On Using UML Diagrams to Identify and Assess Software Design Smells

Thorsten Haendler
Institute for Information Systems and New Media,
Vienna University of Economics and Business, Vienna, Austria
thorsten.haendler@wu.ac.at

Keywords: Software design smells, Unified Modeling Language (UML2), smell detection and assessment, code and design review, software design documentation, refactoring, architectural smells, technical debt management.

Abstract: Deficiencies in software design or architecture can severely impede and slow down the software development and maintenance progress. Bad smells and anti-patterns can be an indicator for poor software design and suggest for a refactoring of the affected source code fragment. In recent years, multiple techniques and tools have been proposed to assist software engineers in identifying smells and guiding them through corresponding refactoring steps. However, these detection tools so far only cover a modest amount of smells and also tend to produce false positives which represent conscious constructs with symptoms similar or identical to actual bad smells (e.g. design patterns). These and other issues in the detection process demand for a code or design review in order to identify (missed) design smells and/or re-assess detected smell candidates. UML diagrams are the quasi-standard for documenting software design and are often available in software projects. In this position paper, we investigate whether (and to what extent) UML diagrams can be used for identifying and assessing design smells. Based on a description of difficulties in the smell detection process, we discuss the importance of design reviews. We then investigate to what extent design documentation in terms of UML2 diagrams allows for representing and identifying software design smells. In particular, 14 kinds of design smells and their representability via UML class and sequence diagrams are analyzed. In addition, we discuss further challenges for UML-based identification and assessment of bad smells.

1 INTRODUCTION

Deficiencies in software design or architecture can severely impact the maintainability and extensibility of a software system (*technical debt*), see e.g. (Kruchten et al., 2012). Bad smells and anti-patterns can be an indicator for poor software design and suggest for a refactoring of the affected source code fragment (Fowler et al., 1999). Smells can be found at different levels (i.e. on level of software source code, software design, and software architecture). Software design smells, in particular, represent flaws in the software design by violating design rules (Suryanarayana et al., 2014). They can be categorized into ABSTRACTION, ENCAPSULATION, HIERARCHY, and MODULARIZATION smells.

In recent years, multiple tools and techniques have been proposed for assisting software engineers in detecting and assessing refactoring candidates as well as planning and performing refactoring steps, see, e.g. (Fernandes et al., 2016). Despite these efforts and advances, several difficulties in the process of smell detection and refactoring still demand for an individual assessment of smell candidates by human experts, also see, e.g. (Tempero et al., 2017). For instance, smell detectors only cover a modest amount of smell kinds and are mostly only available for selected programming languages, see, e.g. (Fontana et al., 2012; Fernandes et al., 2016). Moreover, smell detection tools also produce smell *false positives* (Fontana et al., 2016).

These issues in the detection process demand for a design or code review in order to identify (missed) design smells (*false negatives*), to re-assess detected candidates in order to discard *false positives* and/or to prioritize the candidates for refactoring, see, e.g. (Ribeiro et al., 2016). For this review, in software projects often an explicit software design documentation, e.g. in terms of diagrams of the *Unified Modeling Language (UML2)* (Object Management Group, 2015), can be consulted by software engineers to investigate the design quality of the software system.

In this position paper, we investigate the applicability of UML-based documentation for identifying and assessing software design smells. The hypothesis is that UML design diagrams are suitable as decision-support for recognizing software design issues. Based on an overview of difficulties in the process of detecting and assessing design smells, we focus on answering the question, whether and to what extent software design smells can be identified via reviewing UML-based design documentation of the system under analysis. For this purpose, we analyze 14 different kinds of software design smells reported in research literature (see, e.g. (Fowler et al., 1999; Suryanarayana et al., 2014)) regarding their representability via UML class and sequence diagrams (based on smell symptoms and relevant design context). Moreover, we discuss further challenges for using UML diagrams to identify and assess bad smells, such as the availability and quality of design documentation, the identifiability of false positives and alternative decision-support techniques.

Paper structure. The remainder of this paper is structured as follows. Section 2 gives an overview on several difficulties in detecting and assessing design smells in general for motivating the importance of design reviews. In Section 3, we investigate the identifiability of design smells via reviewing UML2 diagrams. In particular, we analyze the applicability of UML diagrams for representing 14 kinds of design smells. In Section 3.2, further challenges for the UML-based identification/assessment are discussed. Section 4 reflects on related work. In Section 5, the approach's limitations are discussed, and Section 6 concludes the paper.

