Smell Detection in Context

Daniel Speicher
Computer Science III, University of Bonn
dsp@cs.uni-bonn.de

Sebastian Jancke
SOPTIM AG
sebastian.jancke@soptim.de

Good design is essential for long-term evolvability of software systems. To reduce the required additional short-term effort appropriate tools are highly desirable. The most promising approach [3] combines basic metrics to build complex heuristics to detect refactoring opportunities (*design smells*). We still saw - backed by a questionnaire survey in [5] - the need to systematically improve this approach. The problems we want to address are:

1. Insufficient handling of false positives
2. Information overload
   (a) Irrelevance in the context of the current task
   (b) Irrelevance because of historic stability
3. Isolated presentation

The later problems are human computer interaction issues. The first is of systematical nature and requires rethinking of the definition of design smells. Our core hypothesis is:

Hypothesis H1 *Context sensitive smell detection eases the identification of refactoring opportunities during frequent refactorings.*

This hypothesis can be refined into three further hypotheses with respect to three different kinds of context, addressing the three problems.

Hypothesis H2 *A* **structural context** *exploiting relations between defined structures and smells reduces false positives in an adaptive way.*

If a smell detection tool criticizes the code of a developer, it may better be right. If it isn't, it creates additional work that might outweigh the benefits of the automated smell detection. The least such a tool could offer, would be the ability to allow the developer to explicitly deny the existence of a smell at a certain location[1]. But this still creates additional effort, because the exception needs to be rethought on every change and by every developer who works on this code.

[1]As it is possible with compiler warnings for many languages and finally now with Java using the `SuppressWarning` annotation.

But how can a tool - developed based on a lot of expertise - be wrong anyway? We see the main reason in the inability of the tool to guess the design intentions of a developer. Some smells are the result of trade-offs: Certain design qualities were preferred over other design qualities. Therefore we suggest to annotate the design decision, so that the tool can take it into account and the developers can easily review them.[2]

This approach creates the additional effort to define the structures and to annotate (or sometimes automatically detect) the code, only that this additional effort causes benefits in the form of explicit design documentation and possible additional consistency checks.

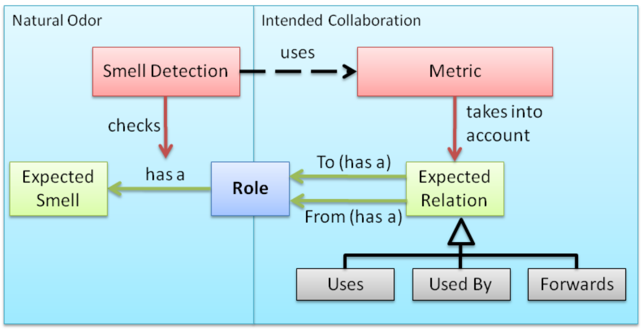


Figure 1: Integration of *natural odors* and *intended collaboration* into the smell detection mechanisms

In [1] one of the authors discussed these trade-offs for the classical design patterns as well as for other documented structures. This discussion lead to a list of smells (*natural odors*[3]), that can and should be ignored.

[2]The detection strategy for Intensive Coupling in [3, p. 121] presents a precursor to this idea: Too many method calls to a few unrelated classes are only considered a smell, if the calling method has at least a few nested conditionals. The reason they give is, that initialization methods and creation methods typically show this coupling although this doesn't cause any problems. We agree with the later, but suggest that the user should be involved in making this exception.

[3]**Data class:** Element in Visitor Pattern, Data Structure in Procedural Components. **Refused Parent Bequest:** Concrete Decorator in Decorator Pattern, Component in Composite Pattern. All smell names following [3].

For smells that are based on coupling measures we declare which dependencies are *intended collaborations*.[4] The metric calculation can then take this knowledge into account and ignore these dependencies while fully counting any other dependency. To illustrate intended collaborations let us consider two examples:

The **Visitor pattern** places functionality that operates on the the data of certain objects (Elements) in separate classes (Visitors). One reason for this separation is, that the Elements build a complex object structure and the functionality belongs rather to the whole structure than to single Elements. Another reason might be, that the functionality is expected to change more frequently and/or is used only in specific configurations. Since the functionality in the Visitor accesses the data of the Elements, this intended collaboration could falsely be identified as Feature Envy.

```
bind(String.class)
  .annotatedWith(Names.named("csvSequence"))
  .toInstance(Simulations.SUCCESS_SCENARIO);
```

