark of OMG.
- The Eclipse ecosystem of programming and modelling tools (Eclipse Modeling Framework).

Model-driven architecture (MDA) is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as models. Model-driven architecture is a kind of domain engineering, and supports model-driven engineering of software systems. It was launched by the Object

Management Group (OMG) in 2001. The model-driven architecture approach defines system functionality using a platform-independent model (PIM) using an appropriate domain-specific language (DSL).Then, given a platform model corresponding to CORBA, .NET, the Web, etc., the PIM is translated to one or more platform-specific models (PSMs) that computers can run. This requires mappings and transformations and should be modeled too.

The PSM may use different DSLs or a general purpose language. Automated tools generally perform this translation. MDA principles can also apply to other areas such as business process modeling (BPM) where the PIM is translated to either automated or manual processes.

MDA is a new way to look at software development, from the point of view of the models. Separates the operational specification of a system from the details such as how the system uses the platform on which it is developed. Three fundamental objectives are: portability, interoperability and reuse.

MDA provides a means to:

- Specify a system independently of its platform
- Specify platforms
- Chose a platform for the system
- Transform the system specifications into a platform dependent system.

Software quality assurance is a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes and procedures. It includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed upon standards and procedures is evaluated through process monitoring, product evaluation and audits. Software development and control processes should include quality assurance approval points where an assurance evaluation of the product may be done in relation to the applicable standards.

2. BAD SMELLS AND ANTI- PATTERNS

Software quality assurance is a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes and procedures. It includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed upon standards and procedures is evaluated through process monitoring, product evaluation and audits. Software development and control processes should include quality assurance approval points where an assurance evaluation of the product may be done in relation to the applicable standards.

There are many known good and bad practices in software engineering, which prove to be a threat to software quality. In the field of software design and source code quality, these practices are called Design Patterns for ‘good’ practices and Bad Smell or Anti-Pattern for ‘bad’ practices. This report summarizes some of these practices.

Design Patterns are known solutions to common design problems in software engineering. They are known good solutions for general design problems. Design patterns are usually defined as a relation between communicating objects of a software system or the way classes are structured. For example, the Decorator pattern can be used to dynamically add responsibilities to an object, rather than having the responsibilities predefined through inheritance. This can add flexibility to objects at runtime which cannot be fixed at compile time. Using Design Patterns correctly should improve the internal reuse of code (such as less duplicate code and more reusability of code) and improve maintainability. It makes extending a product easier and reduces coupling between components so they can be modified without affecting each other.

Opposed to Design Patterns are Anti-Patterns. Anti-Pattern are patterns (e.g., known strategies) that are applied in an inappropriate context. Anti-Patterns are poor solutions of recurring design problems which decrease software quality. They are outlined as violations of various quality rules. Based on this definition, there are two types of Anti-Patterns. The first is Design Patterns that are used in the wrong context. The other variant is known Bad patterns or Anti-Pattern that are used anywhere.

Another bad coding practice is the Bad Smell. Bad Smells or Code Smells are code taints such as long methods, code duplication and data classes. The difference between Bad Smells and Anti-Patterns is that Bad Smells tend to be local code taints within methods or classes. Anti-Patterns are usually more structural problems, such as classes using an inappropriate

hierarchy. Furthermore, Code Smells are usually implementation problems where Anti-Patterns are design problems. For this reason, Anti-Patterns are sometimes called Design Smells.

3. MODEL QUALITY : 6C Model quality goals presented by Mohagheghi et al

Software Quality comprises of quality of product, service, information, processes, people, and system. There are numerous definitions of quality. The ISO 9000 model defines quality as the degree to which a set of inherent characteristics fulfills requirements. ISO 9126, a refinement of the ISO 9000 model, which proposes a quality standard for software product evaluation, defines software quality as the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs.

Software quality focuses four areas viz; product, project quality, process, postproduction quality. The first area, product quality, is concerned with the requirements and specifications of the product as it applies to the attributes or characteristics of the software product. This area could also be referred to as end-product quality. The second area, project quality, is concerned with the metrics and measurements associated with the software production effort. The third area is process or management quality, which is concerned with the processes, planning and controls used to develop and manage the software product. The last area of focus is on post-production quality or deployed application management.

The paradigm of model-based software development has become more and more popular, since it promises an increase in the efficiency and quality of software development. It is sensible to address quality issues of artifacts in early software development phases already, for example the quality of the involved models. Especially in model driven software development, models become primary artifacts where quality assurance of the overall software product considerably relies on the quality assurance of involved software models.

In their article [4], Mohegheghi et al. present the results of a systematic review of literature discussing model quality in model-based software development. Among others, the purpose of the review was to identify what model quality means, i.e. which quality goals are defined in literature. The review was performed systematically by searching relevant publication channels for papers published from 2000 to 2007. From 40 studies covered in the review, the authors identified six classes of quality goals, called 6C goals, in model-based software development. They state that other quality goals discussed in literature can be satisfied if the 6C goals are in place. The remainder of this section shortly introduces the identified 6C goals.

Correctness: A model is correct if it includes the right elements and correct relations between them and, what is most important, if it includes correct statements about the domain. Furthermore, a model must not violate rules and conventions. This definition includes syntactic correctness relative to the modeling language as well as semantic correctness related to the understanding of the domain.

Completeness: A model is complete if it has all necessary information that is relevant, and if it is detailed enough according to the purpose of modeling. For example, requirement models are said to be complete when they specify all the black-box behavior of the modeled entity, and when they do not include anything that is not in the real world.

Consistency: A model is consistent if there are no contradictions within. This definition covers horizontal consistency concerning models/diagrams on the same level of abstraction, vertical consistency concerning modeled aspects on different levels of abstraction as well as semantic consistency concerning the meaning of the same element in different models or diagrams.

Comprehensibility: A model is comprehensible if it is understandable by the intended users, either human users or tools. In most of the literature, the focus is on comprehensibility by humans including aspects like aesthetics of a diagram, model simplicity or complexity, and the use of the correct type of diagram for the intended audience.

