Catalog of Model Smells

Authors: Erki Eessaar and Ege Käosaar

Ver 1.0 (March 6, 2018)

Introduction

The following catalog has been created by translating the code smells from the seminal book of Robert C. Martin.

Martin, R. C., 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Pearson Education.

The exception is the smell G0 that suggests remembering model history and that does not have a counterpart in the code smells of Martin (2009).

The presence of one or more code smells of Martin (2009) in a program code is a sign of technical debt that makes understanding, maintaining, and extending the code more difficult. It does not automatically mean that the program code contains bugs but it makes it more probable because, for instance, it is more difficult to notice bugs in a badly presented code. Similarly, the presence of one or more model smells in a model is a sign that the representation of the system in terms of models carries problems that make it more difficult to understand, maintain, and extend the models. It does not automatically mean that the model presents false claims about the system or the models do not reflect the requirements of clients. Nevertheless, it makes such mistakes more probable because, for instance, it is more difficult to notice these during reviews.

The translation was conducted as a mental exercise by the authors to see the extent of similarities of flaws that modelers and programmers can have in their work results. The translation does not mean translating the smells word by word but rather translating the *idea*. In mathematical terms, translation function *f_translate* from a set *CodeSmells* to a set *ModelSmells* is surjective (Wikipedia), meaning that for every element *m* in the codomain *ModelSmells* of *f_translate* there is at least one element *c* in the domain *CodeSmells* of *f_translate* such that f_translate(c) = m. There are elements of *CodeSmells* that do not have a counterpart in the *ModelSmells*. The function *f_translate* may map more than one element of *CodeSmells* to the same element of *ModelSmells.*

Just like in the book of R.C. Martin, the smells have been divided into classes but the classes are different. Each smell belongs to exactly one class. There are four classes of model smells in the catalog.

- General smells of models and diagrams (it is denoted with the letter “G”).
- Smells about the style of models and diagrams (it is denoted with the letter “S”).
- Smells about the naming of models, diagrams, and their elements (it is denoted with the letter “N”).
- Smells about testing models and testing modeled systems based on models (it is denoted with the letter “T”).

In each class, we have tried (sometimes unsuccessfully) to present smells that are more general before the smells that are more specific.

Compared to the free-form presentation of code smells of Martin (2009), we use a more structured presentation. In this sense the presentation is more similar to the presentation of antipatterns (Karwin, 2010), (El-Attar and Miller, 2010). Each smell is presented by using the following structure. The structure contains mandatory and optional sections. If there is nothing to write under the optional section, then its title is omitted from the smell.

- **Short identifier**, which contains the identifier of the class of smells and long identifier i.e. **name**. Short identifiers are used for cross-referencing and long identifiers form a vocabulary that can be used to characterize models and their representation (diagrams).
- Reference to the **code smells** based on that the smell has been defined.
- **General idea** (the core) of the smell.
- **Related model smells** (Optional). Together these smells describe a family of problems. If one discovers a smell in a model, then it is quite probable that its related smells are present as well. We refer from the more specific model smells to the more general model smells but not vice versa in order to decouple the smells and allow us using more general smells without having to consult with the more specific smells. It corresponds to the suggestion of Martin (2009) in the smell "Base Classes Depending on Their Derivatives" that "In general, base classes should know nothing about their derivatives." Some model smells are at the same level of abstraction and in this case we consider these as peers and cross-reference these. For instance, there are smells about not following modeling style conventions and model element naming conventions. We consider these smells to be at the same level.
    - NB! Short identifiers of related smells are hyperlinks in the document!
- **Scope**. Depending on the used modeling language, diagrams might not be classified as model elements. For instance, UML 2.5 defines diagrams as "graphical representations of parts of the UML model". According to the UML definition a diagram contains graphical elements that represent elements in a UML model. However, there are smells that apply only to model elements, only to diagrams, or both. To make this information explicit this section can have values: "Model elements", "Diagrams", and "Model elements + Diagrams". We see here an analogy with the database world. Diagrams and model elements have the same relationship as database views and base data structures in a database. Database views are database objects and in some cases (for instance, naming and avoiding exact duplicates) should follow the same design rules as base data structures. On the other hand, there are also differences. For instance, in case of base data structures of operational databases, we should try to reduce data redundancy within and across data structures as much as possible (see the database normalization theory and the orthogonal design principle). On the other hand, values of views are automatically calculated based on the values of base data structures and having some data redundancy within or across views is not a problem because it is a controlled redundancy. Just like database views are a part of a database, diagrams should be considered a part of the model that they help to represent. To summarize, the first version of smells is influenced by UML, considers diagrams as

parts of models but does not count diagrams as model elements i.e. we write about model elements *and* diagrams.

- Explanation in general terms of how to **refactor** models to remove the smell. The section might describe multiple ways of refactoring in case of the same problem.
- **Examples** of the smell that refer to specific model types. The examples are based on the experiences of authors.
- References to the **already documented model smells** that address the same problem (Optional). The smells in this catalog are defined in very general terms. It turns out that the investigated literature does not offer smells that describe the problems exactly at the same generality level. Instead the smells from other sources are examples of the smells in case of specific model types. Each referenced smell is an example for zero or more smells in our catalog. The section contains references to the smells from the following sources.
    - UML model smells in the early software development phases (Arendt and Taenzer, 2010). The smells are about class diagrams, use case diagrams, and state machines. There are 17 smells.
    - Antipatterns of use case models (El-Attar and Miller, 2010). There are eight antipatterns.
    - Smells of process models (Weber et al., 2011). There are eight smells.
    - Information modeling smells that can appear in conceptual data models (McDonald and Matts, 2013). There are seven smells.
    - Model smells of MATLAB/Simulink models (Gerlitz et al., 2015). There are 21 smells.
    - Model smells in the models that are used to depict object-oriented programs. Misbhauddin and Alshayeb (2015) conducted literature review of UML model refactoring and present in Table 6 the list of investigated model smells. All the model smells have a corresponding object-oriented code or system architecture smell. Most of the smells are defined based on UML class diagrams. There are 27 smells.
- References of the model patterns, which oppositely to antipatterns refer to the best practices that can be used to refactor the models in order to eliminate the model smell (Optional). The section contains specifically references to the UML model patterns that are presented by Evitts (2000). Each referenced UML model pattern offers a way of refactoring in case of zero or more smells in our catalog.

Smell descriptions may contain references. Different smell descriptions may refer to the same material. Thus, to avoid duplication, we do not have an optional local references description in the end of each smell but a global reference list in the end of the catalog.

There is quite a lot of literature about model smells in design models, particularly in software class models. On the other hand, there is not much work about the possible smells in system analysis models. Most of the smells in the catalog are general enough to be applicable to any kind of software model. However, while writing this, we have had specifically in mind model

types that one can use during system analysis. Eessaar (2014) explains some of the model types. The examples of the smells are also based on the model types.

- Specification of subsystems and their relationships (could be, for instance, textual or a set of UML package diagrams).
- Activity model.
- Use case model.
- Conceptual data model that consists of entity-relationship model and textual specifications of entity types and their attributes. The entity-relationship model can be created based on UML class model.
- Domain model that representation consists of domain class model and textual specifications of domain classes and their attributes. The domain class model can be created based on UML class model.
- State machine model. In UML terms we are interested in protocol state machines that allow us to express lifecycles of main entity types.
- Contracts of database operations.
- System sequence model.

In case of model testing and model-based system testing we use the explanation of Easterbrook (2010) to distinguish between validation and verification. Smells that belong to the class "T" Model-related testing should include all the following.

- Model verification – whether a model is well engineered, including follows best practices and avoids the model smells.
- Model-based system testing.
    - Model-based verification of the system – whether the modeled system is well engineered, including follows best practices, avoids design and architecture smells, and conforms to the recorded requirements.
    - Model-based validation of the system – whether the modeled system actually fulfills the requirements of its clients and future users.

We agree with Martin (2009) that the catalog is not complete and possible can never be complete. There are endless possibilities to make things in a sub-optimal way. The translation was intended as a thought experiment in order to see the extent of correspondence between the principles of modeling and coding and also being able to communicate the similarities to others.

Table of Contents

General Smells of Models

G0: Not Remembering the History

Code smells: -

General idea: One has to modify a model but is later unable to recall how the understanding of the system and presentation of this understanding has evolved over time. One is unable to justify why the system currently is structured and behaves like it does. One cannot confirm as to whether the modeled system follows the patterns of software evolution (Lehman, 1996) and database evolution (Vassiliadis, 2017). One cannot start a fork of a model based on the model version of specific time in history. One cannot restart the modeling process based on the model version of specific time in history.

Scope: Model elements + Diagrams

Refactoring: Keep model versions (including diagrams) to be able to make flashback queries about the earlier descriptions of the system as well as for forking or restarting the work. This is, for instance, a very useful material for investigating the evolution of the system and understanding of the system. Save models in a format that you will be able to use in the future. Either keep the modeling software or periodically save models in new formats without losing information that they carry. Use a versioning system or design and implement a catalog structure, naming system of files and folders, a process of keeping the history, and a process of backing it up.

Examples:

- Any model that needs modification.

G1: Metadata about Models in a Wrong Place

Code smells:

- C1: Inappropriate Information

General idea: One or more claims about a model (for instance, the authors, the last time of modification, known issues) are kept in a place that is not dedicated to recording and presenting these and where it is easy to miss these and forget to update these. For instance, these claims are presented as comments on diagrams or within textual documents. Big number of comments clutter diagrams. The same comment element may contain information that have different purposes (present metadata, give some background of the modeled system etc.). Such comments violate the separation of concerns principle.

It is easy to break the link between the comment and its target.

- It may be that a model is within a container (for instance, a package) and information about the model or some of its elements is presented as a comment on the diagram that is in the container. If the diagram and model elements are separated to different containers, then the link between the comment subject and the comment is lost or at least it is more difficult to grasp the link by the readers.
- It may be that information about a model element is presented as a comment on a diagram. If a representation of the model element is removed from the diagram, then depending on the comment the link between the comment subject and the comment is lost or at least it is more difficult to grasp the link by the readers.

Related model smells:

- G0 – the model might contain comments with change history because there is no system in place to keep old versions of models.

Scope: Model elements + Diagrams

Refactoring: If you use a versioning system, then use it as a single, unambiguous, authoritative source of the information about the authors and modifications. If you do not use a versioning system but use a CASE tool, then if possible keep the metadata as a part of specifications of model elements and do not represent it within models as comments on diagrams.

Keep information about the issues that have to be fixed in a model in an issue tracking system (a separate system or integrated with the CASE tool) or in a dedicated log.

Examples:

- Information about the authors and the time of last modification of entity relationship model as a comment on a diagram that was created by using a CASE tool.
- Information about the mistakes that have to be fixed as a comment on a diagram that was created by using a CASE tool.
- Information about the authors and the time of last modification as a part of the structure of a description of a use case or a contract of a database operation.

- Information about the known mistakes as a part of the structure of a description of a use case or a contract of a database operation.

Model patterns for refactoring (Evitts, 2000):

- 8.6 Components Manage Change
- 8.7 Configured and Released Packages

G2: Expired Model Element

Code smells:

- C2: Obsolete Comment
- C5: Commented-Out Code
- G9: Dead Code

General idea: A model that represents the current understanding of the modeled system contains at least one claim about the real world that at the present time is not true because it is expired (out-of-date). Thus, the model gives a wrong impression about the current state of the modeled system. In case of models of software/information systems these claims could be about real-world objects (M0-layer of the Meta-Object Facility (MOF)) or about their generalized structure or behavior (M1-layer of MOF).

Related model smells:

- G0 – the model might contain expired elements because there is no system in place to keep old versions of models. Thus the expired elements are kept in the current model version just in case.

Scope: Model elements

Refactoring: It is inevitable that systems evolve and our understanding of the systems evolves as well. Remove all the expired model elements from the model or modify these so that they represent a current view of reality again. If you remove an element by deleting it from a diagram, then pay attention that you drop it from the model not just remove the representation from the diagram.

Examples:

- An obsolete comment.
- A subsystem that does not exist anymore according to the business architecture.
- A sequence of actions in an activity model or system sequence model that is not executed any more due to the changes in the process.
- A commented out section of a textual model like use case model or contracts of database operations.
- An attribute of an entity type in a conceptual data model or a domain class in a domain model that corresponding data is not needed by the system any more.
- A constraint to the values of an attribute that does not hold any more.
- A state or state transition that does not describe the current view of the lifecycle of an entity type.

G3: Stating the Obvious

Code smells:

- C3: Redundant Comment

General idea: A model contains at least one model element that presented claims are so obvious to the intended audience (axioms) that it is unnecessary to explicitly present these. In case of models of software/information systems these claims could be about real-world objects (M0-layer of the Meta-Object Facility (MOF)) or about their generalized structure or behavior (M1-layer of MOF).

Scope: Model elements + Diagrams

Refactoring: Be cautious and conservative in deciding that something is obvious. If the element should be in the model, then add information or remove the obvious information without removing the element. Otherwise remove the redundant element. If a container of claims (for instance, a comment element, a diagram or the entire model) bundles together multiple claims and the result of removing claims is an empty container, then remove the container as well.