2 DIFFICULTIES IN IDENTIFYING AND ASSESSING DESIGN SMELLS

The process of identifying candidates for refactoring can be roughly divided into the following two steps: *first*, detecting smell candidates and, *second*, assess the candidates in order to rule out whether the candidate should be refactored or not. Table 1 provides an overview of 14 kinds of software design smells with aspects relating to difficulties in identification and assessment. The smells represent a subset of the design smells covered by (Fowler et al., 1999) and (Suryanarayana et al., 2014)[1] and will later be examined regarding their representability in UML diagrams (see Section 3). For each smell kind, popular aliases (or very similar smells) used in research literature or industrial practice are listed. The smells are categorized by the design principle (or rule) which is violated by them (*Violated Design Principle*; i.e. ABSTRACTION, ENCAPSULATION, HIERARCHY, and MODULARIZATION). Also a short description of smell symptoms is provided, based on (Fowler et al., 1999; Suryanarayana et al., 2014).

Smell coverage of detection tools. In recent years, multiple tools and techniques addressing the detection of smells (*first step*) have been proposed. For an overview, see, e.g. (Fontana et al., 2012; Fernandes et al., 2016). For spotting smell candidates via symptoms, detection tools apply rules and threshold based on different metrics mostly by leveraging static program analysis techniques. In general, these tools come with certain limitations regarding the availability for programming languages (mostly only a few selected languages) and the quite moderate coverage of smell kinds, especially for smells that are categorized as design smells (see below).

In Table 1, the smell coverage of two popular smell detectors (*DECOR* and *JDeodorant* which analyze the source code using static analysis techniques) and of one popular UML model smell detector (*EMF Refactor*) are illustrated. The chosen detectors are exemplary, but representative regarding the amount of covered smell kinds; see, e.g. (Fontana et al., 2012). The detectors *DECOR* (Moha et al., 2010) and *JDeodorant* (Tsantalis, 2017) are available for *Java*-based programs only; both also provide automated refactoring for the detected smells.

- Among the 9 smells detected by *DECOR* only 4 can be categorized as design smells, which are in particular DATACLASS, LARGECLASS, MESSAGECHAIN and SPECULATIVEGENERALITY.
- *JDeodorant* covers 5 smells, of which 3 are design smells; i.e. FEATUREENVY, MULTIFACETEDABSTRACTION (in terms of GODCLASS), and DUPLICATEABSTRACTION (in terms of DUPLICATEDCODE).
- *EMF Refactor* is a Eclipse plugin e.g. for detecting and refactoring smells in UML models. In total, it addresses 27 kinds of smells, of which 6 can be seen as design smells; i.e. DATACLUMPS, LARGECLASS, SPECULATIVEGENERALITY, DIAMONDINHERITANCE, UNUSEDCLASS, and DUPLICATEABSTRACTION (but only in terms of EQUALLYNAMEDCLASS).

[1](Fowler et al., 1999) describe symptoms, causes and variants as well as refactoring options for 22 code smells, of which some are design smells. (Suryanarayana et al., 2014) destinguish 25 explicit software design smells.

Table 1: Overview of 14 software design smells categorized by design principles (violated by the smells) with aliases and symptoms. The smell coverage by popular smell detectors and the exemplary false positives illustrate the importance of design reviews for identifying and assessing smell candidates.