Confinement: A model is confined if it agrees with the modeling purpose and the type of system. This definition also includes relevant diagrams on the right abstraction level. Furthermore, a confined model does not have unnecessary information and is not more complex or detailed than necessary. Developing the right model for a system or purpose of a given kind also depends on selecting the right modeling language.

Changeability: A model is changeable if it can be evolved rapidly and continuously. This is important since both the domain and its understanding as well as requirements of the system evolve with time. Furthermore, changeability should be supported by modeling languages and modeling tools as well.

4. UML MODEL SMELLS : UML Class Diagram Smells

In this portion we describe selected UML model smells found in literature that are suitable in an early stage of a model-based software development process. For each model smell a short description is given as well as possible indicators to detect this smell in a given model. Furthermore, we present a list of quality characteristics and quality goals affected by this smell, and an example complete each model smell description.

4.1 Attribute Name Overridden

Description : The class defines a property with the same name as an inherited attribute. For this smell, it is essential that the property redefines the inherited attribute in order to conform to the UML specification. The redefinition of attributes might be confusing to model viewers. Furthermore, this smell might produce conflicts in model-driven processes. During code generation, this smell may inadvertently hide the attribute of the parent class.

Affected quality characteristics and goals : Redefined attributes may lead to more complexity and might be a typical case for redundant modeling. Simplicity, Redundancy, Comprehensibility, Consistency, Confinement, Changeability, Correctness.

Example : In Figure 4.1 there is an attribute horsepower in class Car that redefines the equally named attribute in abstract superclass Vehicle. This is done to specialize the type of the attribute, i.e. there is a restriction of the attribute's type. Sometimes, such a redefinition might be confusing and decreases the model's comprehensibility.

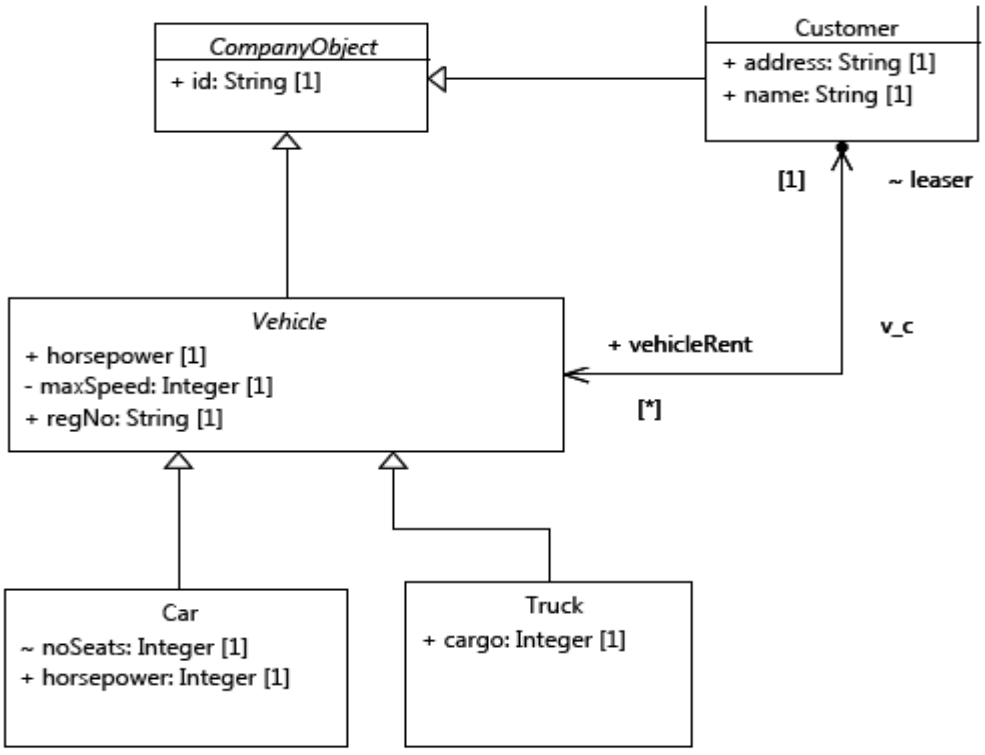

Figure 4.1 Example UML Model Smell Attribute Name Overriden

4.2 Concrete Superclass

Description: An abstract class that is a subclass of a non-abstract class reflects poor design and a conflict in the model's inheritance hierarchy. In other words, if an abstract class has any super classes these have to be abstract as well.

Affected quality characteristics and goals: Concrete super classes of abstract subclasses may not reflect a model aspect in the right way. Furthermore, this may lead to more complex models that are harder to understand. Precision, Simplicity, Correctness, Comprehensibility.
Example: Figure 4.2 shows an example class hierarchy. Abstract class PublicBuilding together with its subclasses Library and Church represent a valid class hierarchy whereas class House exactly addresses smell Concrete Superclass. If this class were also an abstract class, for example named Building, the entire class hierarchy would be valid again.