Examples:

- A comment on a diagram that states that it is a diagram and perhaps also names the diagram type and used modeling language.
- The name "uses" on a dependency between a model element (for instance, a package) that depicts an area of competence and a model element (for instance, a package) that depicts a functional subsystem.
- The name "is followed by" or "control flow" of a control flow element in an activity model.
- A comment on an activity diagram that the participating actors must have privileges to perform the actions.
- A triggering event of a use case that just states that the main actor of the use case "wants to start" the use case.
- A comment about an aggregation on an entity-relationship diagram or a domain class diagram that it represents an aggregation.
- The role name that repeats entity type name in the association of a conceptual data model.
- The name "is connected" of an association or "aggregation" of an aggregation in a UML class model.
- A definition of an entity type that repeats the name of the entity type or uses its synonym.
- A precondition in the contract of a database operation that there should be data structures in the database that support registration of the data to which the contract refers to.
- A postcondition in the contract of a database operation that states that data must satisfy all the declared constraints.
- A guard condition on a state transition of a state machine model out of the state S that declares that an entity must be in the state S.

- A guard condition of a message in a system sequence model that states that the invoker must have necessary privileges to perform the operation.

G4: Poorly Written but Necessary Textual Model Element

Code smells:

- C4: Poorly Written Comment

General idea: A model contains at least one textual claim that presentation has bad grammar, incorrect punctuation, uses wording that is unclear to some model consumers, is not well-structured, rambles, or is too long.

Related model smells:

- G3 – it might be that you state the obvious.

Scope: Model elements

Refactoring: Know your audience and use terminology that should be known to a typical reader. Rewrite the text. Write text that is as long as needed but as short as possible. Be precise and concise, follow grammar and punctuation rules. The reader of the text must know more after finishing its reading instead of being sorry about the wasted time.

Examples:

- A comment that consists of acronyms.
- Bad grammar in an extended format use case.
- An extended format use case that is formatted as one long paragraph instead of using newlines and other means to stress the structured nature of this specification.
- Too long definition of an entity type or domain class that in addition to relevant aspects describes aspects of it that are irrelevant for understanding the essence of the entity type in a particular system.
- Incorrect punctuation in a pre- or postcondition of a contract of database operation.
- A pre- or postcondition of a contract of a database operation uses synonyms of the names of the entity types. It makes it difficult to mentally link the contracts with conceptual data model.

G5: Model Elements or Diagrams That Are Not Needed by Any Other Model Elements

Code smells:

- F4: Dead Function

General idea: A model contains at least one model element that depicts a part of the system that according to the model is not needed by any other part of the system. A model contains irrelevant comments that are not needed to understand the system. Consequently, the model element (including comment) is not needed by any other element of the models that describes the system. A model contains an empty diagram i.e. the diagram is not needed to express or describe any model element.

Related model smells:

- G2 – it might be that some element is not needed because view to the system has changed and the element has expired.

Scope: Model elements

Refactoring: Remove all the unnecessary model elements from the model or modify these so that they again represent a current view of reality. Modification may include modifications of other model elements so that they start to reference to the element in question.

Examples:

- An empty diagram (activity, use-case, system sequence, entity-relationship, design class, state transition). The diagram does not express any model element, including a comment.
- A visual model element that is not presented on any diagram but nobody is missing the element. Störrle (2003) gives an example that such element may be created during model cloning through copy/paste. A modeler may want to create a clone on purpose. It may happen that a modeler wants to undo the creation but only deletes the element from a diagram and not from the model.
- A functional subsystem that is not used by any area of competence.
- An action in an activity that does not have the incoming control flow.
- A use case that is not connected with any actor and that is not a subfunction-level use case to any other use case.
- An entity or an attribute in conceptual data model that corresponding data is not needed by any use case and is consequently not read and modified by any database operation.
- A domain model of a functional subsystem that is not a part of the system business architecture any more.
- A contract of a database operation that is not referenced from any use case.
- A state machine model that describes the lifecycle of an entity type that is not specified by the conceptual data model any more.
- A system sequence model that describes a scenario of a use case that is not a part of the model any more

Already documented model smells

UML models (Arendt and Taenzer, 2010)

- Unused Element
- Unused Use Case
- No Incoming

Process models (Weber et al., 2011)

- PMS6: Unused Branches

Information models (McDonald and Matts, 2013)

- #4 – no relationship

MATLAB/Simulink models (Gerlitz et al., 2015)

- Unused Signal

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- Lazy Class
- No Incoming Transitions
- Lava Flow

G6: Redundancy

Code smells:

- G5: Duplication
- G25: Replace Magic Numbers with Named Constants
- G28: Encapsulate Conditionals
- G35: Keep Configurable Data at High Levels

General idea: A model contains uncontrolled redundancy. By rephrasing the definition of data redundancy (Date, 2006), a model contains at least two distinct representations, either direct or indirect, of the same or considerably similar claim. Each claim is expressed by a model fragment that consists of one or more model elements. Störrle (2013) observes in the example of UML classes that each model element contains zero or more model elements (attributes, operations, association ends etc. in the example of classes). The problem is uncontrolled redundancy i.e. modelers not the modeling tool has the task to propagate the updates. Definition of data redundancy (Date, 2006) points only to the identical i.e. duplicate claims. Moreover, in the context of modeling a redundancy could also appear in terms of claims that have a high degree of similarity (Störrle, 2013).

Comparison of two values v1=v2 should evaluate to TRUE if v1 and v2 are the very same values, regardless of their internal (physical) representation, which is hidden from the user (Date and Darwin, 2014). Similarly, decision as to whether two model fragments or two diagrams are equal or highly similar should be made based on their user-visible components. It should not be made based on hidden system-defined identifiers and also not based on the metadata (for instance, authors, last modification), which is by default not visible to the user and quite probably has been assigned automatically by the modelling software.

The claims could be either predicates (truth-valued functions) or propositions (predicates with no parameters). In case of models of software/information systems these claims could be about real-world objects (M0-layer of the Meta-Object Facility (MOF)) or about their generalized structure or behavior (M1-layer of MOF). For instance, we can think about a UML class diagram (belongs to the M1-layer) as a representation of a composite predicate and a UML object diagram (belongs to the M0-layer) as a representation of a composite proposition that corresponds to the predicate. Similarly, a class and an object in the model are a predicate and proposition, respectively.

Related model smells:

- G1 – it could be that information is in a wrong place, modelers do not notice it, and they duplicate the information in other places.
- G2 – not removing expired elements may cause a situation that at some point the once expired element describes again the current state of the system. However, a modeler does not notice the old element and creates a new element.
- G5 – if an element is not needed by any other element, then a modeler may easily forget the existence of the element in the model. If one has to use the element again, then it may lead to the creation of the element from scratch instead of using the existing element.

Scope: Model elements + Diagrams

Refactoring: Having redundancy in a model means that the model contains combinatorial effects i.e. changing a model element requires changing other model elements as well as the ripple effect (Mannaert et al., 2012). The amount of work for propagating updates and the probability of introducing inconsistencies or propagating mistakes as the result increases with the size of the model. Such model does not conform to the rules of normalized systems and is thus not highly evolvable.

Duplications in models are known as "model clones" (Störrle, 2013). Identifying types of duplication in software artifacts, automating their detection, and working out means to remove these are important research topics (Rattan et al., 2013).

It is impossible to remove some duplications. Störrle (2013) points to the loopholes in modeling languages (in the example of UML) that make the existence of some duplication in models almost inevitable. It depends on the used modeling language. For instance, it may be impossible to define a comment at the model level and then link it to multiple diagrams.

Some duplications are not harmful if treated with care. These are introduced on purpose by modelers. For instance, modelers may want to describe competing alternatives that have some semantic overlap (Störrle, 2013). Another example is that modelers duplicate a part of the model in order to use the duplicate as the basis for describing another part of the system. LaToza et al. (2006) call these fork clones. There are duplicates until the work is not finished. Yet another example are model elements that representation is duplicated on multiple diagrams in order to allow model viewer to mentally connect different diagrams. In the latter case the details of the element should be visible only on one diagram and in other diagrams it should be represented as a placeholder/link with no internal details.

Preventable and undesirable duplication has different forms (Sonmez, 2012), (Störrle, 2013), (Rattan et al., 2013) and the ways of dealing with it depend on the duplication type. A necessary step before removing the following duplications is making sure as to whether the duplication is created on purpose or not.

- Exact clones
    - **General idea**: A model contains two or more instances of identical model fragments or diagrams.
    - **Refactoring**: Retain only one instance. Refactor other elements of the same model and other models so that their elements refer to the remaining instance.
    - **Examples**:
        - Two or more exactly the same packages (together with their content) that describe the same subsystem.
        - Two or more exactly the same use case descriptions. For instance, it could be that use case "Identify users" that represents a crosscutting concern has been described in the context of different functional subsystems instead of describing it only once and referring to it from different places.
        - Two or more exactly the same state machines.
        - Two or more exactly the same classes that represent an entity type in different packages that represent different registers. For instance, entity type Classifier is duplicated in the packages of multiple registers.

        - Exactly the same attributes of a supertype and its subtype in a conceptual data model (Arendt and Taentzer, 2010).
        - Two or more exactly the same entity relationship diagrams.
        - Two or more exactly the same parameters of a database operation (Arendt and Taentzer, 2010).
- Renamed clones
    - **General idea**: A model contains two or more model fragments that elements differ only by their name. A model contains two or more diagrams that differ only by their name. The names may be in the different natural languages.
    - **Refactoring**: The same as in case of exact model clone
    - **Examples**:
        - Two or more packages that represent subsystems. The packages differ only by their name but not by their content.
        - Two or more use cases descriptions that differ only by the name of use case.
        - Two or more contracts of database operations that differ only by the name of the operation.
        - Two or more entity relationship diagrams that differ only by the name of the diagram.
- Clones with different appearance
    - **General idea**: A model contains two or more model fragments that consist of textual elements and the fragments differ only by the order of elements or by the order of the subsections of the elements. A model contains two or more diagrams that differ only by the layout.
    - **Refactoring**: The same as in case of exact model clone
    - **Examples**:
        - Two or more use case descriptions with different order of subsections.
        - Two or more sets of extended format use cases that differ only by the order of the use cases.
        - Two or more contracts of database operations that differ only by the order of pre- or postconditions.
        - Two or more entity relationship diagrams where the position of elements on the diagram are different but elements themselves are exactly the same.
- Language clones
    - **General idea**: A model contains multiple textual model fragments that differ only by the used natural language but not by the content that was expressed by the language. On the other hand, two diagrams of the same subject matter but with different notations is not a problem if the diagrams are based on the same model elements and translation to different notations is performed by the modeling software on the fly. This would be an example of Dont Repeat Yourself design principle. For instance, on one class-diagram based entity relationship diagram the UML notation is used but on another the IDEF1X notation is used.
    - **Refactoring**: If the entire audience has ability to understand one particular language, then use the language and the refactoring is the same as in case of exact model clone. Otherwise you have to retain the duplicates but should designate the primary copy where the modifications are made and that are at some later point propagated to other copies.

    - **Examples**:
        - Two or more descriptions of the same use case written in different languages.
        - Two or more contracts of the same database operation written in different languages.
- Clones that use different model types.
    - **General idea**: The same system aspect for the same perspective is described by using different model types so that the descriptions overlap.
    - **Refactoring**: Although it may be viewed as an inevitability there is still duplication. A solution would be view-driven software engineering that is proposed by Atkinson and Draheim (2013). Different types of models would be views on top of the Single Underlying Model (SUM). It means that they would be generated on demand from the single underlying information source and all the updates made in the generated models must be propagated back to the source. A more short-term solution is to critically review the model types that are required in a project and reduce the number of model types in order to avoid duplication.
    - **Examples**:
        - The same process is modeled as an extended format use case, activity model, and a system sequence model.

The following duplications may refer to the possibilities to make generalizations and increase abstraction. This increases the vocabulary used in models and allows modelers to see a similarity between concepts that at the first glance seem to have no overlap. It also contributes towards reusing model elements. These correspond to the semantic clones (Rattan et al., 2013). One could say that they characterize unfinished models.

- A domain model or conceptual data model that is based on the UML class model contains multiple classes that share some attributes or relationships.
  Refactoring: Define a class that represents a more general concept and use generalization relationship to connect it with the subclasses. Move the duplicated attributes, associations, aggregations, and compositions of all the subclasses to the superclass. Keep the subclasses to make the generalization hierarchy explicit and not lose information about the modeled world.
- A process model (for instance, activity model, scenarios in the extended format use cases) contains multiple decision points with the same Boolean expression.
  Refactoring: Encapsulate the repetition as a separate action or subfunction-level use case and refer to it from where it is necessary.
- Multiple process models (for instance, activity model, scenarios in the extended format use cases) describe a similar algorithm.
  Refactoring: Consider defining a more generalized version of the process description from where you refer to descriptions of sub-processes that are specific to some situation.
- Multiple process models (for instance, activity model, scenarios in the extended format use cases) describe the same data modification or query operation.
  Refactoring: Specify data reading and data modification operations by using contracts and refer to these contracts from different process models.
- A state machine contains two or more states that can be generalized to a superstate.
  Refactoring: Define a superstate but also retain the substates. Define state transitions that are common to all the substates at the level of the superstate.

Examples:

- There is a Boolean expression that is used in case of multiple decision points of activity model.
- Multiple activity models describe a similar algorithm.
- The same entity type has been defined in different packages.
- The same attribute has been defined in the context of different domain classes.