Violated Design Princ.	*Software design smell based on (Fowler et al., 1999; Suryanarayana et al., 2014)*	*Aliases (used in research or industry) and smells with similar symptoms*	*Symptoms description based on (Fowler et al., 1999; Suryanarayana et al., 2014)*	*DECOR*	*JDeodorant*	*EIMF Refactor*	*Smell false positives oriented to (Fontana et al., 2016)*
ABSTRACTION	DATACLUMP	(a kind of) MISSINGABSTRACTION	Clumps of data used instead of a unit (e.g. class)	–	–	√	–
	MULTIFACETEDABSTRACTION	LARGECLASS, GODCLASS, lack of cohesion	Unit (e.g. classes) with more than one responsibility	√*	√*	√*	STATE DP, generic class, e.g. configuration class, GUI widget toolkits
	UNUTILIZEDABSTRACTION	UNUSEDCLASS, SPECULATIVEGENERALITY	Not or barely used units (e.g. class or method)	√	–	√	recently developed program elements not yet covered by tests, null implementation
	DUPLICATEABSTRACTION	CODECLONE, DUPLICATEDCODE, functionally similar methods (as kind of DUPLICATEABSTRACTION)	Multiple units (classes or methods) with identical (or similar) internal and/or external structure or behavior	–	√*	√*	inherited or overridden method
ENCAPS.	DEFICIENTENCAPSULATION	Hideable public attributes or methods	The accessibility of attributes or methods is more permissive than actually required	–	–	–	–
	LEAKYENCAPSULATION	–	A unit that exposes implementation details via its public interface	–	–	–	–
HIERARCHY	SPECULATIVEHIERARCHY	SPECULATIVEGENERALITY, speculative general types	One or more types in a hierarchy are used speculatively (only based on imagined needs)	√	–	√	–
	UNNECESSARYHIERARCHY	TAXOMANIA (taxonomy mania)	A variation between classes is mainly/only captured in terms of data (structural features)	–	–	–	–
	DEEPHIERARCHY	DISTORTEDHIERARCHY	An unnecessarily deep hierarchy	–	–	–	–
	MULTIPATHHIERARCHY	REPEATEDINHERITANCE, DIAMONDINHERITANCE	A subtype inherits both directly and indirectly from a supertype	–	–	√	–
MODULARIZATION	FEATUREENVY	(a kind of) BROKENMODULARIZATION, Misplaced operations	Methods are more interested in features owned by foreign classes than in features of the owning class	–	√	–	VISITOR DP, STRATEGY DP, DECORATOR DP, PROXY DP, ADAPTER DP
	DATACLASS	(a kind of) BROKENMODULARIZATION, RECORDCLASS, DATACONTAINER	Classes providing data but having no (or only few) methods for operating on them	√	–	–	EXCEPTIONHANDLINGCLASS, LOGGERCLASS, SERIALIZABLECLASS, configuration class, Data Transfer Object (DTO)
	CYCLICDEPENDENDMODULARIZATION	CYCLICDEPENDENCY, (DEPENDENCY) CYCLES	Two or more units (e.g. classes, methods) mutually depend on each other	–	–	–	VISITOR DP, OBSERVER DP, ABSTRACTFACTORY DP
	MESSAGECHAIN	(a kind of) BROKENMODULARIZATION	A client unit (e.g. method) calls another unit, which then in turn calls another unit, and so on (navigation through class structure)	√	–	–	BUILDER DP, FACADE DP, test-class method

As also can be seen from Table 1, some of the selected smell kinds are not covered or are only covered partially.

Smell false positives. Due to the ambivalence of metrics-based smell detection, tools tend to produce false positives. The symptoms which indicate a bad smell can also be the result of code and design constructs, which have been consciously implemented by a software engineer. (Fontana et al., 2016) identified based on a literature review such *false positives* for 12 code smells and anti-patterns, which represent mainly ABSTRACTION and MODULARIZATION design smells, and provide corresponding false positives . These smell false positives can be categorized as follows (Fontana et al., 2016):

- *imposed anti-patterns and smells* (consciously implemented) which are, e.g. the result of applying a design pattern or imposed by using a specific programming language or framework, or by performing optimizations.
- *inadvertent anti-patterns and smells* (created by tools) which are, e.g. caused by source-code generators or program representation, or result in the analysis scope.

We extended this list based on experience from software projects (see the row at the right in Table 1). For an overview of design patterns, also see (Gamma et al., 1995).

Design reviews. The difficulties reflected above illustrate that even by applying automated decision-support tool for smell detection further human investigation is necessary in order to identify *false negatives* (i.e. smells not detected by tools) and/or to (re-)assess detected candidates for discarding *false positives*. In addition, it is often necessary for software design evaluation to include contextual knowledge on the design rationale provided by design experts such as software architects. For this purpose, code and design reviews are performed. However, the direct manually investigation of the source code can be tedious and erroneous, especially for large software systems. In addition, since some design issues do not manifest via the source code, it can become difficult to identify them via a code review alone.

For this reason, in software projects often design documentation based *Unified Modeling Lan-*

guage (UML2) is available which represents the *quasi-standard* for documenting software design. The UML provides notations for modeling structural and behavioral software design aspects.[2]

3 IDENTIFYING SOFTWARE DESIGN SMELLS IN UML DIAGRAMS

3.1 Representability of Design Smells in UML Diagrams

In this section, we analyze the applicability of UML2 class and sequence diagrams to provide relevant information for software engineers to identify and/or assess software design smells during a design review. The *Unified Modeling Language* (Object Management Group, 2015) provides different diagrams types for modeling/documenting structural and behavioral (and other) aspects of primarily object-oriented software systems.