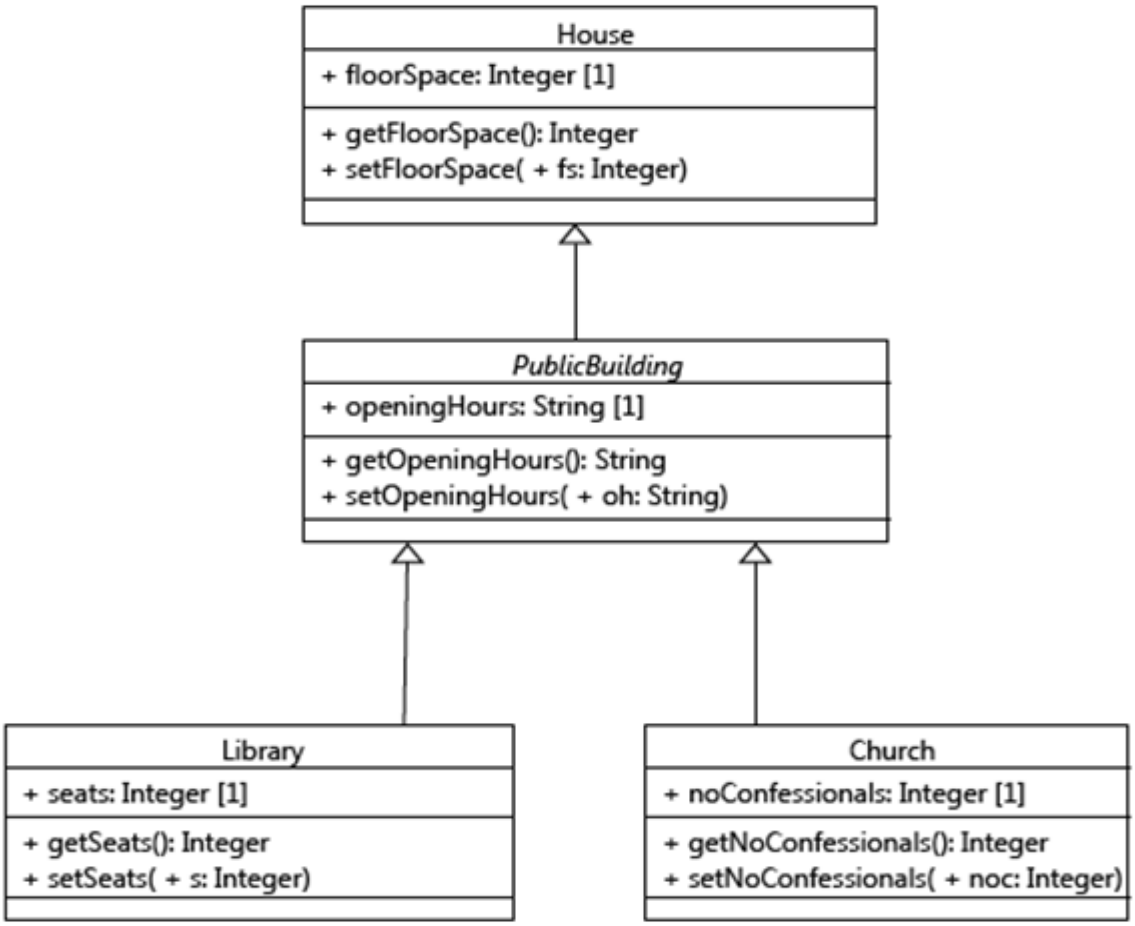

Figure 4.2 Example UML Model Smell Concrete Superclass

4.3 Data Clumps

This smell is described as interrelated data items which often occur as 'clump' in the model. Often, there are the same three or four data items together in lots of places, either attributes in classes or parameters in operation signatures..

For attributes they define:

1. More than three attributes stay together in more than one class.
2. These attributes should have same signatures (same names, same types, and same visibility).
3. These data fields may not group together in the same order.

For parameters they define:

1. More than three input parameters stay together in more than one operations' declaration.
2. These parameters should have same signatures (same names, same types).
3. These parameters may not group together in the same order.

Detection: This smell can be detected by matching corresponding patterns based on the abstract syntax of UML. A more common alternative of this smell, independent from the number of involved elements, also considers metrics Number of attributes and Number of parameters, respectively.
Affected quality characteristics and goals: Data clumps represent redundantly modeled aspects. They may be harder to understand and may not conform to a modular design. Redundancy, Simplicity, Cohesion/Modular Design, Comprehensibility, Changeability, Correctness.

Example: In Figure 4.3 there are attributes customerName, customerStreet, customerZip, and customerCity that occur in altogether four different classes.