Already documented model smells:

UML models (Arendt and Taenzer, 2010)

- Attribute Name Overridden
- Data Clumps
- Multiple Definitions of Classes with Equal Names

Process models (Weber et al., 2011)

- PMS3: Redundant Process Fragments

MATLAB/Simulink models (Gerlitz et al., 2015)

- Duplicate Model Part
- Redundant Signal Paths
- Duplicate Signal in Bus

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- Duplication
- Data Clumps
- Extraneous Adjacent Connector

G7: Mixing Different Abstraction and Decomposition Levels

Code smells:

- G6: Code at Wrong Level of Abstraction
- G7: Base Classes Depending on Their Derivatives
- G34: Functions Should Descend Only One Level of Abstraction

General idea: Modelers create models with different levels of abstraction in order to explain the modeled system from different perspectives. The rows of the Zachman Framework (Zachman, 2017) represent these perspectives. The higher the row is in the framework, the more abstract system view it represents. Different perspectives have different corresponding interested parties and their corresponding models are used for different tasks. To explain the same perspective, one may create models with different levels of decomposition (detail). The described situation may lead to the following problems.

- An explanation of the system from an audience perspective (a row of the Zachman Framework) mixes models that correspond to the different perspectives.
- A model of an aspect of the system (a column of the Zachman Framework) mixes elements from the models of the aspect that are created for different perspectives.
- A model mixes elements from the different levels of decomposition.

Similar problems can occur with diagrams, even if their underlying models are at the suitable level of abstraction and decomposition. It is possible to represent elements from different models (and thus from different abstraction and decomposition levels) together on the same diagram.

Related model smells:

- G1 – mixing different levels of abstraction or different levels of decomposition within the same model means also that some information is presented in a wrong place.

Scope: Model elements + Diagrams

Refactoring: "Good software design requires that we separate concepts at different levels and place them in different containers" (Martin, 2009). Create models and diagrams that do not mix different levels of abstraction and different levels of decomposition. The elements of each model or on each diagram should belong to the same decomposition level of a system that is one level lower than the level referenced by the name of the model or diagram. For instance, we have an entity-relationship model with the name "Register of persons" that represents a conceptual view of the register.

Examples:

- A package diagram that depicts subsystems and their relationships contains details (in addition to names) of use cases that elaborate functional subsystems or entity types that elaborate registers.
- A system analysis use case model mixes summary-level, user goal-level, and subfunction-level use cases. Cockburn (2001) suggests these types of use cases.
- A comment of a conceptual data model describes indexes that designers should create to speed up certain queries.

- A conceptual data model describes some entity types and some database tables.
- An entity-relationship diagram (a part of conceptual data model) presents both registers (data-centric subsystems of a system) and entity types or entity types and tables that implement the entity types.
- A conceptual data model describes surrogate keys and foreign keys in case of entity types. These are design concepts that are related with the design of a SQL database.
- A class in a domain model contains an operation, i.e. the class represents a software class not a domain concept.
- An analysis-level contract of a database operation refers to database tables instead of entity types.
- An analysis-level contract of a database operation contains comments about improving performance of the stored procedure that would be created to implement the operation. The latter is design question and depends on the used software and hardware platform.
- A contract of a database operation describes things that happen outside the database (for instance, in the user interface or in the physical world).
- The effect in a state transition of an analysis state machine model describes software implementation on a particular platform.
- A system sequence model shows internal dealings of the system, i.e. does not treat the system as a black box.

Already documented model smells:

Use case models (El-Attar and Miller, 2010)

- a7. Multiple Actors Associated With One UC (rationale 3)

Model patterns for refactoring (Evitts, 2000):

- 5.2 Implementation or Representation
- 6.1 Domain Model is Essential
- 6.2 Actors Play Essential Roles
- 6.3 Factor the Actor
- 6.4 Essential Actions
- 6.5 Essential Vocabulary
- 6.6 Objectify Internal Roles
- 6.7 To Be Model
- 6.8 As Is Model
- 7.1 Manageable Product
- 7.2 Product Stakeholders Are Model Clients
- 7.3 Product Events in Context
- 7.4 Use Cases Represent Requirements
- 7.5 Boundary-Control-Entity
- 8.1 Separation of Concerns
- 8.2 Whole Components

G8: Inconsistencies of Representation

Code smells:

- G11: Inconsistency

General idea: Similar things within or across models or diagrams are done in different ways.

Related model smells:

- G6 – redundancy leads to inconsistencies.
- G9 – high coupling makes it more difficult to notice inconsistencies.
- S1 – too big diagrams make it more difficult to notice inconsistencies.
- G22 – multiple languages in a model increase the need of context switches of the modelers and model testers and thus increase the possibility of inconsistencies.

Scope: Model elements + Diagrams

Refactoring: Establish a convention of doing things (preferably by the consensus of all the modelers) and follow it.

Examples:

- The same color or shape of representations of model elements means different things on different diagrams.
- Some names of functional subsystems contain the word "management" and some do not.
- Some use case names use active voice and some passive voice.
- Some use case name starts with a strong verb and some not.
- Some use case names have an uppercase letter at the beginning of the first word, some use case names have the an uppercase letter at the beginning of each word, and some use case names use only lowercase letters.
- Use case names in an use case diagram and in a textual specification of use cases use uppercase and lowercase letters differently.
- Different extended format use case descriptions have different sections (parts of structure).
- The use of synonyms in different models. For instance,
    - A large part of the business architecture of information systems consists of functional subsystems and registers. It is possible to find these based on the main entity types i.e. each main entity type has a corresponding functional subsystem and register (Eessaar, 2014). In this case the name of the functional subsystem and the register should be derived from the name of the entity type. However, it could be that the names of the subsystems are derived from a synonym of the entity type. For instance, there is "Product management functional subsystem" in the business architecture and "Merchandise" as the main entity type. It could also be that the names of different types of subsystems refer to the synonyms of their corresponding main entity type. For instance, there is "Product management functional subsystem" and "Register of merchandise" that correspond to the main entity type "Stock".
    - The names of actors are different on a use case diagram and in the textual descriptions of use cases.

    - Contracts of database operations refer to the same entity type by using synonyms of its name.
- On some entity-relationship or domain class diagrams more general entity types / domain classes are placed to the upper part of the diagram and on some diagrams to the lower part.
- In case of a conceptual data model some but not all registers are represented as packages within the model.
- In some state machine models the triggering events of state transitions mention an actor and in some state machine models they do not.
- Some names of system events in system sequence models start with a verb and some are not.

Already documented model smells:

UML models (Arendt and Taenzer, 2010)

- Inverted Operation Name

MATLAB/Simulink models (Gerlitz et al., 2015)

- Inconsistent Port Name
- Inconsistent Interface Definition

G9: High Coupling

Code smells:

- G14: Feature Envy

General idea: Elements of the modeled system depend on each other too much. The element in a container should be interested in the elements of the container they belong to, and not the elements of other containers. Exploring models of highly coupled systems requires extra effort, because we have to look together more models and bigger models than in case of a system that features low coupling.

Scope: Model elements

Refactoring: Reduce the dependencies between the system elements by trying to apply the design theorems of normalized systems (Mannaert et al., 2012). Reorganize models accordingly. In case of use case model do not use inclusion dependency between use cases to show the order of their execution. Use pre- and postconditions of use cases to decouple these from each other so that they do not have to know about each other and to implicitly indicate their order of execution (Gallen, 2012). Decompose comments into smaller comments, each of which has a separate task.

Examples:

- A functional subsystem of a system needs most of the registers (data centric subsystems) for functioning.
- A process depicted in an activity model, system sequence model, or in an extended format use case has to invoke multiple sub-processes. In this way, one may use functional relationships between processes to represent sequential dependencies (Gallen, 2012).
- A database operation makes changes in multiple registers.
- Lifecycles of different entity types, which can be expressed with the help of state machine models, depend on each other a lot.

Already documented model smells:

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- God Class (Blob)
- Swiss Army Knife
- Too Strong Coupling
- Message Chains
- Feature Envy

Model patterns for refactoring (Evitts, 2000):

- 5.8 Model the Seams
- 5.11 Opaque Packages
- 7.3 Product Events in Context
- 7.5 Boundary-Control-Entity

G10: No Clear Focus

Code smells:

- G12: Clutter
- G13: Artificial Coupling
- G36: Avoid Transitive Navigation

General idea: A model element or a diagram does not have a clear focus. Its human consumer has difficulties in understanding its main message. Thus, it violates the separation of concerns design principle, which is among other places required in the normalized systems (Mannaert et al., 2012). Therefore, if one wants to modify a model element or a diagram, then one has to make cascading changes due to the combinatorial effects. The bigger the modeled system is, the more changes one has to make and the more difficult it is to evolve the models and the system.

Related model smells:

- G1 – information in a wrong place means that the information has been unnecessarily coupled with model elements, violating the separation of concerns principle.
- G2 – expired model elements present unnecessary information in the context of the current understanding of the system.
- G6 – if the same information is repeated in multiple model elements or on multiple diagrams, then it means redundancy. In most of these places the information is unnecessary. Thus, it is better to represent it in one place and refer to it if needed.
- G7 – model or diagram elements that are at the wrong abstraction or decomposition level mean artificial coupling. These elements should belong to different models or diagrams that serve different purposes.
- G9 – an information that at the first glance seems unnecessary might be actually necessary and a sign of high coupling.

Scope: Model elements + Diagrams

Refactoring: Change the element or diagram so that it has only one task. If representing an element e in the context of an element e' is needed to better understand e', then the representation of e in the context of e' should be minimal that is needed in order to understand e'. Usually an identifier of e is enough. It should act as a reference to the detailed description of e. For instance, in the extended format description of a use case the scenario description should contain identifiers of database operations not duplication of their content. Another example is that each entity type may appear on multiple entity relationship diagrams but its attributes and constraints should be visible on only one of the diagrams – a diagram of the register that contains the data corresponding to the entity type. On other diagrams its representation is scaled down (only name) and it is presented in order to show relationships.

Examples:

- A package diagram that should show what registers are needed by a functional subsystem presents all the registers of the system, including the ones that the functional subsystem does not need.
- An activity model or system sequence model presents multiple weakly related or unrelated processes, perhaps also repeating these on multiple diagrams.

- A use case diagram of a functional subsystem displays some use cases that belong to another subsystem and are not related with the current subsystem in any way.
- A use case description in the extended format describes changes in the database that happen as the result of going through the scenarios of the use case instead of referencing to the needed database operations.
- An entity-relationship diagram of a register r presents a detailed view (attributes, constraints) of an entity type that does not belong to r but has at least one direct relationship with an entity type that belongs to r.
- An entity-relationship diagram of a register r presents a detailed view (attributes, constraints) of an entity type that does not belong to r and does not have any direct relationships with any of the entity types that belongs to r.
- An entity type that actually combines multiple entity types in a conceptual data model.
- A conceptual data model has comments about functional subsystems and use cases that are used to represent these.
- A domain class diagram of a functional subsystem f presents a detailed view (attributes) of a class that does not describe f but has at least one direct relationship with a class that describes f.
- The effect of a state transition in a state machine model describes in a detailed manner resulting changes in the database instead of referencing to the needed database operations.
- A contract of database operations contains detailed description of processes that invoke it instead of just referencing to these.
- A contract of database operation describes an operation that has multiple tasks.

Already documented model smells:

UML models (Arendt and Taenzer, 2010)

- Data Clumps
- Large Class
- Long Parameter List

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- God Class (blob)
- Long Parameter List
- Too Low Cohesion
- Scattered Parasitic Functionality
- Connector Envy
- Ambiguous Interfaces

Model patterns for refactoring (Evitts, 2000):

- 4.3 Explicit Elision
- 5.3 Digestible Chunks
- 5.9 Packaging Partitions
- 5.11 Opaque Packages
- 6.1 Domain Model is Essential
- 6.2 Actors Play Essential Roles
- 6.3 Factor the Actor

- 6.4 Essential Actions
- 6.5 Essential Vocabulary
- 6.6 Objectify Internal Roles
- 6.7 To Be Model
- 6.8 As Is Model
- 7.1 Manageable Product
- 7.2 Product Stakeholders Are Model Clients
- 7.3 Product Events in Context
- 7.4 Use Cases Represent Requirements
- 7.5 Boundary-Control-Entity
- 7.6 Product Chunks Digest Easily
- 7.8 Use Cases: Work as Packages
- 8.1 Separation of Concerns
- 8.2 Whole Components

G11: Information in an Unexpected Part of a Model

Code smells:

- G17: Misplaced Responsibility

General idea: A model that contains textual part and visual part (diagrams) presents one or more claims about the modeled world in a part of the model that is not the best suited for it and is thus unexpected to the model consumer. In case of models of software/information systems these claims could be about real-world objects (M0-layer of the Meta-Object Facility (MOF)) or about their generalized structure or behavior (M1-layer of MOF). A program that generates a new artifact based on a model might not find the claims and thus the generation result is incomplete.

Related model smells:

- G6 – if for the sake of faster model reading the information is not moved to the best suitable place but is instead duplicated, then it causes redundancy.
- S3 – placing information to an unexpected part of a model is a violation of modeling conventions.