Based on several studies, see, e.g. (Arisholm et al., 2006; Scanniello et al., 2018), there is evidence that the availability of design documentation in software projects in terms of UML diagrams enhances the comprehensibility of program source-code and can lead to significant improvements regarding the functional correctness of modification tasks, especially for complex tasks (such as the identification and refactoring of design smells).

For our investigation, we focus on UML2 class and sequence diagrams, since they are the most common in industry projects for modeling structural and behavioral design aspects respectively, see, e.g. (Laitenberger et al., 2000):

- In particular, UML class diagrams represent the class structure of a software system (e.g. with attributes and operations) and relations between these classes (e.g. in terms of associations, generalizations, and dependencies).
- UML sequence diagrams in turn represent interactions between class instances (i.e. objects) at run-time. The objects are represented by lifelines which interact by mutually exchanging messages (e.g. method calls). This way, sequence diagrams allow for documenting the intended or actual behavior of objects during (system) usage scenarios.

[2]For other decision-support techniques in design investigation (such as quality-analyzer tools), also see Section 3.2.

For investigating the UML-based representability of the design smells, we describe the minimal *structural and behavioral design scope* which include all elements that are affected by the smell symptoms.[3] Depending on the kind of smell, this relevant context can include structural and/or behavioral design aspects of the system under analysis. Based on the design context, we then present exemplary UML class and sequence diagrams. For the purpose of comprehensibility, the diagrams are simple, synthetic and syntactically reduced. Figs. 1, 2, 3 and 4 depict the structural and behavioral design scopes with exemplary UML-based representation in terms of UML class and sequence diagrams. The smell symptoms are highlighted in red. For each smell, we then discuss, whether the design smell can be represented via the UML diagrams and whether reviewing the diagram allows for identifying the smell. This is, we reflect, whether the corresponding diagram provides the information needed for the smell identification. In addition, where appropriate, we reflect on other relevant difficulties for smell identification and assessment, especially based on the UML diagrams.

ABSTRACTION smells. Fig. 1 depicts the four ABSTRACTION smells with exemplary UML diagrams reflecting the structural and behavioral design scopes.

- DATACLUMP: The class diagram does not allow for identification of the smell, since no information on the data usage is available. Based on multiple sequence diagrams reflecting different usage scenarios, a repeated joint usage of the data can indicate a DATACLUMP smell.
- MULTIFACETEDABSTRACTION: A class diagram can weakly indicate a candidate for MULTIFACETEDABSTRACTION, e.g. by multiple relationships (e.g. dependencies) from/to other classes. This information can be complemented by sequence diagrams (reflecting different usage scenarios), which might illustrate that a class is interacts with certain other classes in different scenarios, which would be an indicator for multiple responsibilities.
- UNUTILIZEDABSTRACTION: In case that a class has no (or very little) relationships to other classes, a class diagram indicates such a UNUTILIZEDABSTRACTION. Given that the relationships are defined in source code, but never actually used at run-time, the sequence diagrams

[3]For smell symptoms, see Table 1; for other aspects of smell detection/assessment and refactoring (such as potential causes, variants, or refactoring techniques), please consult research literature such as (Fowler et al., 1999; Suryanarayana et al., 2014).