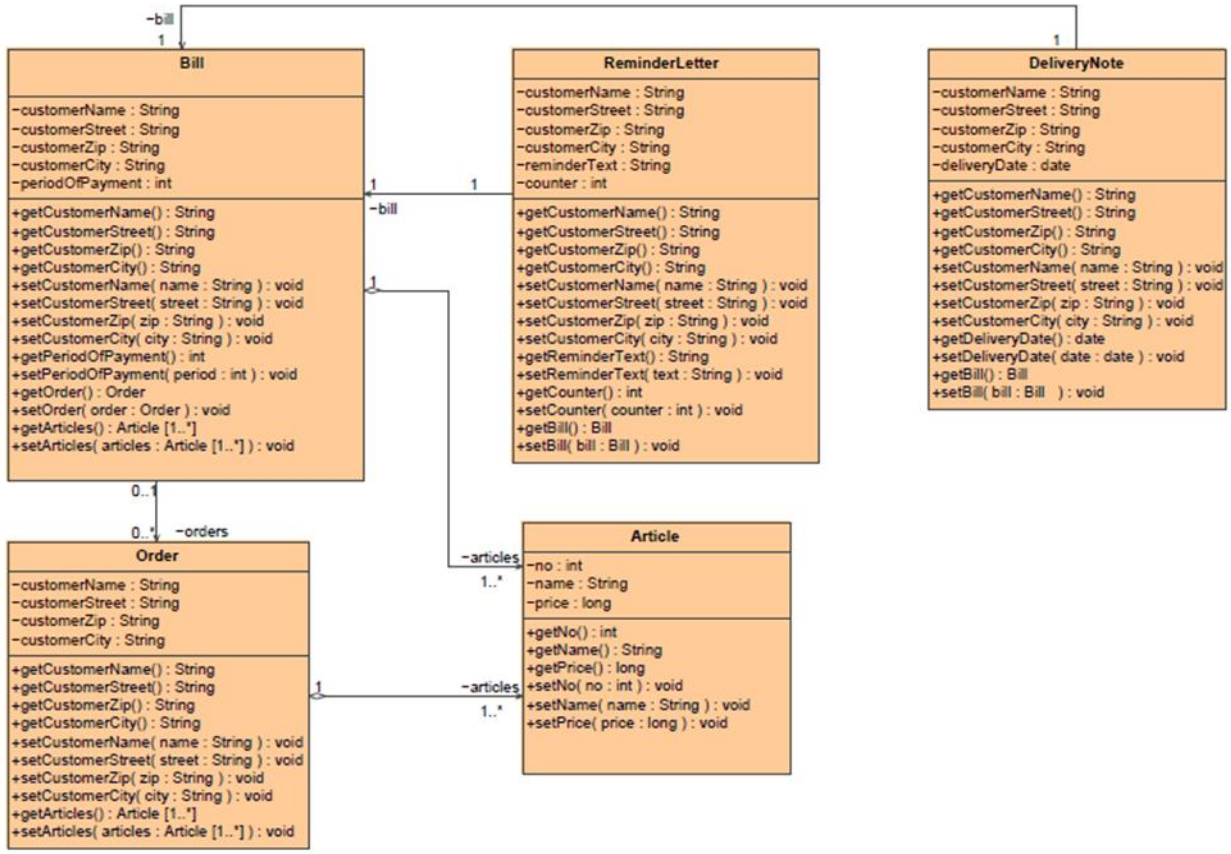

Figure 4.3 Example UML Model Smell Data Clumps [4]

4.4 Inverted Operation Name

Description: The behavior of complex operations in a class diagram is often modeled using sequence diagrams. Sequence diagrams are interaction diagrams and so they involve sender and receiver. Additionally, actors can be modeled. Starting with a sequence diagram, the developer names an operation as if this operation is an action of the sender. However, in the corresponding class diagram this operation would be named as if it is an action of the receiver. So, starting with a sequence diagram it might occur that an operation name makes no sense in the context of its receiver.

Detection: This model smell is hard to detect using syntactical check since it exclusively affects semantic concerns. Here, the use of text similarities combined with pre-defined pairs of inverse names would be helpful.

Affected quality characteristics and goals: Wrong named operations are semantic modeling failures and may be harder to understand. Semantic Adequacy, Simplicity, Correctness, Consistency, Comprehensibility.

Example: On their web site, Cunningham & Cunningham, Inc. gives an impressive example of this smell: I had to scratch my head over the name deliverPart on an interface the other day. When I looked at the code, I realized that it should have been named acceptPart. How does this happen? Well, the developer started with a sequence diagram and named the method as if it was an action from the caller.

4.5 Large Class

Description: A class should model an entity representing one single aspect of a given domain. So, its features (attributes and operations) should be balanced well. A class having too much features belonging to different concerns hints for too much information that should be expressed by this class. Often, this is the central class of a diagram. In this case, the surrounding classes may be inordinately small, which is also a smell. In any case, the significant difference in the relative sizes of the classes is the important thing.

Detection: This model smell can be easily detected by observing the class diagram with all members shown. Another check is to use metrics Number of attributes and Number of operations to determine the relative sizes of the classes in a calculational way.

Affected quality characteristics and goals: Large classes do not represent a good modular design and may contain redundant information.Presentation, Cohesion/Modular Design, Redundancy, Comprehensibility, Changeability, Correctness.

Example: In Figure 4.4 it is obvious that class Bill represents this model smell. Except for its remarkable number of operations which are mainly accessors or mutators, this class owns much more attributes than the average attribute number of the other classes.

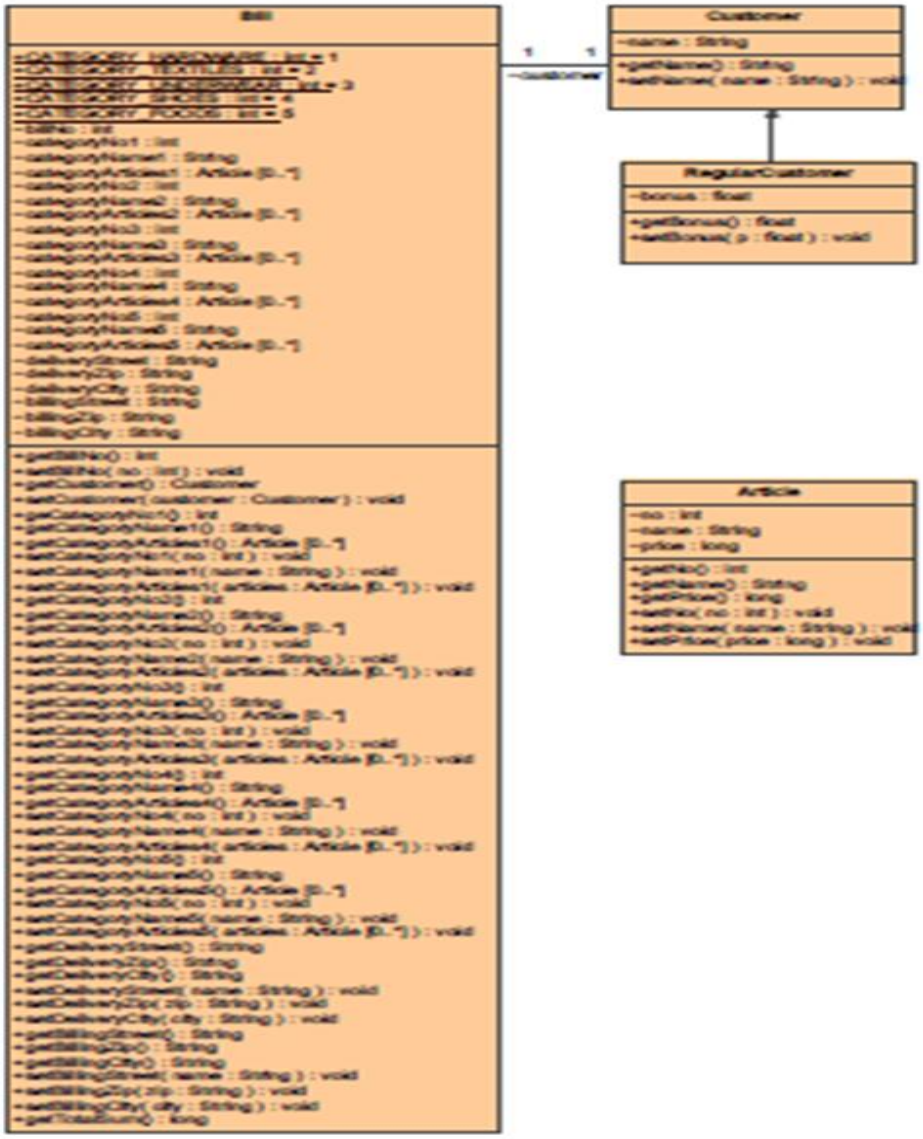

Figure 4.4 Example UML Model Smell Large Class [4]

4.6 Long Parameter List

Description: An operation has a long list of parameters that makes it really uncomfortable to use the operation. Long parameter lists are hard to understand and difficult to use. Furthermore, using long parameter lists is not intended by the object-oriented paradigm. An operation shouldhave only as much parameters as needed for solving the corresponding task. It is recommended to pass only those parameters that cannot be obtained by the owning class itself.

