13th International Conference on Software Technologies (ICSOFT 2018)

On Using UML Diagrams to Identify and Assess Software Design Smells

Thorsten Haendler

Vienna University of Economics and Business (WU Vienna)

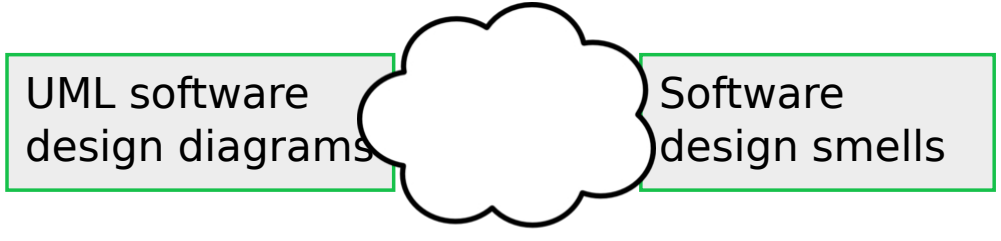
thorsten.haendler@wu.ac.at

deficiencies in software design/architecture can impede development progress

→ software refactoring important, but understood as difficult task (*Tempero et al., 2017*)
identification of *bad smells* as step in the refactoring process

software design smells violating software design principles, i.e., *Abstraction*, *Encapsulation*, *Modularization* and *Hierarchy* (*Suryanarayana et al., 2014*)

diagrams of the *Unified Modeling Language (UML)* as *quasi-standard* for software design documentation



→ ***Hypothesis***:
UML diagrams are suitable for identifying and assessing software design smells

3 Questions

1. *Why are design reviews important/necessary?*
→ **Relevance** of design reviews for smell identification

2. *Are UML diagrams suitable for identifying software design smells?*
→ **Representability** of software design smells via UML diagrams

3. *What other aspects need to be considered?*
→ Further **challenges** for UML-based smell identification

Relevance of design reviews 1/4: **smell coverage**

at least 25 software design smells (*Suryanarayana et al., 2014*)

smell coverage of popular smell detection tools:

	covered smells in total	covered design smells
DECOR	9	4
JDeodorant	5	3
EMF Refactor	27	6

In addition: smell detection tools mostly only available for a few programming languages

→ *manual* ***identification of missed smell candidates*** *(false negatives)*

Relevance of design reviews 2/4: **smell false positives**

smell symptoms can also be the result of conscious constructs (*Fontana et al. 2016*)

2 groups:
→ imposed smells, e.g., as result of applying a design pattern
→ inadvertent smells, e.g., as result of development tools (e.g., code generators)

Fontana et al.: false positives for at least 7 kinds of software design smells

→ *manual **assessment** of tool-identified smell candidates in order to discard false positives*

Relevance of design reviews 3/4: **examples**

Cyclic-dependend Modularization (aka *CyclicDependency*)

→ not detected by any of the three smell detectors
→ false positive: e.g., *Visitor* design pattern with double-dispatch protocol (visitor and visited element)

MessageChain (as kind of *BrokenModularization*)

→ only detected by *DECOR*
→ false positive: e.g., *Facade* design pattern

Relevance of design reviews 4/4: **code vs. design reviews**

issues described above illustrate
→ human investigation in terms of code/design review necessary

but: investigation of source code directly is tedious and error-prone

some design issues do not manifest via source code
→ source code only static representation (no run-time behavior)
→ abstraction from code to design level necessary

UML diagrams popular and often available in software projects
→ with notations for structural and behavioral design aspects

Representability of software design smells via UML diagrams 1/4: **objective**

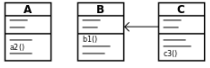
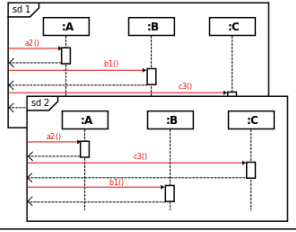
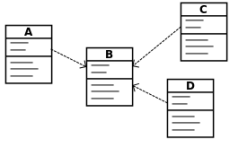
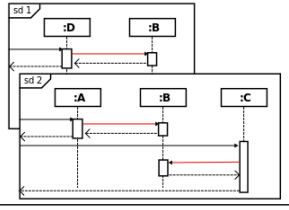
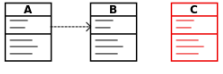
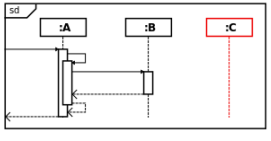
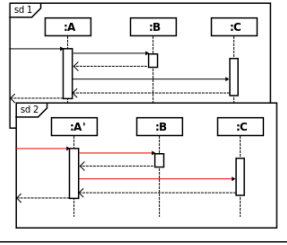
UML class and sequence diagrams most popular for representing structural and behavioral design aspects respectively

Objective:
investigate whether diagrams provide the information needed for representing and identifying the smells