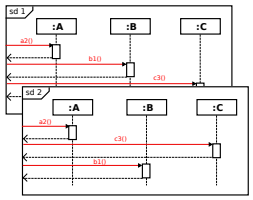
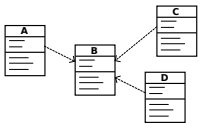
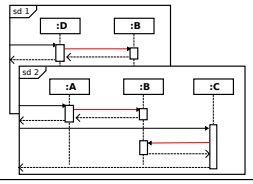
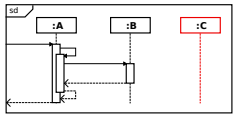
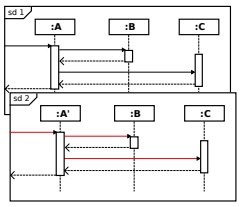
Design Smell	*Structural Scope*	*Exemplary Class Diagram*	*Behavioral Scope*	*Exemplary Sequence Diagram*
DataClump	Classes owning the candidate data, e.g., getter methods **a2()**, **b1()**, and **c3()**, used during a specific or multiple usage scenarios		Interactions between instances of owning classes during usage scenario(s) *(indicating that methods **a2()**, **b1()** and **c3()** are repeatedly used together, in multiple scenarios)*	
MultifacedAbstraction	Candidate class (**B**) with used/using methods or attributes and corresponding using/ used methods with owning foreign classes		Interactions of candidate class (**B**) with multiple classes during different usage scenarios *(indicating that **B** holds probably multiple responsibilities)*	
UnutilizedAbstraction	Candidate class (**C**) with using classes/methods (if rarely used)		In case of rarely used class, interactions of it. *(else, no interaction available for instances of class **C**)*	
DuplicateAbstraction	Candidate class (**A'**) with owned features (e.g., methods and attributes) and relationships (if any) *(indicating syntactical clone)*		Interactions of instances of candidate class (**A'**) in terms of calls from and to the lifeline *(indicating functional similar clone)*	

Figure 1: Selected ABSTRACTION smells with exemplary UML class and sequence diagrams reflecting the structural and behavioral design scope and smell symptoms (see Tab. 1. Smell symptoms are highlighted in red.

can indicate candidates for functionally UNUTILIZEDABSTRACTIONs.

- DUPLICATEABSTRACTION: Candidates for syntactical duplicates can be spotted via class diagrams by comparing the owned features and relationships. In addition, sequence diagrams allow for identifying candidates for semantical duplicates in terms of functionally similar clones. For instance, in case, the sequences of calling and called method (and the types of the passed arguments) are identical, a candidate is identified.

For all four ABSTRACTION smells (except for the syntactical variants of UNUTILIZEDABSTRACTION and DUPLICATEABSTRACTION) that sequence diagrams reflecting the usage scenarios are necessary to identify the corresponding smell candidates.

ENCAPSULATION smells. Fig. 2 depicts the DEFICIENTENCAPSULATION smell. A class diagram alone provides information on the access modifier of the candidate feature. To investigate how the attribute is actually used, sequence diagrams reflecting usage scenarios can, e.g. indicate that a candidate feature is not used by other classes, which indicated a DEFICIENTENCAPSULATION smell.

HIERARCHY smells. Fig. 3 depicts the four addressed HIERARCHY smells.

- SPECULATIVEHIERARCHY: Based on the class diagram alone, it can not be seen whether a hierarchy is speculative, sine no information on the actual usage are available. The sequence diagram, in addition, might indicate that the inherited features of the candidate class are never actually used.
- For the three HIERARCHY smells UNNECESSARYHIERARCHY, DEEPHIERARCHY and MULTIPATHHIERARCHY class diagrams obviously can indicate the corresponding smell candidates via generalization relationships; sequence diagrams can not be used for identification.

MODULARIZATION smells. Fig. 4 depicts the different MODULARIZATION smells addressed in this analysis with corresponding UML examples.

- FEATUREENVY: Based on a class diagram alone, a FEATUREENVY candidate can not be spotted, since the UML does not provide elements to model dependencies between features. The relationships (including dependencies) between classes do not indicate a FEATUREENVY. How-

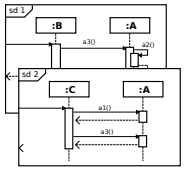
Design Smell	*Structural Scope*	*Exemplary Class Diagram*	*Behavioral Scope*	*Exemplary Sequence Diagram*
DeficientEncapsulation	Candidate attributes and/or methods (**a2()**) with owning class *(indicating that feature is publicly available)*		All interactions with class of candidate attribute/method *(indicating that **a2()** is not used by other classes)*	

Figure 2: Selected ENCAPSULATION smells with exemplary UML class and sequence diagrams reflecting the structural and behavioral design scope and smell symptoms (see Tab. 1. Smell symptoms are highlighted in red.