Detection: This smell can be simply detected by observing the model or by evaluating metric Number of parameters.

Affected quality characteristics and goals: Long parameter lists may be harder to understand and may contain redundant information. Presentation/Aesthetics, Simplicity, Cohesion/Modular Design, Comprehensibility, Changeability, Correctness.

Example: In Figure 4.5 class CustomerRelationshipManager owns two operations each having a long parameter list. Here, this smell can easily be detected by observation.

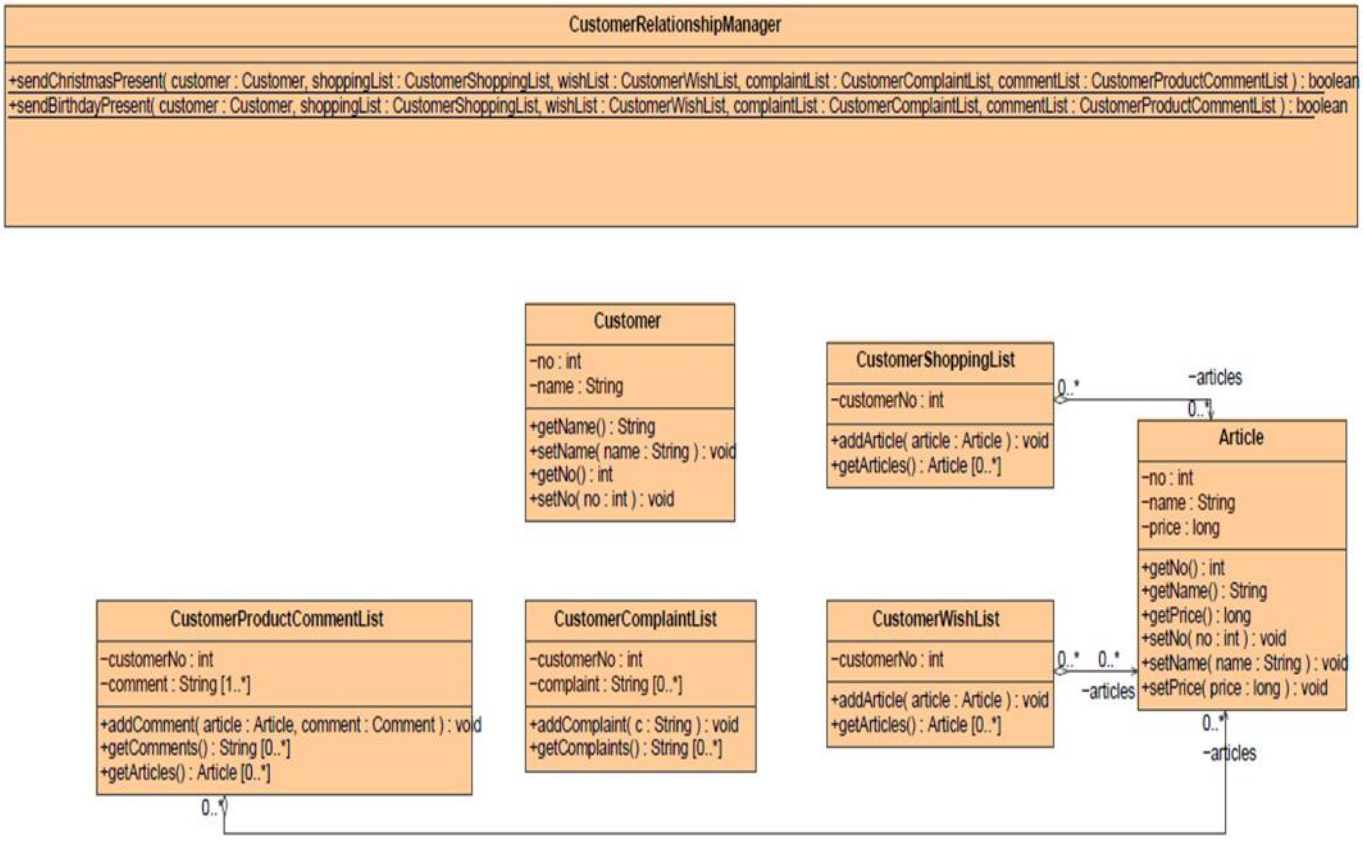

Figure 4.5 Example UML Model Smell Long Parameter List [4]

4.7 Multiple Definitions of Classes with Equal Names

Description: This smell occurs if in a single model more than one class has the same name. The different classes with the same name may be defined in the same diagram or in different diagrams. It is essential that equally named classes are owned by distinct packages (namespaces) in order to respect the uniqueness of qualified names in UML. Equally named classes could lead to misunderstandings of the modeled aspects. Furthermore, this smell will cause problems during code generation in a model-driven process.

Detection: This smell can be detected by matching a corresponding pattern based on the abstract syntax of UML.

Affected quality characteristics and goals: Equally named classes are redundancy at its best. Redundancy ,Correctness, Consistency, Comprehensibility,Changeability.

Example: In Figure 4.6 there is a class Customer in package Rental owning attributes name and address as well as associating its rented vehicle. Furthermore, there is another class Customer in package Accounting modeling the aspect that a Customer holds at least one account. This situation reflects a typical case of redundant modeling that can be concretized by smell Multiple Definitions of Classes with Equal Names.