Scope: Model elements + Diagrams

Refactoring: For all the relevant information, which needs to be represented in a model, find the best place to represent it. Do not present in a visual/textual part of the model information that is better to represent in the textual/visual part of model. By following the principle of least astonishment (Wikipedia), put information to the place that is the least surprising to the model consumer. Do not duplicate information in the process.

Examples:

- A use case scenario is on a use case diagram instead of being in the textual description of the use case.
- Textual definition of an entity type or a domain class is on an entity-relationship diagram or domain class diagram, respectively, instead of being in a separate textual description.
- Definition of a constraint to the values of a single attribute is on an entity-relationship diagram as a comment. It would be better to represent it as a part of textual definition of attributes.
- Description of relationships between entity types is in the textual document instead of representing relationships in the entity relationship model.
- Information about constraints as a comment in a use case model.

Model patterns for refactoring (Evitts, 2000):

- 8.5 Specification Backplane

G12: Too Few Details

Code smells:

- G19: Use Explanatory Variables

General idea: An explanation of a concept or a process provides too few details. It is too vague, short, or general. In case of process descriptions, we get little or no information about its internal steps. Moreover, the description may use a vocabulary that is not meaningful and understandable to the model consumer.

Related model smells:

- G4 – lack of details in a textual model element is an example of poorly written but necessary textual model element.
- G7 – having a model element that provides little details may be a sign of mixing different abstraction levels within the same model.
- G13 – having a process description that provides little details may be a sign that the modeler does not understand the modeled process.

Scope: Model elements

Refactoring: High level descriptions have their place and consumers during the first steps of system development. However, as we proceed with the development, we must add details.

In case of textual descriptions of concepts use structuring of the text by focusing in each section to a certain aspect of the concept. Each section should have only one task. Use metaphors in order to make the concept better understandable. In case of process models replace vaguely-named general activities with sub-processes where individual actions have more meaningful names.

Examples:

- An activity model or a system sequence model that depict scenarios of a use case represents high-level activities.
- A use case is a summary level and does not go into the details of the depicted processes.
- A textual description of an entity type or a domain class is badly written. It is short, general, and does not provide information about the constraints to the data values that should be recorded about the entities that have the type. These constraints must always be satisfied and are derived from the business rules that guide the organization that will start to use the system.
- A state machine model shows high-level simple states. In a more detailed model some of these should be replaced with composite states or submachine states.
- A high level comment on a diagram stating that there is a mistake or refactoring opportunity but not explaining what is exactly the problem.

Already documented model smells:

Use case models (El-Attar and Miller, 2010)

- a7. Multiple Actors Associated With One UC (rationale 3)

Information models (McDonald and Matts, 2013)

- #6 – many to many (missing information)
- #7 – undefined functions (missing information)

Model patterns for refactoring (Evitts, 2000):

- 4.1 Attributes as Compositions to Types
- 4.5 Tombstone Packages
- 4.8 Dual Associations
- 4.9 Billboard Packages
- 5.4 Attach the Actor
- 5.5 Business Rules Invariably Constrain
- 5.6 Dynamic Object Types
- 5.7 Many-to-Many Class Trio
- 5.8 Model the Seams

G13: Not Understanding the Modeled Process

Code smells:

- G21: Understand the Algorithm

General idea: A modeler does not entirely understand the modeled process. As a result, the resulting model might be syntactically correct and its described process might even achieve the expected goal. However, the modeler is unable to later evolve the model and suggest optimizations to the modeled process. Sometimes such process models have multiple decision points and are overly complicated. On the other hand, sometimes the process description is too vague and high-level. The process description could also leave out vital sub-processes or require steps that are illogical.

Related model smells:

- G12 – if you do not understand the modeled process, then most certainly the result is an unclear model.

Scope: Model elements

Refactoring: Modelers should understand the subject that is depicted in their models. They should collaborate with clients if needed in order to improve their understanding.

Examples:

- An extended format use case, activity model, a system sequence model, or a state machine model that describes a process that is unclear to the modeler.
- A banking system where one can add money to an account but cannot withdraw it.
- A store checkout system that requires clients to each time register their personal data.
- A system that captures transactional data (for instance, orders) but does not offer to the management aggregate view over this data. Instead, the system presents to the management detailed information about each entity or does not present to them information at all.
- A sales system that does not allow its users to generate invoices.

G14: Dependencies in Models Are Not Visible Everywhere

Code smells:

- G22: Make Logical Dependencies Physical

General idea: A dependency between model elements is not explicitly presented in all the parts of the model where these elements must appear.

Related model smells:

- G15 – not making the dependencies visible everywhere is an example of imprecise model.
- G19 – make implicit dependencies a bit more explicit by using a logical order.

Scope: Model elements + Diagrams

Refactoring: Make the implicit dependency explicit. If model elements between which there is a dependency have both visual and textual representation that has to be in a model, then show the dependency in both these representations.

Examples:

- There is an include relationship between use cases that appears on a use case diagram but does not appear in the extended format textual descriptions of the use cases.

Already documented model smells:

Use case models (El-Attar and Miller, 2010)

- a8. A description of an actor that is not depicted in the UC diagram (rationale 1)

MATLAB/Simulink models (Gerlitz et al., 2015)

- Mismatch Between Visual and Data Flow
- Hidden Signal Flow

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- Hidden Concurrency

Model patterns for refactoring (Evitts, 2000):

- 4.5 Tombstone Packages
- 7.7 Product Traces Support Robustness

G15: Imprecise Model

Code smells:

- G26: Be Precise

General idea: A model element is ambiguous or imprecise, leading to the costly fixes at the later development stages. A necessary model element is missing from the model.

Related model smells:

- G12 – omission of details at the later stages of system development becomes harmful and signals imprecision in the model in question.

Scope: Model elements

Refactoring: Add precision to the specification of the model element. Add missing model elements. Explain things explicitly instead of expecting that all the model readers have exactly the same understanding as you assume. Search the help of client representatives to clarify the matters presented in the model.

Examples:

- Missing area of competence "Owner".
- A specification of subsystems presents areas of competence as functional subsystems. Actually each area of competence represents a role and each functional subsystem represents a main function. Each area of competence uses the services of one or more functional subsystems.
- A specification of subsystems presents all the entity types as separate registers. Actually each register corresponds to a main entity type and logically groups together data that corresponds to one or more entity types.
- Actor "Employee" (or synonym like "Worker") in a process model instead of more specific actors that correspond to the specific roles in the organization. Each role has specific tasks but the model overlooks this by showing essentially that each employee can fulfill any task. Actor "Employee" is justified in the models only if there is at least one process that is connected to all the workers, regardless of their role.
- An extended format use case description does not contain references to the database operations although at least some of these have also been defined.
- Actors "Employee" and "Manager" in a use case model without generalization relationship that shows that each manager is also an employee. Thus the manager is also connected to all the use cases of "Employee" in addition to the specific use cases to managers.
- Definition of an attribute that expected values are measurement results does not specify the unit of measurement.
- The specification of expected values of an attribute presents values that do not belong to the attribute type.
- A constraint to the attribute values is missing from a conceptual data model.
- A set of generalizations between domain classes or entity types that is not modeled as a generalization set and that lacks specification of constraints that apply to the set.

- A state transition in a state machine does not present neither triggering event nor resulting action.
- In a state machine model real final states are modeled as regular states and there is a single artificial final state that has incoming state transitions from all the regular states.
- Postconditions of a contract of a database operation do not show the creation and destruction of relationships between entities.

Already documented model smells:

UML models (Arendt and Taenzer, 2010)

- Attribute Name Overridden
- Concrete Superclass
- No Incoming

Use case models (El-Attar and Miller, 2010)

- a1. Accessing a Generalized Concrete
- a2. UCs Containing Common and Exceptional Functionality (rationale 1)
- a4. Functional Decomposition: Using the Extend Relationship (Generalization Instead of Extension)
- a6. Accessing an Extension UC (rationale 2,3)
- a7. Multiple Actors Associated With One UC (rationale 1)
- a8. A Description of an Actor That is Not Depicted in the UC Diagram (rationale 2,3)

Information models (McDonald and Matts, 2013)

- #1 – an item is on the output that is not in the domain model
- #2 – items on model, that are not on the output
- #3 – two bits of information in one place (1NF)
- #4 – no relationship
- #5 – one to one relationship. Are they the same thing?

MATLAB/Simulink models (Gerlitz et al., 2015)

- Pass-through Signal
- Cyclic Signal Path

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- Refused Bequest
- Poor Use of Abstraction

G16: Unauthorized Representations

Code smells:

- G32: Don't Be Arbitrary

General idea: A part of a model, some models from a set of models, or diagrammatic representations of the models do not follow the agreed upon structure. The structure determines what diagrams to make, what is the scope of each diagram, what textual information to present, and what is its structure. According to the broken windows theory (Wikipedia) it would attract additional changes that do not follow the structure, making the models gradually harder to comprehend.

Related model smells:

- S3 – it is similar in the sense that members of the team do not follow an agreement. In case of S3 the agreement is software industry-wide whereas in case of G16 the agreement is more localized i.e. within the team, within the project, or within the development organization.
- G8 – unauthorized representations might be a reason of inconsistencies of representation.

Scope: Model elements + Diagrams

Refactoring: Establish the structure and follow it. Be open to changing the structure if better ideas come up but decide this together with all the modelers who participate in the modeling effort as well as with the representatives of model consumers.

Examples:

- Models of different subsystems that have the same type (for instance, functional susbsystem or register) are presented in differently structured packages.
- Using different types of diagrams for describing from the same perspective different subsystems that have the same type.
- Some extended format use cases have structure elements that were not agreed before the start of the modeling project.
- In case of some functional subsystems textual descriptions of use cases are accompanied with a use case diagram and in case of some functional subsystems it is not the case.
- Conceptual data models of different registers provide differently structured attribute definitions.
- Invariants of a database operations (constraints to attribute values and relationships between entities) as a part of the contract of the operation instead of being a part of a conceptual data model.
- Domain class models of different functional subsystems provide differently structured domain class definitions.
- Different contracts of database operations have different structure.
- Different state machines provide differently structured information about the triggering events of state transitions.

G17: One Process with Multiple Tasks

Code smells:

- G15: Selector Arguments
- G23: Prefer Polymorphism to If/Else or Switch/Case
- G30: Functions Should Do One Thing

General idea: A process description bundles together multiple sub-processes and thus the described process has multiple tasks. During each execution of the process a particular sub-process has to be selected. The resulting model element is complicated, hard to comprehend, and violates separation of concerns design principle. In case of use case models different sub-processes have different preconditions. However, the coupling forces us to present the union of all their preconditions as the precondition of the use case. This in turn is incorrect because use case preconditions are the same for all of its scenarios.

Related model smells:

- G9 – one process with multiple tasks is an example of high coupling.
- G10 – a process that fulfills multiple tasks has no clear focus. It would be better to divide different tasks between different processes.
- G16 – careless refactoring may cause duplication.

Scope: Model elements

Refactoring: Replace the process with multiply processes that satisfy the separation of concerns principle. Similarly to the decomposition of tables or relational variables during the database normalization the decomposition should be nonloss (Date, 2006). It means that together all these resulting processes should describe the same behavior as the original process. Second condition is that each of the resulting processes should be needed to provide that guarantee i.e. decomposition does not result with the duplication among the resulting operations.

In case of a use case model replace a summary level use case with multiple user goal-level use cases. In case of an activity model or a system sequence model replace the model with a separate model for each sub-process. In case of a contract of database operation replace the operation with a separate operation for each different task. The resulting operations should not have selector parameters, which value is used to select the appropriate sub-process.

Examples:

- An activity model, a system sequence model, or the main scenario of an extended format use case contains decision points (if statements), which determine a subset of the process that has to be executed. For instance, use case “Manage products” has such subsets for registering, modifying, and deleting products.
- Contract of a database operation has a selector parameter that value determines which subset of the operation to execute. For instance, operation “Manage products” has tasks to register, modify, and delete product data.

Already documented model smells:

UML models (Arendt and Taenzer, 2010)

- Long Parameter List

Use case models (El-Attar and Miller, 2010)

- a2. UCs Containing Common and Exceptional Functionality (rationale 2)
- a4. Functional Decomposition: Using the Extend Relationship (Specific Extensions)
- a7. Multiple Actors Associated With One UC (rationale 3)

Process models (Weber et al., 2011)

- PMS2: Contrived Complexity
- PMS4: Large Process Models

MATLAB/Simulink models (Gerlitz et al., 2015)

- Subsystem with Multiple Functions
- Subsystem Interface Incongruence
- Independent Local Signal Paths

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- Scattered Parasitic Functionality

G18: Unnecessary Decomposition

Code smells:

- G18: Inappropriate Static

General idea: A concept or process has been decomposed into too small pieces that clutter diagrams and textual models. It make understanding the concept or process more difficult than it should be.

Related model smells:

- G6 – a detailed element that is referred by multiple other elements helps modelers to reduce duplication within the model and one should be careful not to reintroduce duplication while removing this smell.
- G17 – although having subfunction-level use cases narrows the tasks of each participating use case it makes it also more difficult to comprehend the model.