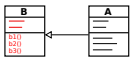
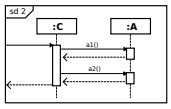
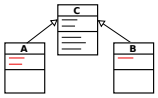
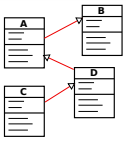
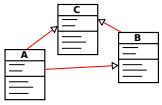
Design Smell	*Structural Scope*	*Exemplary Class Diagram*	*Behavioral Scope*	*Exemplary Sequence Diagram*
SpeculativeHierarchy	Candidate class (**B**) with subclasses (and superclasses)		All Interactions of subclasses (here **A**) *(indicating that features of candidate class **B** are not used)*	
UnnecessaryHierarchy	Candidate classes **A** and **B** (with features) and subclasses *(indicating that variability of subclasses only in terms of attributes)*			
DeepHierarchy	Candidate class (inheriting subclass, class **C**) with all involved superclasses *(indicating a deep hierarchy)*			
MultipathHierarchy	Candidate class (inheriting subclass, class **A**) with all involved superclasses *(indicating multiple hierarchy paths to superclass **C**)*			

Figure 3: Selected HIERARCHY smells with exemplary UML class and sequence diagrams reflecting the structural and behavioral design scope and smell symptoms (see Tab. 1. Smell symptoms are highlighted in red.

ever, a sequence diagram reflecting the method calls triggered by the candidate method during one or multiple usage scenarios might illustrate that more foreign methods are used than by the own class and can indicate a FEATUREENVY.

- DATACLASS: A class diagram can show that a class has no methods at all. In addition, a sequence diagram can indicate DATACLASS candidates which have methods, but which do not access the own data.
- CYCLICALLYDEPENDENTMODULARIZATION: Relations (e.g. associations and dependencies) can indicate a circular dependency between classes. Also in sequence diagrams these dependencies can be illustrated via corresponding messages between lifelines.
- MESSAGECHAIN: Chains of relationships between classes in class diagrams do not indicate a MESSAGECHAIN smell. In contrast, messages in sequence diagrams can obviously illustrate them with their depth level.

Preliminary findings on UML-based representability of design smells. The examples shown above illustrate that all selected design smells with symptoms can be represented by combining UML class and sequence diagrams. In particular, in order to express relationships between classes, the UML provides different kinds of `Relationship`, such as `Association` and `Dependency`, see (Object Management Group, 2015). In contrast, it does not allow for expressing dependencies between `Operations` (in terms of method-call dependencies). For identifying dependency-related symptoms (especially relevant for MODULARIZATION, ABSTRACTION, and also ENCAPSULATION SMELLS) the sequence diagrams can provide additional information by representing method/feature calls in terms of sequences of (mutual) `Messages`, which allows for investigating the details of method-call dependencies, e.g. for identifying method-based CYCLICDEPENDENCIES. A prerequisite for this is that the sequence diagrams reflect actual/intended usage scenarios. Moreover,

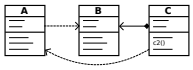
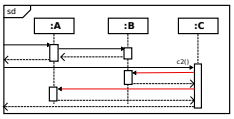
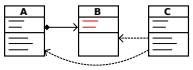
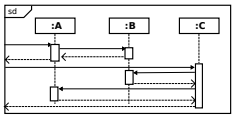
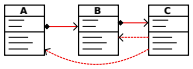
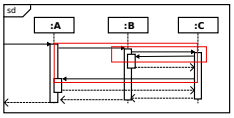
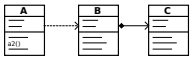
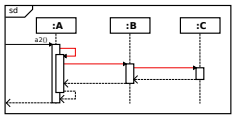
Design Smell	*Structural Scope*	*Exemplary Class Diagram*	*Behavioral Scope*	*Exemplary Sequence Diagram*
FeatureEnvy	Candidate method (**c2()**) and its owning class as well as used features with classes		All method calls triggered by the candidate method **c2()** during a usage scenario *(indicating that more foreign features are used than by own class)*	
DataClass	Candidate class (**B**) with provided data (features) and the classes using the data *(indicating that no methods exist)*		All interactions of the candidate class (**B**) *(indicating that B uses no methods for operating on own data)*	
CyclicDependency	All classes with features involved in dependency cycle *(indicating structural dependencies; **B** and **C** direct, **A** and **C** indirect)*		All inter-class method calls between the involved classes during a usage scenario *(indicating cyclic call dependencies, **B** and **C** as well as **A** and **C**)*	
MessageChain	Calling and called methods in the chain with owning classes		All method calls triggered (directly and transitively) by candidate method **a2()** during a specific usage scenario *(indicating a chain with depth of 3)*	

Figure 4: Selected MODULARIZATION smells with exemplary UML class and sequence diagrams reflecting the structural and behavioral design scope and smell symptoms (see Tab. 1. Smell symptoms are highlighted in red.

class diagrams provide techniques for identifying HIERARCHY smells via the `Generalization` relationship between sub-type and super-type.