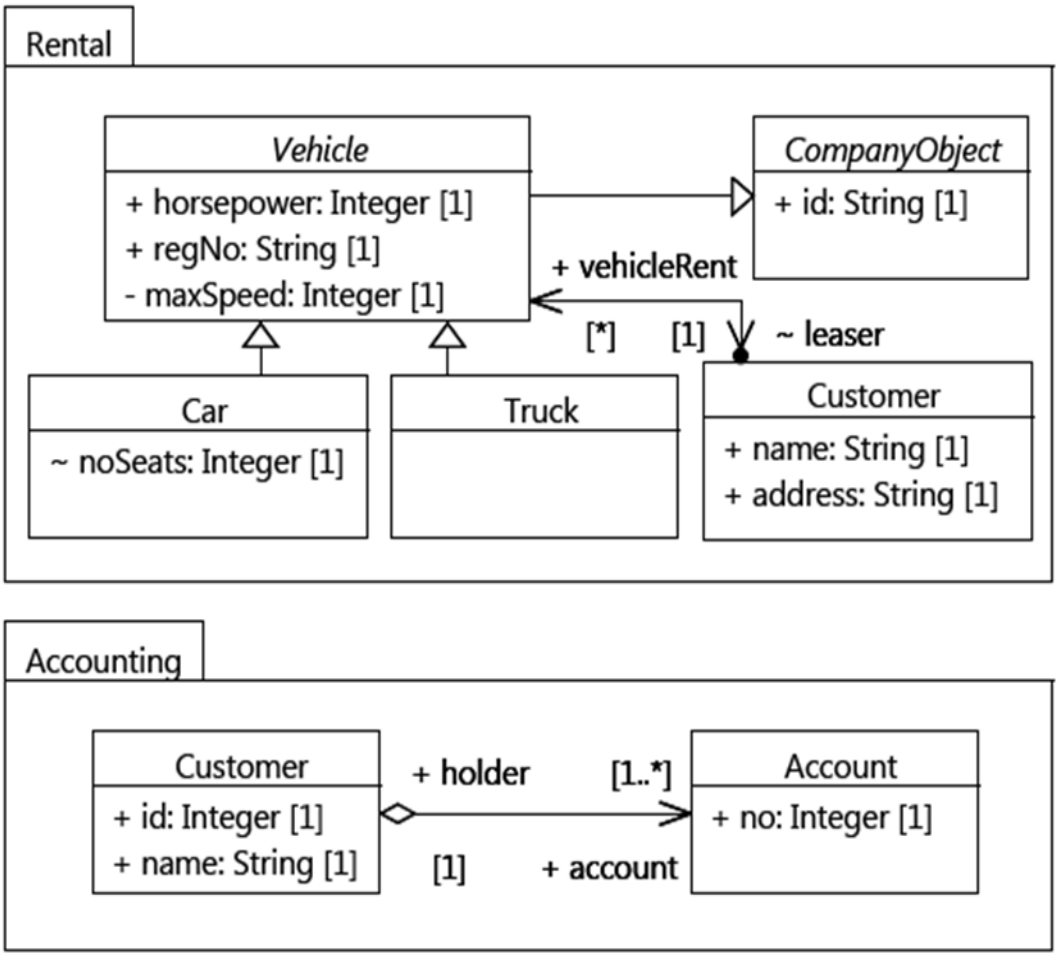

Figure 4.6 Example UML Model Smell Multiple Definitions of Classes with Equal Names

4.8 Primitive Obsession

Description: In this smell, primitive data types like String or Integer are used to encode data that would be better modeled as a separate class. Mostly this is done since developers are reluctant to use small classes for small tasks. Here, the use of even small classes might be a better choice to increase the understandability of the model. Furthermore, it is against the object-oriented paradigm to treat domain objects, even small ones, as primitive type instead of a class modeling its constituent parts.

Detection: This smell can hardly be detected. An indicator for this smell might a high value of metric Number of constant attributes since this might be a hint for the misuse of an enumeration. Also, a high value of metric Number of attributes combined with primitive attribute types might indicate the existence of this smell.

Affected quality characteristics and goals: Using primitive types instead of small classes might show problems in modular design. Furthermore, this might be imprecise and might reect semantic misunderstandings. Cohesion/Modular Design, Semantic Adequacy, Precision, Simplicity, Correctness, Confinement, Comprehensibility.

Example: In Figure 4.7 this smell occurs twice. First, there are four constant attributes which would be better modeled as enumeration. Furthermore, there are many attributes of primitive type Integer which partially adhere to each other and naturally present a point or coordinate, respectively.

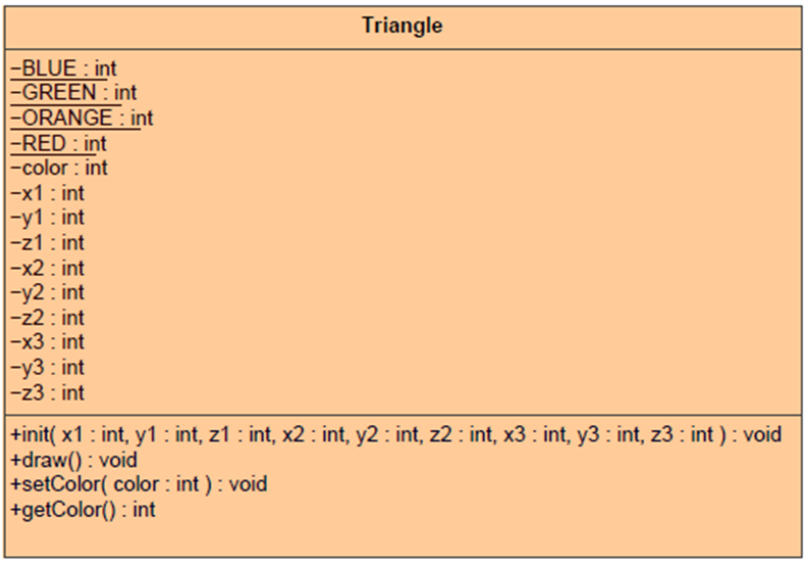

Figure 4.7 Example UML Model Smell Primitive Obsession [4]

4.9 Specialization Aggregation

Description: The association is a specialization of another association. This means, that there is a generalization relation between the two involved associations. People are often confused by the semantics of specialized associations. The suggestion is therefore to model any restrictions on the parent association using constraints.