Scope: Model elements + Diagrams

Refactoring: It requires experience and careful examination of the context to decide as to whether an element is too detailed or not. If the too detailed element is invoked by some other element (in case of process models), then merge the invoked and invoking element. If the too detailed element is a part of inheritance/generalization hierarchy (for instance, entity subtypes in case of a conceptual data model; subclasses in case of a domain model; substates in case of a state machine model), then merge the more detailed element and its generalization. If the model does not contain an element that represent such generalization, then try to find it and add it to the model. For instance, subfunction-level use cases that describe distinct steps of a process should be merged together into one user goal-level use case. Another case is the speculative generality model smell (Arendt and Taenzer, 2010) in case of which there is an abstract class with only one concrete class that inherits from it. In this case the inheritance hierarchy represents unnecessary decomposition and the two classes should be merged into one concrete class.

Examples:

- Having a separate diagram for each model element.
- A functional subsystem that corresponds to a user goal-level use case but not to a family of related user goal-level use cases.
- A register that correspond to an entity type that is not a main entity type.
- Actors in a use case model that are too fine-grained. For instance, Evitts (2000) suggests modelers normally prefer actor ATM_Customer over actors Depositor, Withdrawing Customer, Balance Checker etc.
- A use case model contains a subfunction-level use case u' that is included by zero or one use case u. u' does not duplicate the process description of any other use case i.e. there is no need to include u' from other use cases. u' does not participate in any other (extend, generalization) relationship between use cases. This marks unnecessary decomposition and complication of the model.
    - User goal-level use case "Delete product" includes subfunction-level use case "Notify owner". There are no other use cases in the use case model that include "Notify owner". There are no other use cases that contain the same functionality as described by "Notify owner".

    - There are subfunction-level use cases “Select product”, “Determine quantity”, “Determine deliver address”, “Initiate payment”. These use cases describe the steps of the product ordering use case of an e-shop.
- A process model (scenarios of a use case, an activity model, a system sequence model) that does not represent an elementary business process but instead depicts a single step of it.
- A domain class or an entity type with too many subclasses and deep inheritance hierarchy.
- Having for each attribute of an entity type a set of database operations for creating, modifying, and deleting value of the attribute.
- A state machine with too detailed set of substates.

Already documented model smells:

UML models (Arendt and Taenzer, 2010)

- Extends
- Speculative Generality

Use case models (El-Attar and Miller, 2010)

- a3. Functional decomposition: Using the include relationship
- a5. Functional decomposition: Using pre and postconditions

Process models (Weber et al., 2011)

- PMS5: Lazy Process Models

MATLAB/Simulink models (Gerlitz et al., 2015)

- Superfluous Subsystem
- Superfluous Bus Signal
- Deeply Nested Subsystem
- Hierarchy

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- Functional Decomposition
- Poltergeist
- Speculative Generality
- Lazy Class
- Middle Man
- Scattered Parasitic Functionality

G19: Illogical Order

Code smells:

- G31: Hidden Temporal Couplings

General idea: The order of use cases on a diagram, use case textual descriptions in a textual specification, the order of activity diagrams or system state diagrams that depict the uses cases in a document, or the order of contracts of database operations in a textual specification does not match a logical order of their execution. A logical order is determined by the lifecycles of the entity types that data management these model elements describe.

The order of state machine diagrams in a document does not take into account dependencies between entities that lifecycles they present.

The order of descriptions of functional subsystems and registers does not take into account dependencies between entities that data they help us to manage.

Related model smells:

- G14 – by changing the placement of specifications (spatial property), we make implicit dependencies a little bit more explicit i.e. make it a bit more easy to comprehend these.
- S2 – long lines on a diagram mean that related elements are not clustered together and just like illogical order of elements it increases mental effort that is needed to comprehend the relationship.

Scope: Model elements + Diagrams

Refactoring: We should not present in the use case model and contracts of database operations the explicit order of executing use cases and operations, respectively. Nevertheless, such order exists and is implicitly determined by their pre- and postconditions (Gallen, 2012). If a use case or an operation achieves its postconditions, then it means that some necessary preconditions of other use cases or operations are fulfilled, respectively. The use cases and operations should be ordered accordingly. No use case should be presented before all the use cases that are needed to achieve its preconditions are presented. No operation should be presented before all the operations that are needed to achieve its preconditions are presented. In the textual documents the order comes top down. In diagrams the use cases should be presented in a clockwise order or in the top down order. State transitions of state machines of the main entity types have corresponding use cases and database operations (Eessaar, 2016). This information can be used to determine a logical order of use cases and operations in a presentation. There are multiple logical orders because different entities of the same type may have different lifecycles.

The order of system sequence diagrams and activity diagrams that are used to depict the scenarios of use cases should follow the order of use cases. If one does not create a use case model but implicitly finds user goal-level use cases, which correspond to elementary business processes, and presents the processes with the help of activity diagrams or system sequence diagrams, then the ordering algorithm of use cases should be used.

In case of state machines of entity types consider relationships between the entity types. If the existence of one or more entities with the type E is a precondition of entities with the type E' to come into existence, then place the state machine diagram of E before the state machine diagram of E'.

If the functional subsystems and registers are found based on the main business entity types (Eessaar, 2014), then the order of presentation of the subsystems should consider relationships between the entity types. If the existence of one or more entities with the type E is a precondition of entities with the type E' to come into existence, then place the description of the functional subsystem that corresponds to E before the description of the functional subsystem that corresponds to E'. If the existence of one or more entities with the type E is a precondition of entities with the type E' to come into existence, then place the description of the register that corresponds to E before the description of the register that corresponds to E'.

The same problem can occur while looking the models by using a CASE tool. Usually models, model elements, and diagrams are presented within a single hierarchy. If the CASE tool permits, then order the elements within the hierarchy based on the same principles.

Examples:

- Textual description, activity diagram, or system state diagram of use case "Delete product" is presented before the corresponding artifact of use case "Register product". In order to delete a product, one firstly has to register it.
- Operation "delete product" is presented before operation "register product". In order to delete a product, one firstly has to register it.
- State machine diagram of *Order* is presented before state machine diagram of *Product*. In order to make an order there have to be one or more products that are ready for ordering.

Already documented model smells:

MATLAB/Simulink models (Gerlitz et al., 2015)

- Non-optimal Port Order
- Non-optional Signal Grouping

G20: Insufficiently Described Behavior at Borderline Cases

Code smells:

- G3: Incorrect Behavior at the Boundaries
- G33: Encapsulate Boundary Conditions

General idea: A process model does not sufficiently take into account borderline cases that might derail the process. Systems that implement the modeled processes are thus unable to adequately react to these cases resulting with delays, stress, confusion, or loss of profit.

Related model smells:

- G13 – having incorrect behavior at borderline cases may be a result of not understanding the modeled process.
- G15 – insufficiently described behavior is an example of imprecise model.

Scope: Model elements

Refactoring: It is impossible to foresee all the exceptional cases and some of the cases have too low appearance probability. Try to establish borderline cases that are the most probable and take these into account in the process models.

Examples:

- Process models of an e-shop system do take into account that a supplier of products might become insolvent (less probable) but do not take into account that a package might get lost (more probable).

Already documented model smells:

Process models (Weber et al., 2011)

- PMS7: Frequently Occurring Instance Changes

G21: Process Does Not Match its Name

Code smells:

- G2: Obvious Behavior Is Unimplemented

General idea: A process model does not fulfill the expectations of the model consumers.

Related model smells:

- G13 – leaving out obvious behavior might be a result of not understanding the modeled process.
- G15 – having unexpected behavior is a form of imprecision.
- N1 – we assume behavior based on the name of the behavioral element and thus we can say that the name of the element does not describe sufficiently its nature.

Scope: Model elements + Diagrams

Refactoring: Based on the principle of least astonishment (Wikipedia), each process model should describe behavior that is expected by the model consumers. Otherwise model consumers cannot count on their intuition but instead always have to dig into the details of the model. The names of processes, their sub-steps, and diagrams that depict these are an important source of expectations. Thus, in case of each process model element or diagram use the name that is consistent with the behavior that is described under the name.

Examples:

- The name of a use case diagram name does not match the union of behavior described by the use cases that are presented on the diagram.
- The name of an activity diagram does not match its described behavior.
- The name of an activity does not match the behavior described by its activity nodes and edges.
- The name of a system sequence diagram does not match its described behavior.
- The name of a use case name does not match behavior described by its scenarios.
- The name of a database operation name does not match with the behavior that one would expect from the operation based on its postconditions.

G22: Multiple Languages in a Model

Code smells:

- G1: Multiple Languages in One Source File

General idea: A model contains at least one pair of claims about the real world where the claims are presented with the help of different natural languages.

A model that corresponds to a cell of the Zachman Framework (models one system aspect from one perspective) (Zachman, 2017) contains at least one pair of claims about the real world that have been presented with the help of different software languages. At the same time at least one of the languages is capable to present both these claims i.e. using multiple software languages is a matter of choice and not a matter of need.

In all these cases modelers and model users have to perform mental context switches, which distracts, wastes energy, and increases possibilities of mistakes.

Related model smells:

- G6 – with multiple languages it is easier to introduce duplication, meaning that the models contain the same claims presented in different languages.

Scope: Model elements + Diagrams

Refactoring: Use the smallest possible set of languages that allows you to represent a system in as expressively as you want. If possible, use one natural languages across all the models of the system. If it is not possible, then at least try to use one natural language across all the models that describe different aspects of the system from one perspective. For instance, it could be that models that describe the system from the business management perspective are in one natural language in order to be understandable to the widest range of future users of the system. They might want to give feedback and want to get comfortable with the system. On the other hand, the models from the engineer perspective are in English in order to make the system implementation understandable to a wide range of developers. On the other hand, due to the holistic nature of models it does not make sense to describe different aspects of the system from one perspective in different natural languages. For instance, application developers have to understand the database design and database designers the choices of application designers. Thus, process and data models should be in the same language.

Examples:

- Comments are in different natural languages.
- Names of different diagrams are in different natural languages.
- Names of different subsystems are in different natural languages.
- Some of the subsystems and their relationships are modeled by using UML packages and some are modeled by using ArchiMate enterprise modeling language.
- Textual descriptions of different use cases are in different natural languages.
- Names of different actions in activity models are in different natural languages.
- Some processes are modeled as UML activity models and some by using BPMN.
- Names of different entity types/domain classes are in different natural languages.

- Some entity relationship diagrams use UML class diagram notation and some Barker's notation.
- Different state machine models use different natural languages to describe triggering events of state transitions.
- Some state machines are represented as UML protocol state machines and some as Specification and Description Language state machines.
- Preconditions of a database operation are in one natural language and postconditions of the same operation are in another natural language.
- Different contracts of database operations are in different natural languages.
- Names of different system operations are in different natural languages.

G23: Not Using the Feedback of a CASE Tool

Code smells:

- G4: Overridden Safeties

General idea: A model is created by a CASE tool i.e. by using a tool that in addition to specifying a model allows its user to check the model and generate other artifacts from the model. Modelers have switched off displaying the warnings and notices of the system or just ignore these. Thus, they turn a blind eye towards potential problems, which early discovery makes it less costly to fix the problems.

Scope: Model elements

Refactoring: It is advantageous to fix problems in analysis models and not to propagate these into the models of the following phases. Switch on the functionality of displaying notices and warnings. If a CASE tool provides a possibility to verify models, then use it. Prefer CASE tools that are able to test models to the CASE tools that are unable to do that. Do not ignore notices and warnings of the CASE tool and try to fix problems as quickly as possible.

Examples:

- Any model that can be created by using a CASE tool.

Model patterns for refactoring (Evitts, 2000):

- 5.10 Let the Tools Do the Work

G24: Generating a New Artifact from a Model Takes a Lot of Time

Code smells:

- E1: Build Requires More Than One Step

General idea: A common functionality of CASE tools allows us to generate new artifacts (models for different perspectives, documentation, code, etc.) based on existing models. It means transforming a model to a new artifact. There are different problematic situations that can also occur simultaneously. The first is the worst. We assume that if models are in different projects, then these should be opened separately. Any CASE functionality that works with the models should be initiated separately in case of each project and can use only the models in the project as its input. Each project uses one or more physical files to store the model.

- A model from a cell of the Zachman Framework (i.e. a model that describes a system aspect from one perspective) (Zachman, 2017) is decomposed in a manner that its parts are in different CASE projects.
- Models that describe different system aspects from the same perspective are in different CASE projects.
- Models that describe the system from different perspectives are in different CASE projects.

To complicate things further, different models or parts of the same model could be created by using different CASE tools. All this makes transformations more time consuming and error prone. Sometimes one has to open and close multiple projects and give the same command multiple times to initiate the generation of pieces of a new artifact. The resulting pieces have to be merged manually. If generating a new artifact requires input from multiple models that are in different projects, then one firstly has to find a way to merge the input models to one project. In the end it reduces the motivation of modelers to use CASE tools in the future projects.

Related model smells:

- G6 – having multiple projects that contain models of the same system can easily end with redundancy across projects.
- G22 – the use of different CASE tools might be a sign that different software languages are used to describe the system and perhaps the number of languages could be reduced.
- G23 – having models distributed between different projects hampers the ability of the CASE tool to give a quality feedback.