3.2 FURTHER IDENTIFICATION AND ASSESSMENT CHALLENGES

In the following, we reflect on selected further challenges for UML-based smell identification and assessment.

Locating the relevant design context. Manually created and maintained UML diagrams often lack with regard to up-to-dateness and consistency with the documented software system. In contrast, (automatically) reverse-engineered UML diagrams (especially by applying dynamic analysis techniques) come with the problem of large model size and a high detail level which impedes comprehending the diagram (Fernández-Sáez et al., 2015). For this reason, in recent years, different techniques for interactively exploring or configuring the scope of the diagrams have been proposed, see, e.g. (Bennett et al., 2008; Haendler et al., 2015). Independent from the method of creating the design documentation, still the challenge remains to locate the relevant part of a design diagram (*design scope*, see above), especially in case of assessing a given smell candidate.

Distinctiveness of smells. Another aspect is the distinctiveness of the symptoms represented via the UML diagrams. As seen by the examples in Section 3, the diagrammatic representation alone would presumably not allow to identify and to distinguish specific kinds of smells, or even false positives. For example, the examples of class diagrams for FEATUREENVY, CYCLICDEPENDENCY and DATACLASS have a very similar appearance. However, it becomes clear that a UML-based visualization not in every case supports the comprehension of the analyzed issue.

4 RELATED WORK

To the best of our knowledge, so far there is no research addressing the specification or identification of software design smells via UML diagrams. However, closely related is research aiming at modeling code smells and corresponding refactorings in UML diagrams:

The model-smell detector *EMF Refactor*, see, e.g. (Arendt et al., 2009), provides several techniques for assuring the model quality of *Ecore* and *UML2* models. In particular, the tool provides 23 quality metrics for Ecore models which cover 3 kinds of smells for Ecore models and 22 corresponding refactoring techniques. For UML2 models, it provides in total 107 metrics which cover 27 model smells. The addressed smells largely do not reflect software-design issues as reflected in this paper (see Section 2). The

study reported in (Arendt and Taentzer, 2010) focuses on model smells for the early stage of a model-based software-development process. Among other things, Arendt et al. present a catalog with 17 UML model smells consisting of a description, detection techniques, refactorings, quality characteristics affected, and an example represented in terms of a UML diagram. (Rojas et al., 2017) analyze the effects of creating and refactoring smells in conceptual models (based on smell definitions by *EMF Refactor*) on the technical debt of the underlying source code measured by applying *SonarQube* (Campbell and Papapetrou, 2013). They mainly focus on the refactoring effort in correlation with the measured TD. Within this, they also map correlations between smells in *Java* source code and model smells in UML class diagrams. However, both studies do not address design smells as described in this paper and do not reflect on their representability.

Moreover, approaches are related that evaluate the impact of applying UML diagrams for the localization of design defects and for performing software maintenance activities in general. (Laitenberger et al., 2000) investigate in a controlled experiment how design defects can be located via UML diagrams. As a result, they provide mappings between defects and diagram types. However, they address defects that can not be categorized as design smells and pursue a more general approach based on model quality attributes (such as completeness or consistency). For instance, (Arisholm et al., 2006; Scanniello et al., 2018) investigated in empirical studies that the availability of UML-based design documentation enhances the comprehensibility of the system source code, especially with regard to performing complex maintenance tasks.

As a complement to both groups of research, we present a conceptual investigation of the applicability of UML diagrams for identifying software design smells.

5 DISCUSSION

In this position paper, we report on work-in-progress and present results of an investigation on the applicability of UML2 diagrams for identifying and assessing software design smells. The analysis has a conceptual and exploratory character. It provides a first systematization with preliminary results that demand for empirical evaluation which will we approached in future work.

We only focused on UML2 class and sequence diagrams, because of their popularity in industry and since they are most common for structural and behavioral aspects respectively on the design level. However, other diagram types of the UML2 such as state charts, activity diagrams or component diagrams might also serve as a basis for the identification of certain smell kinds, which should be investigated in further research.

For the purpose of comprehensibility, simple, synthetic and syntactically reduced UML diagram examples have been presented. Moreover, we focused on an exemplary set of 14 software design smells with representatives for each violated design principle.