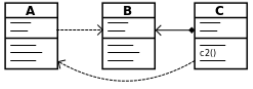
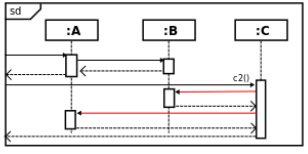
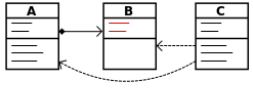
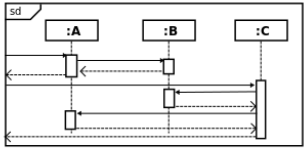
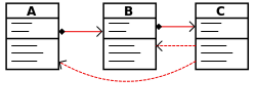
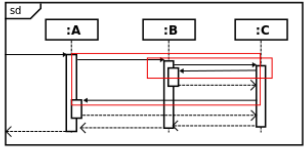
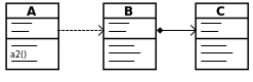
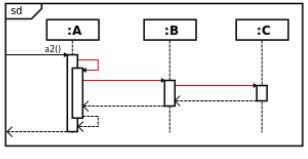
minimal structural and behavioral design scope
→ all elements affected by the smell symptoms as described by *(Fowler et al. 1999)* and (*Suryanarayana et al., 2014*)

exemplary synthetic UML diagrams reflecting the identified design scope

Representability of software design smells via UML diagrams 2/4: ***Abstraction* smells**

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)*	

Representability of software design smells via UML diagrams 3/4: ***Modularization* smells**

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)*	

Representability of software design smells via UML diagrams 4/4: **preliminary findings**

by only reviewing UML class diagrams most design smells are not identifiable (except a few *Hierarchy* smells)

Problems:
- limited expressiveness of dependencies class diagrams (e.g., **`Dependency`** as relationship between **`Classifiers`**, not between methods/fields
- some smells are indicated by behavioral aspects, e.g., for expressing responsibilities or usage contexts/scenarios (e.g., *MultifacedAbstraction*, *DataClump*)

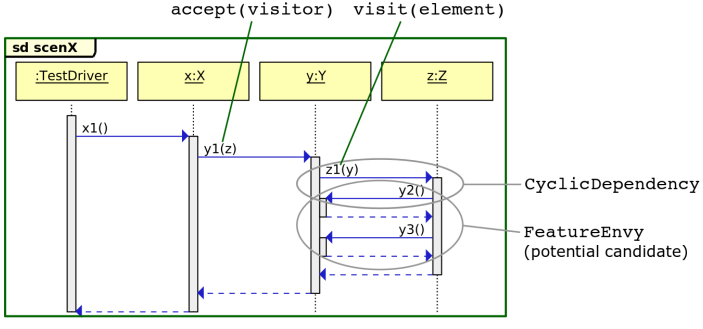
by combining UML class and sequence diagrams (which reflect usage scenarios)
→ all symptoms and design scopes of selected design smells can be represented

Further challenges for UML-based smell identification

Consistent and up-to-date UML-based design documentation
→ approaches for reverse-engineering UML diagrams

Locating the relevant design context
especially in large reverse-engineered diagrams
→ approaches for interactively exploring/tailoring the diagrams
but: no approaches for exploration/tailoring with regard to the smell scope

Distinctiveness of design smells
among each other (very similar appearance)
to identify false positives (e.g., visitor pattern)
→ demands for experienced
software design experts



Conclusion and next steps 1/2

Summary:

difficulties in smell detection demand for design reviews, e.g., via UML diagrams

we investigated the representability of 14 kinds of software design smells via UML class and sequence diagrams

Preliminary results:
UML class diagrams mostly not sufficient (except for a few *Hierarchy* smells)
→ but by combining UML class and sequence diagrams all selected design smells can be represented

further challenges exist, e.g., in identifying the relevant design scope, especially in UML sequence diagrams reflecting multiple usage scenarios

Conclusion and next steps 2/2

Discussion:
conceptual and exploratory character
→ but first systematization with preliminary results

Limited regarding: diagram types, kinds of smells and selected examples (small, synthetic)

Future Work:
→ empirical study for comparing occurrence in source code with appearance in reverse-engineered UML design diagrams

→ intelligent tuoring system (ITS) for supporting in acquiring techniques for smell assessment and refactoring; with tailorable UML diagrams as decision support, also see (*Haendler et al., 2017*)

Thank you for your attention!

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Questions & Discussion

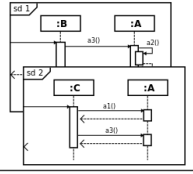
thorsten.haendler@wu.ac.at

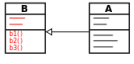
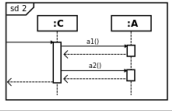
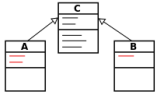
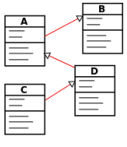
EFMD EQUIS ACCREDITED · AACSB ACCREDITED · ASSOCIATION OF MBAS ACCREDITED

Appendix 1/2: Difficulties in smell identification

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	EMF 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)	–	–	√	–
	MultifacedAbstraction	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

Appendix 2/2: Representability of software design smells via UML diagrams: ***Encapsulation* and *Hierarchy* smells**

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)*	

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**)*