Scope: Model elements + Diagrams

Refactoring: If you use a CASE tool for modeling, then all the models that describe different aspects of the same system from the same perspective and that can be created by using the CASE tool, should be in the same project. In some CASE tools it is possible to embed textual documents to projects making it possible to keep both visual and textual models in the same place. In order to reduce the number of projects it is alright to also have models for different perspectives together in the same project as well. The project should be under version control. The work of Elaasar and Labiche (2012) shows that the interchange of UML models between different CASE tools by using XMI format may result with losing information from models. The reasons are errors and ambiguities in the XMI standard, support of different versions of

the XMI standard by different tools, and lack of testing between tools. Thus, a better strategy is to start with one project and not to rely on later refactoring by merging the projects.

Examples:

- Entity relationship models of different registers are in different modeling projects.
- Use case models of different functional subsystems are in different modeling projects.
- Use case diagrams and textual descriptions of use cases are in different modeling projects.
- State machine models and use cases that correspond to the state transitions of these are in different modeling projects.
- Use case models and system sequence models that describe their scenarios are in different modeling projects.

Already documented model smells:

Process models (Weber et al., 2011)

- PMS8: Frequently Occurring Variant Changes

Model patterns for refactoring (Evitts, 2000):

- 5.10 Let the Tools Do the Work

G25: Ill-considered Types of Attributes

Code smells:

- J3: Constants versus Enums

General idea: The selection of the type of an attribute of an entity type in a conceptual data model or a domain class in a domain class model does not take into account possible operations with the values of these attributes or the extreme values of the attribute. In these models the types are not platform-specific and are only approximations. CASE tools generally offer a small set of predefined types for using in this situation. Nevertheless, modelers should try to be as precise as they can.

Related model smells:

- G15 – poorly selected attribute types in a conceptual data model or domain class model is an example of imprecise model.

Scope: Model elements

Refactoring: For each type there is a set of operators for performing operations with the values that belong to the type. For each attribute use a type that the best match the profile of expected operations with the values of the attribute.

Examples:

- A national identification number attribute has integer type although the users would not perform arithmetical operations with the identification numbers but instead extract encoded information by finding a substring of the number.
- The identifier of clients does not consider a possibility that the number of clients may exceed 32767 (the biggest interger that belongs to the SMALLINT type).

Already documented model smells:

UML models (Arendt and Taenzer, 2010)

- Primitive Obsession

Object-oriented models (Misbhauddin and Alshayeb, 2015)

- Primitive Obsession

G26: Parameters with Boolean Type

Code smells:

- F3: Flag Arguments

General idea: A Boolean parameter of a database operation may indicate that the operation has multiple tasks and the value of the parameter determines the task that the operation is expected to achieve during a particular execution. On the other hand, searching problems based on this smell may give false positive results because operations may have legitimate Boolean parameters, which values are recorded in a database.

Related model smells:

- G17 – Boolean parameters may indicate that a process or operation has multiple tasks.

Scope: Model elements

Refactoring: Decompose the operation to have a separate operation for each task. Decomposition mean that the original is replaced with multiple operations, which are more focused. The resulting operations do not need a Boolean parameter, which value would guide their internal execution logic.

Examples:

- An operation has two tasks – to register products in a database and to delete a product from the database. Which task to fulfill depends on the value of the Boolean parameter *is_insert*. If the value of the parameter is TRUE, then the operation has to fulfill the insert task, otherwise the delete task.

G27: Negative Conditionals

Code smells:

- G29: Avoid Negative Conditionals

General idea: A guard condition in an activity model, state machine model, or system sequence model is double negative. A constraint definition in a conceptual data model or domain class model is a double negative. A pre- or postcondition in a use case model or a contract of database operation is a double negative.

Related model smells:

- G15 – With such conditionals it is easier to make a mistake in the formulation and the result is an imprecise model.

Scope: Model elements

Refactoring: Reformulate conditional (Boolean expression) so that instead of double negative it uses a positive.

Examples:

- A guard condition that it must not be the case that a product does not belong to any category. A more understandable guard condition is that a product must belong to at least one category.
- A constraint that it must not be the case that e-mail address does not contain exactly one @ sign. A more understandable constraint is that e-mail address must contain exactly one @ sign.
- A precondition of a use case that it not must be the case that user is not logged in.
- A postcondition of a database operation that it must not be the case that the relationship between two entities has not been registered.

Smells About the Style of Models

S1: Too Big Diagrams

Code smells:

- F1: Too Many Arguments
- G8: Too Much Information

General idea: A diagram represents too much information, making it difficult to comprehend and distinguish relevant from irrelevant.

Related model smells:

- G10 – if diagram has too many elements, then it means that it does not have a clear focus.
- G18 – having unnecessary decomposition may cause too big diagrams.

Scope: Diagrams

Refactoring: Make each diagram reasonably small and focused. As a rule of thumb follow the 7+/-2 rule by placing approximately so many elements to each diagram. Find out as to whether users of the model will print the diagrams out and what paper size they will typically use. Make sure that each resulting diagram is readable if it is printed out to a typically sized paper (for instance, A4). Create multiple smaller diagrams with an overlap instead of a single big diagram i.e. decompose a big diagram.

Create diagrams based on the framework of subsystems (Eessaar, 2014). Each subsystem type determines one or more model types that corresponding models can be used to describe subsystems with such type. For instance, from the conceptual (owner) viewpoint registers can be described by using a conceptual data model, state machines of main entity types, and contracts of database operations. Each such model type corresponds to one or more aspects of the Zachman Framework (Zachman, 2017) and to exactly one viewpoint. For instance, conceptual data model describes data aspect of the system from the conceptual viewpoint. It describes the data over all the registers. Its entity-relationship diagrams help us to produce register-centric views to the data. In case of each subsystem create one or more diagrams for each relevant model type. A model element could be presented in multiple diagrams but its details should be visible only in one diagram. For instance, instead of creating an entity relationship diagram over all the registers or some arbitrary subsets of registers, create a set of diagrams for each register. For each entity type there should be exactly one diagram that presents it details (attributes). Each modeled entity type describes data that belongs to a register. Therefore, this diagram must be one of the diagrams of the register. In order to show relationships the entity type may appear on other diagrams as well (of the same register or different registers) but its details (attributes) must be hidden.

For instance, in an e-shop system there is the *register of products*. Its description contains multiple entity-relationship diagrams. One of the diagrams displays the details of the entity type *Product*. Other entity-relationship diagrams of the register as well as entity-relationship

diagrams of some other registers (for instance, *register of orders* and *register of reviews*) display a scaled-down description of the entity type *Product*. If one wants to see the details of *Product*, then there is exactly one diagram for that. If, for instance, the details of *Product* have to be changed, then only one diagram changes and has to be replaced in a documentation.

Examples:

- A use case diagram that presents use cases of multiple functional subsystems.
- A domain class model that presents the details of domain classes of multiple functional subsystems.
- An activity diagram that presents detailed information about all the processes of a functional subsystem.
- An entity relationship diagram that presents the details of multiple registers.
- A system sequence diagram that presents detailed information about the processes across multiple functional subsystem.

Already documented model smells:

Process models (Weber et al., 2011)

- PMS4: Large Process Models

MATLAB/Simulink models (Gerlitz et al., 2015)

- Long Port List

Model patterns for refactoring (Evitts, 2000):

- 4.2 Providing Focus
- 4.3 Explicit Elision
- 4.11 Seven Plus or Minus Two

S2: Long Lines on a Diagram

Code smells:

- G10: Vertical Separation

General idea: A diagram presents two or more model elements that are connected through a relationship. The element are placed relatively far from each other, meaning that the line that represents the relationship has to be quite long. On its path it may cross other visual elements on the diagram, including other lines. Therefore, understanding as to what elements are connected with each other on the diagram requires extra mental effort.

Related model smells:

- G19 – illogical order of elements means that related elements are not clustered together and just like long lines it increases mental effort that is needed to comprehend the relationship.
- S1 – long lines are often present on too big diagrams where one has to connect elements that have to be placed too far from each other.
- S3 – avoiding long lines is an example of a modeling style convention.

Scope: Diagrams

Refactoring: On each diagram related elements should be placed close to each other in order to avoid long lines. One should avoid crossing lines and if it is not possible, then make these jump one another (Ambler, 2014a).

Examples:

- A long dependency line between two packages that shows that one subsystem depends on another.
- A long control flow line between two actions on an activity diagram.
- A long relationship line between a use case and an actor on a use case diagram.
- A long generalization line between two domain classes on a domain class diagram.
- A long association line between two entity types on an entity relationship diagram.
- A long state transition line between two states on a state machine diagram.
- A long line that represents invocation of a system operation on a system sequence diagram.

Model patterns for refactoring (Evitts, 2000):

- 4.4 Tree Routing

S3: Not Following the Popular Modeling Style Conventions

Code smells:

- G24: Follow Standard Conventions

General idea: A model does not follow modeling style conventions that are commonly used in the industry. These conventions elaborate the appearance and placement of model elements on diagrams. Such conventions do not require extra documentation that accompanies the models because the models themselves are the best example of their usefulness and application. However, modelers should agree between each other that they will use such conventions.

Related model smells:

- G8 – not following widely agreed conventions may increase the possibility of inconsistencies, especially if the modeling process is a teamwork.
- G16 – it is similar in the sense that members of the team do not follow an agreement. In case of S3 the agreement is software industry-wide whereas in case of G16 the agreement is more localized i.e. within the team, within the project, or within the development organization.
- N3 – names of model elements are a matter of style as well.

Scope: Diagrams

Refactoring: Team members should agree what convention to follow. Change the style of models and do it consistently in all parts of the modeling project. Do the refactoring horizontally, i.e. if you change the style of a particular type of artifact (for instance, entity relationship diagram), then change its style consistently in the entire modeling project. Good starting points for studying conventions are (Evitts, 2000) and (Ambler, 2014b).

Examples:

- Comment text on a note is not left-justified.
- Comment text in a textual model is not between /**/.
- Packages that reference different types of subsystems are not vertically layered.
- The start point is not in the top-left corner of an activity diagram.
- The extending use case is not below its parent use case on a use case diagram.
- A subclass is not below its superclass in case of a domain class diagram or an entity-relationship diagram.
- The end state is not in the bottom-right corner of a state machine diagram.
- Human and organization actors are not on the left-most side of a system sequence diagram.

Model patterns for refactoring (Evitts, 2000):

- 4.6 Inheritance Goes Up
- 4.10 Text Workarounds
- 5.1 Standard Diagrams
- 8.3 Icons Clarify Components
- 8.4 Pictures Depict Nodes

S4: Structure by Naming

Code smells:

- G27: Structure over Convention

General idea: Structuring of a model is done with the help of names of the elements instead of using specific features, like packages that the modeling language provides for the structuring purpose. Model elements that are used to describe the same part of the system, from the same aspect, and for the same perspective are grouped together with the help of similar names that follows a certain pattern. On the other hand, longer names take more effort to read and understand. Too long names might be unsuitable for using in the source code. The modeler as well as model reader have to be aware and follow naming patterns.

Related model smells:

- G8 – if the name does not match intention any more, then the result is inconsistency. For instance, entity type *Person_Country*, is moved from the *Register of persons* to the *Register of classifiers*. According to a naming pattern its new name should be *Classifier_Country* but the element has not been renamed.

Scope: Model elements + Diagrams

Refactoring: Use multi-level hierarchy of packages or other similar features that are provided by a modeling language in order to establish and communicate the structure of the model. Do not repeat in the names of model elements and diagrams the names of the packages that contain these.

Examples:

- The entire system description from the owner perspective is placed into one package that has no sub-packages. Diagram names are used for grouping together related diagrams that represent the system from one aspect. For instance, in case of entity relationship diagrams the names of the diagrams refer to the model type (ERD) and register (data centric subsystem).
    - ERD_Register_of_products
    - ERD_Register_of_orders
    - ERD_Register_of_clients
    - …

  The names of other model elements also refer to the parts of the system that they help to describe. For instance, the names of entity types begin with the name of a register that hosts their corresponding data.
    - Classifier_Country
    - Classifier_Product_state_type
    - …

  Instead of relying on naming it is better to create a multi-level package structure. For instance, at the highest level there is a package *Analysis*. It contains one package for each subsystem type (Eessaar, 2014). One of the packages is *Registers*. It may have stereotype <<layer>> to depict that its content describes one layer of the business architecture. The package contains one sub-package for each register of the system.

The name of each such package has the structure *Register_of_X* (where X is the name of a main entity type) and each such package has stereotype <<subsystem>>. Each such package contains one or more entity-relationship diagrams. The name of the diagrams should not refer to the name of the register but should describe the exact focus of the diagram. For instance, in case of *Register of products*:

- ERD_Products (main data)
- ERD_Products (product categories)
- ERD_Products (product properties)
- …

Each UML class that is used to depict an entity type is placed to the package that corresponds to the register that hosts its corresponding data. The names of entity types should not contain the name of the register that contains them. For instance, better names of entity types that belong to the *Register of classifiers* would be:

- Country
- Product_state_type
- …

Smells about the Naming of Models and Their Elements

N1: The Name Does Not Express Its Nature

Code smells:

- G16: Obscured Intent
- G20: Function Names Should Say What They Do
- N1: Choose Descriptive Names
- N4: Unambiguous Names
- N7: Names Should Describe Side-Effects

General idea: The name of a model element or a diagram is too short, vague or general. It may also be too specific. It does not express well enough the intent of a modeler who created it. One has to check supporting textual descriptions in order to understand the meaning of the poorly named element and the context where the element is situated.

Related model smells:

- G8 – some vague names may appear due to the inconsistencies of using names (see the model smell "Inverted Operation Name" of Arendt and Taentzer (2010)).