6 CONCLUSION

In this paper, we investigated whether UML2 class and sequence diagrams provide the information needed for identifying and assessing 14 kinds of software design smells. In particular, we analyzed the smell representability by creating synthetic diagram examples which reflect the minimal structural and behavioral design context to include all system elements affected by the corresponding smell. In addition, we discussed further challenges for UML-based smell identification and assessment. As a first result of this exploratory approach can be stated that by combining UML class and sequence diagrams all selected kinds of software design smells with their symptoms could be represented. However, the examples also illustrate that an identification of design smells or a distinction between the different smells via reviewing the UML diagrams provides some difficulties, since the diagrammatic appearance of smell types can be partially very similar. Especially, it seems challenging to recognize patterns for identifying design smells in UML diagrams, due to the various manifestations of smell symptoms.

This analysis represents a first step to investigate the possibilities of a UML-based smell evaluation. For future work, we plan to validate the findings in an empirical setting by comparing the occurrence of software design smells detected in source code with their appearance in corresponding reverse-engineered UML-based design documentation. Moreover, we aim to develop an intelligent tutoring system (ITS) for guiding software engineers in acquiring techniques for assessing and refactoring design smells. For decision support in refactoring, we plan to provide reverse-engineered and tailorable UML diagrams, also see (Haendler et al., 2017).

REFERENCES

Arendt, T., Mantz, F., Schneider, L., and Taentzer, G. (2009). Model refactoring in eclipse by LTK, EWL, and EMF refactor: a case study. In *Model-Driven Software Evolution, Workshop Models and Evolution*.

Arendt, T. and Taentzer, G. (2010). UML model smells and model refactorings in early software development phases. *Universitat Marburg*.

Arisholm, E., Briand, L. C., Hove, S. E., and Labiche, Y. (2006). The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381.

Bennett, C., Myers, D., Storey, M.-A., German, D. M., Ouellet, D., Salois, M., and Charland, P. (2008). A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software: Evolution and Process*, 20(4):291–315.

Campbell, G. and Papapetrou, P. P. (2013). *SonarQube in action*. Manning Publications Co.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 18. ACM.

Fernández-Sáez, A. M., Genero, M., Chaudron, M. R., Caivano, D., and Ramos, I. (2015). Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: A family of experiments. *Information and Software Technology*, 57:644–663.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *J. Object Technology*, 11(2):5–1.

Fontana, F. A., Dietrich, J., Walter, B., Yamashita, A., and Zanoni, M. (2016). Antipattern and code smell false positives: Preliminary conceptualization and classification. In *Proc. SANER'16*, volume 1, pages 609–613. IEEE.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Haendler, T., Sobernig, S., and Strembeck, M. (2015). Deriving tailored UML interaction models from scenario-based runtime tests. In *International Conference on Software Technologies*, pages 326–348. Springer.

Haendler, T., Sobernig, S., and Strembeck, M. (2017). Towards triaging code-smell candidates via runtime scenarios and method-call dependencies. In *Proceedings of the XP2017 Scientific Workshops*, pages 1–9. ACM.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21.

Laitenberger, O., Atkinson, C., Schlich, M., and El Emam, K. (2000). An experimental comparison of reading techniques for defect detection in UML design documents. *Journal of Systems and Software*, 53(2):183–204.

Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Object Management Group (2015). Unified Modeling Language (UML), Superstructure, Version 2.5.0. [last access: June 8, 2018].

Ribeiro, L. F., de Freitas Farias, M. A., Mendonça, M. G., and Spínola, R. O. (2016). Decision criteria for the payment of technical debt in software projects: A systematic mapping study. In *ICEIS (1)*, pages 572–579.

Rojas, G., Izurieta, C., and Griffith, I. (2017). Toward technical debt aware software modeling. In *IEEE-ACM Ibero American Conference on Software Engineering, CibSE*, pages 22–35.

Scanniello, G., Gravino, C., Genero, M., Cruz-Lemus, J. A., Tortora, G., Risi, M., and Dodero, G. (2018). Do software models based on the UML aid in source-code comprehensibility? aggregating evidence from 12 controlled experiments. *Empirical Software Engineering*, pages 1–39.

Suryanarayana, G., Samarthyam, G., and Sharma, T. (2014). *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.

Tempero, E., Gorschek, T., and Angelis, L. (2017). Barriers to refactoring. *Communications of the ACM*, 60(10):54–61.

Tsantalis, N. (2017). JDeodorant. [last access: June 8, 2018].