Figure 2: Example usage of an EDSL

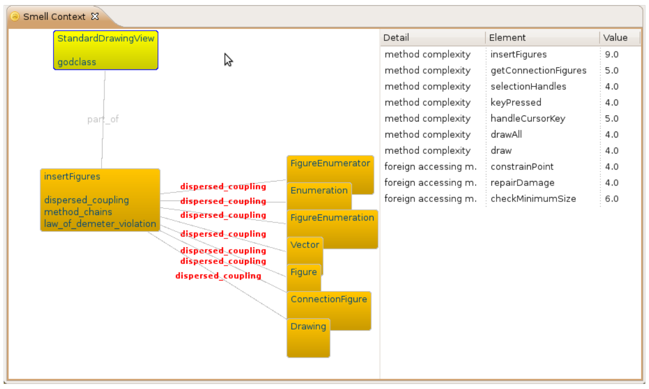
A Domain Specific Language (DSL) is a small language allowing clear and efficient code for a narrow focused domain. If the DSL is created using an existing host language it is called **Embedded DSL** (EDSL). EDSLs offer high readability by nesting and chaining method calls. However, method chaining leads to the violation of a decoupling principle called "Law of Demeter" [4]. This principle restricts method calls to types which are contained or instantiated in the respective type, or are a type of a parameter or a field. EDSLs trade decoupling for better readability.

Hypothesis H3 *Using* **relevance** *contexts (from task [a] and history [b] information) decreases the information load during smell detection and makes it manageable for regular refactoring.*

A smell can be temporarily irrelevant, because the developer is focusing on a task that doesn't involve the module that has the smell. An IDE can support the developer by only showing the interesting artifacts. [2] suggests the following degree-of-interest (DOI) model: An artifact is considered as "interesting", if it is viewed or edited by the developer. It becomes stepwise uninteresting, if the developer works on other artifacts. If a module in the repository has been stable for a long time, this might be as well a reason to ignore its smells. Therefore we extended the DOI model to repositories. Instead on user interaction, the calculation is here based on changes in the repository.

Hypothesis H4 *Visualizing the* **co-location** *of a smell with other smells helps to understand design problems during regular refactoring.*

Smells are often co-located, i.e. can be found in the same module or in dependent modules. This co-location can be incidental or even intrinsic, as e.g. any Data Class will cause its clients to have Feature Envy. Therefore a visualization of the co-located smells is helpful.



Detail	Element	Value
method complexity	insertFigures	9.0
method complexity	getConnectionFigures	5.0
method complexity	selectionHandles	4.0
method complexity	keyPressed	4.0
method complexity	handleCursorKey	5.0
method complexity	drawAll	4.0
method complexity	draw	4.0
foreign accessing m.	constrainPoint	4.0
foreign accessing m.	repairDamage	4.0
foreign accessing m.	checkMinimumSize	6.0

Figure 3: Co-located smells in a "God Class"

Case Study Based on our logic based infrastructure for code analysis [6] we implemented a prototype. It provides the detection heuristics presented in [3] and extended it with the three dimensions of context we mentioned here. A systematically conducted case study for four developer days on a typical industrial software system supports our hypotheses, that integration of context is necessary and useful. Only the relevance filtering based on the temporal degree of interest (H3 b) didn't prove useful because the task relevance filtering provided already enough focus.

References

[1] S. Jancke. Smell detection in context, diploma thesis. University of Bonn, 2010.

[2] M. Kersten. *Focusing knowledge work with task context.* PhD thesis, University of British Columbia, 2007.

[3] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer, 1 edition, 9 2006.

[4] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. *ACM SIGPLAN Notices*, 23(11):323–334, 1988.

[5] E. Murphy-Hill. *Programmer Friendly Refactoring Tools.* PhD thesis, Portland State University, 2009.

[6] D. Speicher, T. Rho, and G. Kniesel. JTransformer - eine logikbasierte Infrastruktur zur Codeanalyse. In *Proc. 9. Ws. Software-Reengineering (WSR 2007)*, pages 21–22, 2007.

[4]**Middle Man:** Abstract Decorator in Decorator Pattern, Facade Pattern, Adapter Pattern, Mediator Pattern. **Feature Envy:** Concrete Visitor in Visitor Pattern, Concrete Strategy in Strategy Pattern, Creation Methods, Service in Procedural Components, Presenter in Model View Presenter, Unit Tests. **Intensive Coupling, Dispersed Coupling:** Facade Pattern, Mediator Pattern, Client of Embedded DSL, View in Model View Presenter, Unit Test. **Method Chain, "Law of Demeter" Violation:** Client of Embedded DSL. **Shotgun Surgery:** Application Programming Interface.