Scope: Model elements + Diagrams

Refactoring: Choose a better name. Fowler (2009) acknowledges that naming things is one of the hardest problems in computer science. Do the refactoring horizontally, i.e. if you change the naming scheme of a particular type of model element (for instance, use case), then change the names consistently in the entire modeling project.

Examples:

- It could be that the element or diagram has no name (Arendt and Taentzer, 2010). Arendt and Taenzer (2010) presents smells about unnamed class model elements (package, class, interface, data type, attribute, operation, or parameter), use case model elements (use case or actor), and state machine elements (state).
- Acronym PM as the name of a functional subsystem instead of "Product management".
- "Food management functional subsystem" and "Register of foods" in the business architecture of a shop. In reality, the shop sells different kinds of products, not only food products. Thus, it is more appropriate to use the name "Product management functional subsystem" and "Register of products".
- An activity diagram or a system sequence diagram with the name "Create" instead of "Create a product".
- A use case or a database operation with the name "Delete" instead of "Delete an order".
- A use case name does not refer to an activity. For instance, "Product detailed report" instead of "Look product detailed report".
- Attribute "time" of entity type or domain class "Consultation" instead of "begin_time".

- Entity relationship diagrams of *Register of classifiers*, with the names “Classifiers 1”, “Classifiers 2”, and “Classifiers 3” instead of names like “State classifiers”, “ISO classifiers”, and “Product classifiers”.
- End state “Finished” in a state machine model of *Order*. On the other hand finishing means in this case archiving and a better name would be “Archived”.

Already documented model smells:

UML models (Arendt and Taenzer, 2010)

- 2.1.4 Inverted Operation Name
- 2.1.7 Multiple Definitions of Classes with Equal Names
- 2.1.11 Unnamed Element
- 2.2.2 Unnamed Use Case Element
- 2.3.2 Unnamed State

Process models (Weber et al., 2011)

- PMS1: Non-intention Revealing Naming of Activity/Process Model

MATLAB/Simulink models (Gerlitz et al., 2015)

- Vague Name
- Unnamed Signal Entering Bus

N2: Names at the Unsuitable Level of Abstraction

Code smells:

- N2: Choose Names at the Appropriate Level of Abstraction

General idea: A name of an element of a model m or the name of a diagram that represents it refers to a perspective that is in the Zachman Framework (Zachman, 2017) below the perspective from where m looks to the system.

Related model smells:

- G7 – the name points to the fact that the element is at the wrong level of abstraction and thus perhaps in a wrong model.

Scope: Model elements + Diagrams

Refactoring: Change the names so that these are suitable for the level of abstraction in the model. Do the refactoring horizontally, i.e. if you change the naming scheme of a particular type of model element (for instance, use case), then change the names consistently in the entire modeling project.

Examples:

- The name "Odoo Sales" as the name of a functional subsystem. The name refers to a particular product. Deciding how to implement a particular subsystem (for instance, by adapting some existing software package) is a design decision.
- The name “DBMS Scheduler” of an actor in an analysis use case model or an activity model. The name could also appear in the state machine diagram as a part of the explanation of the triggering event of a state transition. The goal is to show that a system task should be executed regularly. The name of the actor refers to a design decision. Instead, we should use actor “Time” as the primary actor (Ambler, 2014b) or as the secondary actor (Crain, 2002).
- The name “Database tables” of an entity relationship diagram that is a part of analysis level conceptual data model.
- The name “Identify user with Smart-ID” of an analysis use case, activity diagram, or a system sequence diagram. Smart-ID (Buldas et al., 2017) is a technical mean for user identification and selecting suitable technical means is a design task. Analysis should establish that there is a need for identification in the first place.
- The name “Register Product in a table” of an analysis database operation or a system operation in a system sequence model. Tables are the main building blocks of SQL databases. However, there are a lot of different data models (network, hierarchical, object-oriented, XML, JSON document-based, property graph, key-value pairs, etc) based on that to build up the database. Selecting the data model or a database management system is the task of logical and physical design, respectively.

N3: Not Following the Popular Naming Conventions

Code smells:

- N3: Use Standard Nomenclature Where Possible

General idea: A model element does not follow naming conventions that are commonly used in the industry. Such conventions do not require extra documentation that accompanies the models because the models themselves are the best example of their usefulness and application.

Related model smells:

- G8 – not following widely agreed conventions may increase the possibility of inconsistencies, especially if the modeling process is a teamwork.
- S3 – names of model elements are a matter of style as well.

Scope: Model elements + Diagrams

Refactoring: Team members should agree what convention to follow. Change the naming of model elements and do it consistently in all parts of the modeling project. Do the refactoring horizontally, i.e. if you change the naming scheme of a particular type of model element (for instance, use case), then change the names consistently in the entire modeling project.

Examples:

- The name of a use case does not begin with a strong verb. (Ambler, 2014b).
- The name of an entity type is not in the singular form.
- The name of a state transition in a state machine model is not in the past tense. (Ambler, 2014b).
- The name of a database operation does not begin with a strong verb.

N4: Encoded Names

Code smells:

- N6: Avoid Encodings

General idea: The name of the model element contains encoded information about its type, scope, or intention. In some cases it is good for the readability but if taken too far makes the names long and cryptic.

Related model smells:

- S4 – a purpose of using encodings in the names might be to improve the internal structuring of models.

Scope: Model elements + Diagrams

Refactoring: Remove encodings from the names and do it consistently in all parts of the modeling project. Do the refactoring horizontally, i.e. if you change the naming scheme of a particular type of model element (for instance, entity type), then change the names consistently in the entire modeling project. In order to better express intention in UML, use modeling profiles that allow modelers to attach stereotypes to existing model elements in order to change their meaning and appearance. For instance, one may use the stereotype <<subsystem>> for packages that depict subsystems and <<entity>> for classes that depict entity types. There is at least one case when some kind of encoding is useful. In case of parameters of database operations consider the prefix "p_" or "_" at the beginning of the name. The reason is that the parameter names are derived from the names of attributes. Without this kind of distinction it is difficult to distinguish references to attributes and parameters in the pre- and post-conditions of database operations.

Examples:

- The name of an administrative functional subsystem starts with the prefix "adm_".
- The name of a use case that represents a cross-cutting concern starts with the prefix "cc_u_".
- The name of an activity diagram about a use case starts with the prefix "act_u_".
- The name of an attribute with a string type starts with the prefix "str" in a domain class model or in a conceptual data model.
- The name of the state machine diagram of an entity type starts with the prefix "sm_et_".
- The name of a database operation that modifies data in a database starts with the prefix "op_m_".
- The name of a system sequence diagram about a use case starts with the prefix "seq_u_".

N5: Diagrams Have Too Short Names

Code smells:

- N5: Use Long Names for Long Scopes

General idea: A container (for instance, a package) contains multiple diagrams at the same level i.e. not in different sub-containers. One or more of these diagrams has a short name that does not give sufficient idea about what is expressed on the diagram. To find a diagram from this set one cannot rely on the diagram names and the names of their context i.e. containers. Instead one has to open and look the diagrams one by one. It makes it more difficult and time consuming to find a relevant diagram. A sign of such problem is that diagram names differ only by the attached number.

Related model smells:

- N1 – the undescriptive and short name of a diagram is an example of a name that does not sufficiently express the nature of the named element.

Scope: Diagrams

Refactoring: Give to the diagrams long and descriptive names. The bigger part of the system the diagram expresses the longer name it may need.

Examples:

- Diagrams ERD1, ERD2, etc. in a package that contains a conceptual data model or a part of the model that describes a register.

Smells about Testing Models and Testing Modeled Systems Based on Models

T1: Model-related Testing Takes a Lot of Time

Code smells:

- E2: Tests Require More Than One Step
- T9: Tests Should Be Fast

General idea: It takes a lot of time to verify a model. It takes a lot of time to test a system based on its models. The result of the former may be a reason of the later i.e. model presentation is suboptimal and the models are characterized by one or more model smells. Bad presentation distracts testers, may hide problems, and makes the process less effective. It may also hamper automated testing if, for instance, the model is physically stored across multiple files that are created by different CASE tools. Therefore, testing is not done as often and as thoroughly as it should be or is not done at all. Thus, serious problems in the modeled system are discovered too late that makes them very costly or even impossible to fix.

Related model smells:

- *Model verification*:
  - G23 – switching off or ignoring notices and warnings of the CASE tool means that the human user who perform verification will not get some vital information.
  - G24 – having a model distributed across different CASE projects complicates the verification.
  - All the model smells that belong to the classes G, S, and N. Modelers should discover the smells and fix these as the result of model verification. It paths the way towards more successful model-based system testing.

Scope: Model elements + Diagrams

Refactoring: A prerequisite of frequent model-based system testing is frequent model verification. Automate the model verification and model-based system verification as much as possible. For the sake of increasing automation try to use a CASE tool that has a built-in extension mechanism (for instance, in the form of add-ins, templates, or model based queries) or are created by using a tool (metamodeling system) that allows developers to extend the functionality.

Model-based system validation means reviews, walkthroughs, and inspections both within the development team and together with the representatives of the client and future users of the system. In addition, due to technical restrictions, at least some model-based system verification has to be done manually as well. Therefore, the presentation of models should be as good as possible in order to facilitate the process. Participants should be able to concentrate to the content rather than to the presentation. Therefore, take steps to speed up model-based testing by ensuring that model presentation is as good as possible. A part of it would be model

verification by checking models against model smells and refactoring the models if needed. In order to speed up model verification take steps that are suggested as the fix to the smell G24.

Examples: See related model smells.

Model patterns for refactoring (Evitts, 2000):

- 5.10 Let the Tools Do the Work

T2: Incomplete Model-related Testing

Code smells:

- T1: Insufficient Tests

General idea: There is a model or a part of a model that has not been verified. It may be one of the reasons why there is an aspect of the modeled system or a part of the system that is not tested based on models. Testing does not take place during the entire software project.

Related model smells:

- T1 – a reason could be that testing is too time consuming.

Scope: Model elements + Diagrams

Refactoring: Model-related testing should happen continuously throughout the project as long as there are changes in the models. Model verification should ideally cover all the models that are created during the project. Model-based tests of the system should cover at least the most important parts of the system. Model-based validation requires active client participation in the project. A precondition of effective model-based testing is continuous model verification in order to ensure that the models are free of modeling smells and thus can be used as an effective input to the testing process.

Examples:

- Models are not verified because their physical separation to files and physical distribution of the files to different geographical locations makes it too inconvenient and time consuming.
- Use cases are not (continuously) validated against the (changing) user requirements. As a result it turns out that the use case model is out of date and the functionality that has been implemented based on this does not satisfy the current needs of system clients and users.
- A conceptual data model of the system is divided between different CASE projects that have been created by using different CASE tools. Different entity relationship diagrams use different modeling notation due to the fact that the used CASE tools support disjoint sets of notations. Some parts of the model are in textual documents. The documents mix different natural languages. The structuring of the model is poor. For instance, classes that depict entity types from different registers as well as use cases that describe different functional subsystems are together in a single package, meaning that it is difficult to comprehend these. As a result the conceptual data model is not tested at all. Validation of the system happens only at the end of the project and it turns out that the database is unable to record important data. Verification of the database structure happens also at the end of the project and it turns out that the database features multiple SQL database design antipatterns (Karwin, 2010) that are costly to fix.

Model patterns for refactoring (Evitts, 2000):

- 7.9 Tests Need Models

T3: No Tracking of the Extent of Model-related Testing

Code smells:

- T2: Use a Coverage Tool!

General idea: Each model corresponds to a cell of the Zachman Framework (describes an aspect of the system from one perspective) (Zachman, 2017). A model (as a logical entity) maybe physically distributed across multiple CASE projects. For instance, different CASE projects contain descriptions of different parts of the system. Each such project hosts one or more models that correspond to one or more cells of the Zachman Framework. Some models or their parts may be duplicated across multiple projects. Information about what, when, how, by whom, and why has been tested in relation to models is not recorded. A result is that some models (and thus corresponding system aspects, perspectives, and parts) are subject for too frequent testing whereas some models are used infrequently or are not used at all.

Related model smells:

- T2 – a result is incomplete testing.

Scope: Model elements + Diagrams

Refactoring: For each model-related testing episode record information:

- Why the testing was conducted? (model verification, model-based verification, or model-based validation)
- What model and project were used? What diagrams were used to get the understanding of the modeled system?
- When the testing happened?
- Who participated in the testing?
- How the testing was conducted? (automation, review, inspection etc.)

If possible try to visualize the results and compare it against the list of all the created models and projects to make it easy to notice gaps in the test coverage.

Examples: It could happen with any model.

Model patterns for refactoring (Evitts, 2000):

- 7.9 Tests Need Models

T4: Ignoring Trivial Model-related Tests

Code smells:

- T3: Don't Skip Trivial Tests

General idea: Some checks are discarded as trivial during the model-related testing.

Related model smells:

- T2 – a result is incomplete testing.

Scope: Model elements + Diagrams

Refactoring: It is very difficult to draw a line what kind of model-related tests are trivial and what kind are not. If it does not take much time to check something, then check it because you never know whether a discovered problem, which might be small or at the first glance insignificant, might be a symptom of a bigger problem. A general strategy is to automate as many trivial tasks as possible.