Detection: This smell can be detected by matching a corresponding pattern based on the abstract syntax of UML.

Affected quality characteristics and goals: Specialized associations are hard to understand and might represent redundant modeling since involved classes can be already specializations. Simplicity, Redundancy , Comprehensibility.

Example: In Figure 4.8 there is a class Journey subclassed by class AirJourney. Also there is a similar class inheritance hierarchy including classes Route and AirRoute. Furthermore, there is an association between both subclasses Journey and Route. This association is also specialized by a corresponding association. In fact, this association hierarchy might be confusing.

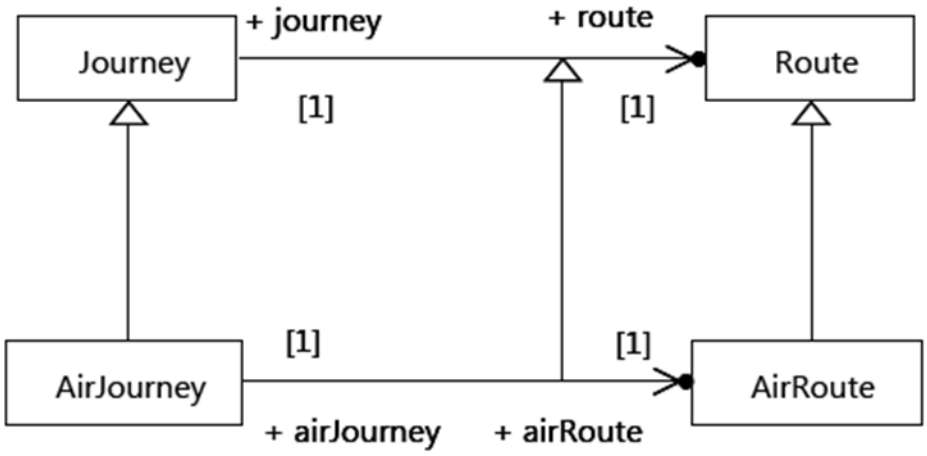

Figure 4.8: Example UML Model Smell Specialization Aggregation

4.10 Speculative Generality

Description: Often, developer model special cases but it is not essential to hold this information in the model. This is done since the developer intends to use this specific information sometime. In such cases this additional elements should be

excluded to avoid an increase in the complexity of the model. Not required information might lead to an ambiguous model. This kind of smell includes: abstract classes that are not doing much, methods with unused parameters, methods named with odd abstract names. The smell occurs if this element has not been inherited/implemented, or is only inherited/implemented by one single class/interface.

Detection: This smell can be detected by matching a corresponding patterns based on the abstract syntax of UML in the case of abstract classes and interfaces. Furthermore, it can be checked whether corresponding metrics like Number of direct subclasses and Number of implementing classes are evaluated to zero or one, respectively.

Affected quality characteristics and goals: This smell may lead to more complex models that might be harder to understand. Simplicity, Presentation, Comprehensibility, Confinement.

Example: In Figure 4.9 there are two abstract classes AbstractLong and AbstractDouble that are only inherited by one single concrete class each. These classes might be modeled to address future concerns. But infact, they are non-essential and shall be removed.

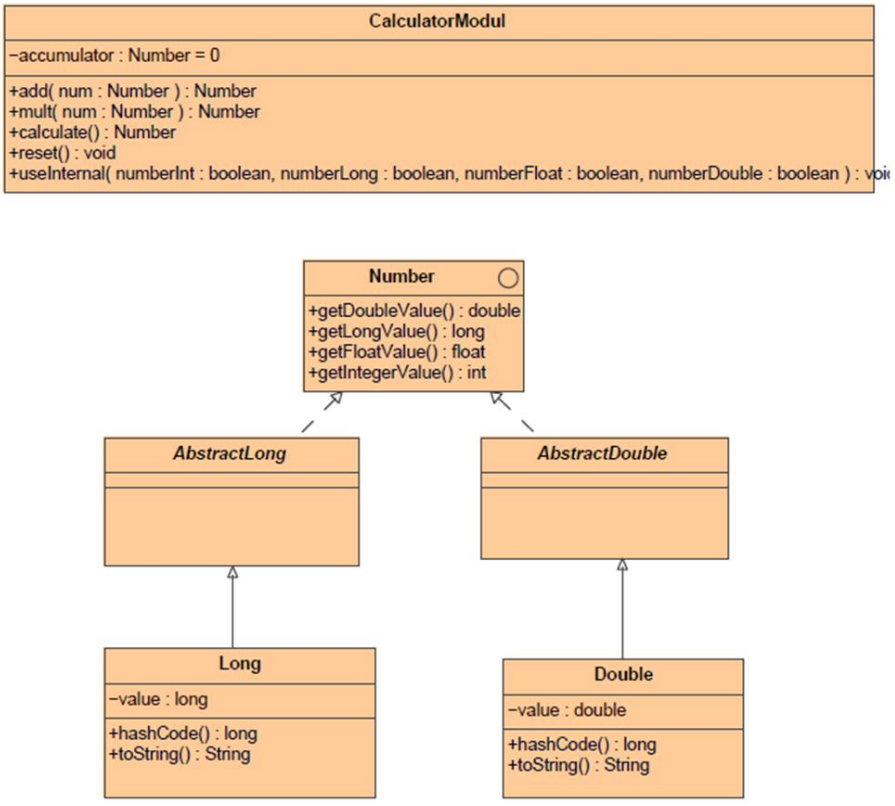

Figure 4.9: Example UML Model Smell Speculative Generality [4]

4.11 Unnamed Element

Description: The model element, i.e. package, class, interface, data type, attribute, operation, or parameter, has no name. This smell summarizes corresponding smells such as Unnamed Class and Unnamed Attribute. According to the UML specification this is no misuse, i.e. the model is still valid. But on the one hand an unnamed element could lead to misunderstandings of the modeled aspect, on the other hand unnamed elements will cause problems during code generation in a model-driven process. However, a model element should be given a descriptive name that reflects its purpose.

Detection: This smell can be detected by matching a corresponding pattern based on the abstract syntax of UML.

Affected quality characteristics and goals: Unnamed model elements may reflect a real world aspect inprecise and incorrect. Furthermore, they might be harder to understand. Simplicity, Conformity, Precision, Consistency, Comprehensibility, Correctness, Completeness.

Example: In Figure 4.10 there are classes SoccerClub, Date, and Person related by several associations. Among others, there is an association between classes SoccerClub and Person but without any names, neither an association name nor corresponding role names. Here, it is very hard to understand the meaning of the association. Does it mean players, trainers, or even board members?

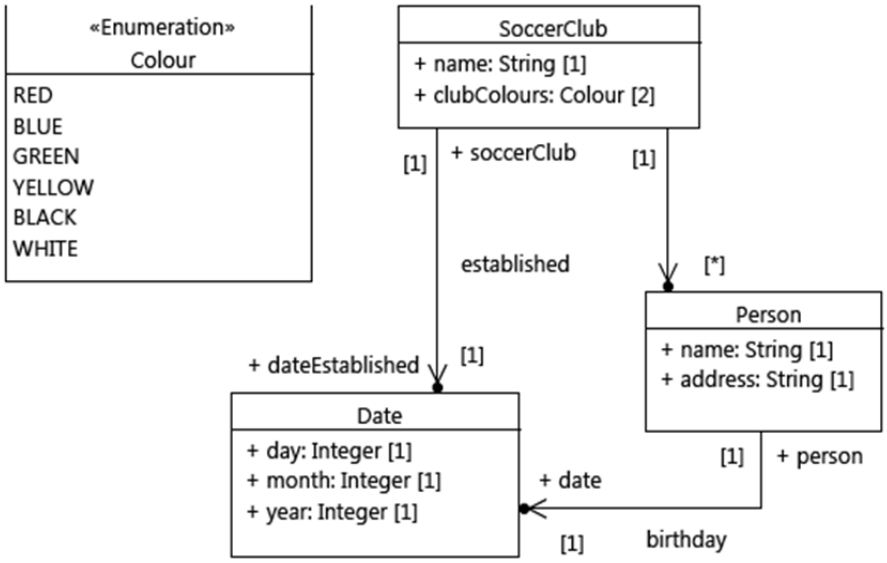

Figure 4.10: Example UML Model Smell Unnamed Element

4.12 Unused Element

Description: An unused model element is useless and indicates incorrect modeling. Either the element represents a valid domain object, i.e. there are missing relationships to further objects, or the modeler wanted to delete the element from the model but removed it only from the diagram. For example, an unused class has no child classes, dependencies, or associations and it is not used as parameter or property type.

Detection: This smell can be detected by matching corresponding patterns based on the abstract syntax of UML. This patterns have to be formulated in a way that the contextual element (class, for example) is not allowed to have any specific relationships to other elements. However, these patterns have to be defined specific to the considered contextual element type. Another way to define this smell is to determine specific metrics and to check whether these metrics are evaluated to zero each. For example, checking specific smell Unused Class requires model metrics Number of direct children, Total number of dependencies, Number of associated classes, Number of times the class is externally used as attribute type, and Number of times the class is externally used as parameter type.

Affected quality characteristics and goals: An unused model element may reflect an imprecise modeling. Precision, Correctness, Confinement.

Example: An example is given in the description of the smell : Unused Class.

FUTURE WORK

Since a manual model review is very time consuming and error prone, it is essential to automate the tasks as effectively as possible. We will implement tools supporting the included techniques metrics, smells, and refactorings for models based on the Eclipse Modeling Framework (EMF), a common open source technology in model-based software development. We will study the Eclipse plug-in EMF Smell supporting specification and detection of smells wrt. specific EMF based models. Some model smells are detectable by the existence of specific anti-patterns in the abstract model syntax. Other smells can be detected by metric benchmarks. For pattern-based model smells, EMF Smell uses the new EMF model transformation tool Henshin and OCL. Henshin is based on graph transformation concepts and uses pattern-based rules that can be structured into nested transformation units with well-defined operational semantics. Metric based model smells can be defined using metrics that are provided by EMF Metrics. Here, metrics can be specified in Henshin, Java, or OCL.

Smells can be categorized as pattern-based smells and metric based smells. For pattern-based smells we discuss the corresponding pattern rules formulated in Henshin or OCL. For metric-based smells we either present the Henshin pattern rule or a Java code snippet respectively OCL expression used for specifying the corresponding model metric.

REFERENCES

[1]. Thorsten Arendta, Matthias Burhennea, Gabriele Taentzera Defining and Checking Model Smells: A Quality Assurance Task for Models based on the Eclipse Modeling Framework .
[2]. Cédric Bouhours, Hervé Leblanc, and Christian Percebois Bad smells in design and design patterns.
[3]. Master's Thesis; Ruben Weilman Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations.
[4]. Thorsten Arendt, Gabriele Taentzer. UML Model Smells and Model Refactorings in Early Software Development Phases, Philips and Marburg University,2010.
[5]. Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple. Definitions and Approaches to Model Quality in Model-Based Software Development - A Review of Literature. Information and Software Technology, 51(12):1646 1669, 2009.
[6]. Thorsten Arendt, Gabriele Taentzer, Implementation Details of Smells and Refactorings for UML Models within the Eclipse Modeling Framework, Philips and Marburg University,2011.
[7]. Thorsten Arendt, Matthias Burhenne, and Gabriele Taentzer. Defining and Checking Model Smells: A Quality Assurance Task for Models based on the Eclipse Modeling Framework. In 9th edition of the workshop Belgian-Netherlands software evolution seminar (BENEVOL 2010),pages 50{54,2010.