Examples: It could happen with any model. For instance, checking the naming style of entity types might look trivial to somebody. However, mixing plural and singular forms of names of entity types may lead to the incorrect selection of multiplicities for relationships and this will lead to a database structure that does not allow us to correctly capture facts about the real world.

Model patterns for refactoring (Evitts, 2000):

- 7.9 Tests Need Models

T5: Ignoring System Parts in Model-based Testing

Code smells:

- T4: An Ignored Test Is a Question about an Ambiguity

General idea: Some parts of the system are ignored during the model-based system testing because requirements about these are unclear, underspecified, or ambiguous.

Related model smells:

- T2 – a result is incomplete testing.

Scope: Model elements + Diagrams

Refactoring: Try to clarify requirements right away, do not postpone it. “Errors made in software requirements analyze are increasing costs by the multiplying factor 3 in each phase. This means that the effort needed to correct them in the design phase is 3 times, in the implementation phase 9 times and in system tests 27 times more expensive than if they would be corrected at the error source; that means in the software requirements analyze.” (Jaakkola et al., 2016). Modify models according to the clarified requirements. Verify the modified models to make sure that the presentation has a good quality. Test the system based on the modified models.

Examples: It could happen with any model.

Model patterns for refactoring (Evitts, 2000):

- 7.9 Tests Need Models

T6: Ignoring Borderline Cases in Model-based Testing of Processes

Code smells:

- T5: Test Boundary Conditions

General idea: The alternative scenarios of processes are ignored while testing the system based on models. The lack of the alternative scenarios is overlooked i.e. assumption is made that everything will happen according to the plan and nothing can go wrong. It is easier to understand and model the main scenario than borderline cases. However, it is easier to make mistakes in case of the latter because they are less understood and perhaps no one has thought about these before.

Related model smells:

- G13 – if testers do not understand the modeled process, then it is easy to overlook borderline cases.
- G20 – insufficiently described behavior at borderline cases is not discovered.
- T2 – a result is incomplete testing.

Scope: Model elements + Diagrams

Refactoring: Do not ignore borderline cases while testing the system based on models. In case of validations work with client representatives who are familiar with the borderline cases.

Examples: It could happen with any model that can be used to depict processes (textual descriptions of use cases, activity models, state machine models, system sequence models).

Model patterns for refactoring (Evitts, 2000):

- 7.9 Tests Need Models

T7: Ignoring the Surrounding Areas of Found Mistakes in Model-related Testing

Code smells:

- T6: Exhaustively Test Near Bugs

General idea: Problems and mistakes tend to concentrate but this observation is ignored in the testing process.

Related model smells:

- T3 – if the testing process is not tracked and the results are not recorded, then it is easy to miss the need to recheck the surrounding areas. In this case it depends on memory and communication of individual testers and is random in nature.

Scope: Model elements + Diagrams

Refactoring: If you identify a problem (for instance, an instance of a model smell) in a model during model verification, then check the entire model (that is possibly divided between different projects and geographical locations) about the possible occurrences of the same problem. If modeling is a team effort and model verification shows that models of a certain individual or team have above average number of mistakes, then pay more attention to their models in general. Perhaps their working habits and attitude towards the quality manifest itself in other models as well.

If model-based testing of a system identifies a problem (related to a system aspect) in a system part, then look more closely the part of the system from all the aspects as well as the same aspect in different parts of the system.

Examples: It could happen with any model.

- Naming problems of entity types in case of a register may also appear in other registers.
- Having too big diagrams or diagrams with too long lines in one place may indicate that the modeler generally ignores the drawbacks of such representation and the problems repeat in models that correspond to the different aspects or perspectives of the Zachman Framework (Zachman, 2017).

Model patterns for refactoring (Evitts, 2000):

- 7.9 Tests Need Models

T8: Ignoring Patterns of Mistakes in Model-related Testing

Code smells:

- T7: Patterns of Failure Are Revealing

General idea: Mistakes are treated as random entities rather than symptoms of a deeper problem that is, for instance, caused by misunderstanding of a certain model type, using an unsuitable tool or language for modeling, or having a wrong mindset towards quality.

Related model smells:

- T3 – if the testing process is not tracked and the results are not recorded, then detecting patterns cannot be a systematic undertaking. In this case it depends on memory and communication of individual testers and is random in nature.
- T7 – ignoring the surrounding areas shows that the testers think that it was a random mistake rather than a manifestation of a pattern.

Scope: Model elements + Diagrams

Refactoring: Exercise yourself towards seeing patterns and metaphors that characterize these by not just looking the modeled systems but also everyday events, places, and artefacts as well as theories from different domains. Another useful effect of this is that one is better able to use patterns, models, smells that have been defined for one domain in another domain as well. This particular set of smells, which has been defined based on code smells, is an example of that kind of exercise.

Try to find a representation (visualization) of mistakes that supports the discovery of patterns. A prerequisite is that the mistakes are somehow recorded in the system. Because manual recording (as well as manual testing) is time consuming and error prone it would be very useful to automate the testing process and a part of it would be automating the recording of mistakes.

Examples: It could happen with any model. For instance, a modeler has misunderstood an important requirement. Therefore there are mistakes in multiple models. If one discovers a mistake and fixes it but does not look other models in terms of the mistake, then the resulting set of models still describes the system incorrectly.

Model patterns for refactoring (Evitts, 2000):

- 7.9 Tests Need Models

T9: Ignoring Patterns of Testing in Model-related Testing

Code smells:

- T8: Test Coverage Patterns Can Be Revealing

General idea: Ideally, testing should cover all the system development artifacts and be a part of every system development phase. In reality it is often not the case and testing happens only as an afterthought, covers only some parts of the system, and happens only towards the end of the development project. During the retrospective meetings of the development projects, the testing process is not treated as something that can give insights about the system or attitudes towards the system and its development.

Related model smells:

- T3 – if the testing process is not tracked and the results are not recorded, then detecting patterns cannot be a systematic undertaking. In this case it depends on memory and communication of individual testers and is random in nature.

Scope: Model elements + Diagrams

Refactoring: Capture testing process and study it to find useful clues about the following.

- What system aspects, system perspectives, and system parts were not tested?
- For what purposes the testing was not used?
- Who avoided the testing tasks?
- When the testing did not happen?

For instance, looking what parts of the system have not been frequently tested based on models (or not tested at all) gives us clues about what system parts are considered to be less important in the modeled system.

Examples: It could happen with any model.

Model patterns for refactoring (Evitts, 2000):

- 7.9 Tests Need Models

References

- Ambler, S., 2014a. General Diagramming Guidelines. Agile Modeling [WWW] http://agilemodeling.com/style/general.htm (22.10.2017)
- Ambler, S., 2014b. Modeling Style Guidelines. Agile Modeling [WWW] http://agilemodeling.com/style/ (22.10.2017)
- Arendt, T., Taentzer, G., 2010. UML Model Smells and Model Refactorings in Early Software Development Phases. Universitat Marburg.
- Atkinson, C., Draheim, D., 2013. Cloud-Aided Software Engineering: Evolving Viable Software Systems Through a Web of Views. In *Software engineering frameworks for the cloud computing paradigm*. Springer London. pp. 255–281.
- Broken windows theory. Wikipedia. [WWW] https://en.wikipedia.org/wiki/Broken_windows_theory (22.10.2017)
- Buldas, A., Kalu, A., Laud, P., Oruaas, M. (2017). Server-Supported RSA Signatures for Mobile Devices. In European Symposium on Research in Computer Security. Springer, Cham. pp. 315–333.
- Cockburn, A., 2001. *Writing Effective Use Cases*. Addison Wesley.
- Crain, A., 2002. Dear Dr. Use Case: Is the Clock an Actor? Rational edge
- Date, C.J., 2006. *The Relational Database Dictionary. A comprehensive glossary of relational terms and concepts, with illustrative examples*. O'Reilly.
- Date, C.J., Darwen, H., 2014. *Databasases, Types, and the Relational Model. The Third Manifesto*. 3rd edn.[WWW] http://www.dcs.warwick.ac.uk/~hugh/TTM/DTATRM.pdf (25.10.2017)
- Dont Repeat Yourself [WWW] http://wiki.c2.com/?DontRepeatYourself (25.10.2017)
- Double negative. Wikipedia [WWW] https://en.wikipedia.org/wiki/Double_negative (11.11.2017)
- Easterbrook, S., 2010. The difference between Verification and Validation. Serendipity. Applying systems thinking to computing, climate and sustainability, 29.11.2010 [WWW] http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/ (11.01.2018)
- Eessaar, E., 2014. A Set of Practices for the Development of Data-Centric Information Systems. In: Information System Development. Improving Enterprise Communication.: 22nd International Conference on Information Systems Development (ISD2013), Seville, Spain, September 2-4, 2013. Eds. José Escalona, M.; Aragón, G.; Linger, H.; Lang, M.; Barry, C.; Schneider, C. Switzerland: Springer. pp. 73-84.
- Eessaar, E., 2016. A Data-Centric Algorithm for Identifying Use Cases. In: Software Engineering Perspectives and Application in Intelligent Systems, Vol. 2: 5th Computer Science On-line Conference 2016 (CSOC2016), April 27 - 30, 2016. Eds. Silhavy, R., Senkerik, R., Oplatkova, Z.K., Silhavy, P., Prokopova, Z. Switzerland: Springer Verlag, pp. 303-316.
- Elaasar, M., Labiche, Y., 2012. Model Interchange Testing: A Process and a Case Study. In European Conference on Modelling Foundations and Applications. Springer Berlin Heidelberg. pp. 49–61.
- El-Attar, M., Miller, J., 2010. Improving the quality of use case models using antipatterns. *Software and Systems Modeling,* Vol. 9. No. 2., pp.141-160.
- Evitts, P., 2000. *A UML Pattern Language*. 1st ed. Indianapolis : Macmillan Technical.

- Fowler, M., 2009. TwoHardThings [WWW] https://martinfowler.com/bliki/TwoHardThings.html (22.10.2017)
- Gallen, W.V., 2012. Use Case Preconditions: A Best-Kept Secret? BAtimes, 16.10.2012 [WWW] https://www.batimes.com/articles/use-case-preconditions-a-best-kept-secret.html (22.10.2017)
- Gerlitz, T., Tran, Q.M., Dziobek, C., 2015. Detection and Handling of Model Smells for MATLAB/Simulink models. In MASE@ MoDELS. pp. 13–22.
- Jaakkola, H., Henno, J., Welzer-Druzovec, T., Thalheim, B., Mäkelä, J., 2016. Why Information Systems Modelling Is Difficult. In SQAMIA. pp. 29–39.
- Karwin, B., 2010. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming (Pragmatic Programmers)*. Pragmatic Bookshelf.
- LaToza, T.D., Venolia, G., DeLine, R., 2006. Maintaining Mental Models: A Study of Developer Work Habits. In Proceedings of the 28th international conference on Software engineering. ACM. pp. 492–501.
- Lehman, M.M., 1996. Laws of Software Evolution Revisited. Proceedings of 5th European Workshop, EWSPT '96 Nancy, France, October 9-11, 1996. LNCS Vol. 1149, Springer, Berlin, pp. 108–124.
- Mannaert, H., Verelst, J., Ven, K., 2012. Towards evolvable software architectures based on systems theoretic stability. *Software Pract. Exper*., Vol. 42, pp. 89–116.
- Martin, R.C., 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Pearson Education.
- McDonald, K., Matts, C., 2013. The Seven Information Smells of Domain Modelling. InfoQ, Apr 18, 2013 [WWW] https://www.infoq.com/articles/seven-modelling-smells (22.11.2017)
- Misbhauddin, M., Alshayeb, M., 2015. UML model refactoring: a systematic literature review. *Empirical Software Engineering*, Vol. 20, No. 1, pp. 206–251.
- OMG Unified Modeling Language ™ (OMG UML) Version 2.5.1, formal/17-12-05 [WWW] https://www.omg.org/spec/UML/2.5.1/ (19.02.2018)
- Principle of least astonishment. Wikipedia [WWW] https://en.wikipedia.org/wiki/Principle_of_least_astonishment (06.11.2017)
- Rattan, D., Bhatia, R., Singh, M., 2013. Software clone detection: A systematic review. *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199.
- Sonmez, J., 2012. Types of Duplication in Code. Simple Programmer, 27.05.2012 [WWW] https://simpleprogrammer.com/2012/05/27/types-of-duplication-in-code/ (30.09.2017)
- Störrle, H., 2013. Towards clone detection in UML domain models. *Software & Systems Modeling*, Vol. 12, No. 2, pp. 307–329.
- Surjective function. Wikipedia. [WWW] https://en.wikipedia.org/wiki/Surjective_function (20.02.2018)
- Vassiliadis, P., Zarras, A.V. Skoulis, I., 2017. Gravitating to rigidity: Patterns of schema evolution–and its absence–in the lives of tables. *Information Systems*, Vol. 63, pp. 24–46.
- Weber, B., Reichert, M., Mendling, J., Reijers, H.A., 2011. Refactoring large process model repositories. *Computers in Industry*, Vol 62, No 5, pp.467–486.
- Zachman, J.A., 2017. The Zachman Framework for Enterprise Architecture TM. The Enterprise Ontology, Version 3.0, [WWW] http://www.zachman.com/about-the-zachman-framework (22.10.2017)