Evolving Software Quality Knowledge

Daniel Speicher

Computer Science III, University of Bonn, Bonn, Germany

Abstract. Instead of having a system of rigid quality criteria, we suggest to co-evolve the knowledge about good and bad design with the code. Based on an infrastructure that represents object-oriented code in a logic factbase, we describe how to defined code critiques ("bad smells") and well established structures ("design pattern") and how to make the bad smells aware of the design pattern. A case study on ArgoUML shows that it is more effective to find unjustified warnings by taking developers knowledge into account then by structural criteria.

1 Introduction

Good software design improves maintainability, evolvabilty and understandability. As any maintenance or evolution step requires the developer to understand the software reasonably good, understandability is the most crucial of these qualities. Therefore it can not be a goal to develop detection strategies for design flaws that a developer does not need to understand to use them. How could a criterion for understandability be meaningful, if it is not understandable itself? Developers should know and understand the detection strategies they use. In this paper we want to argue, that in addition detection strategies should know more of what developers know. It is essential for automated design flaw detection to be adaptable to respect developers knowledge found in the design.

Detection strategies for design flaws need to be generic, as they are meant to apply to many different systems. On the other hand software solutions are at least to some part specific. If there were no need for specificity, it would not require many developers to build software. This does not yet say, that generic quality criteria are inappropriate, as there might be - and probably are - some general principles that should be met always. What it does say, is that developers answer to specific design challenges, so that there is at least some probability that there are good reasons to make a different choice in a specific situation than one would make while discussing design in general. We will present such (moderately) specific situations.

We wrote this article on the background of established Refactoring literature. With a broader acceptance of object-oriented programming at the end of the last century programmers needed advice to build systems with high maintainability and evolvability. Today the catalog of signs for refactoring opportunities (bad smells) [1] developed by Beck and Fowler as well as the catalog of proven design solutions (design pattern) [2] developed by Gamma, Helm, Johnson and Vlissides are common software engineering knowledge.

Speicher D..
Evolving Software Quality Knowledge.
DOI: 10.5220/0003699100360047
In *Proceedings of the 2nd International Workshop on Software Knowledge* (SKY-2011), pages 36-47
ISBN: 978-989-8425-82-9

Marinescu devised in [6] a method to translate the informal descriptions of bad smells [1] and object-oriented design heuristics [7] into applicable so called detection strategies. The method first decomposes the informal descriptions into parts, which are then translated into an expression built of metrics and comparisons with thresholds. These expressions are then composed into one rule with the help of the elementary logical operators. Marinescu and Lanza elaborate in [5] how developers should systematically work through the bad smells (here called disharmonies) and step by step resolve cohesion problems ("identity disharmonies") then coupling problems ("collaboration disharmonies") and finally problems in the type hierarchy ("classification disharmonies"). Developers are in general well equipped with this book and established reengineering und refactoring literature [8], [1], [4].

Overview. The rest of the paper is organized as follows. Section 2 will discuss, how the same design fragment can be a well respected design pattern and an instance of a well know bad smell. As the choice of the design pattern is expected to be the result of trade-offs, the smell detection should be enabled to take the knowledge about the pattern into account and ignore this smell instance. Section 3 therefore develops our approach based logic meta programming. We represent the Java code base as Prolog facts, on which we define detection strategies and design pattern as predicates. Finally we explain, how we suggest to incorporate the knowledge about pattern into the knowledge about smells. Section 4 shortly suggests how to use our infrastructure to develop a process of stepwise evolution of design knowledge. Section 5 finally reports about a case study, that gives an example where taking developer intentions into account, strongly increases the precision of a detection strategy.

2 Visitors Tend to have Feature Envy

Objects bring data and behavior together. This is at the core of object-orientation and allows developers to think of objects in programs similar as of real objects. The deviation from this ideal are the smells *data class* and *feature envy*. A class is a data class if the class has mainly data and only little behavior. A method in a class has feature envy, if it operates for the major part on data of another class. Still in some situations one wants to separate the data and the behavior of one conceptual object into two or more technical objects, which can result in both smells.

The *Visitor pattern* as in Fig. 1 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.

There are variations of the same data-behavior separation in other design pattern, as listed in Tab. 1. We separate data into an extra object, if we want other objects to share this data. So does the ExtrinsicState in the Flyweigth Pattern and the Context in the Intepreter Pattern. As a result the classes which use this data (Flyweight in Flyweight,

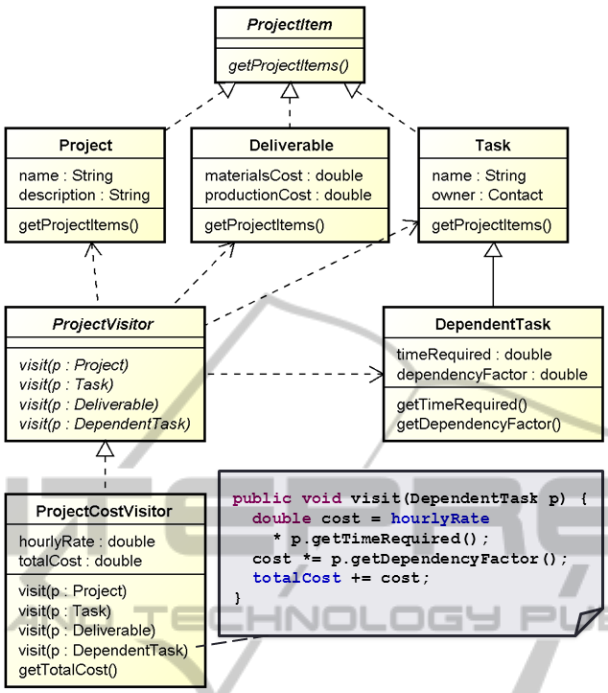


Fig. 1. A simple Visitor: The `ProjectItems` build a tree structure via the `getProjectItems()` method. The `ProjectCostVisitor` implements the single responsibility to calculate the total costs of a project. He accesses the data of the `ProjectItems` to fulfill this responsibility.

Expressions in Interpreter) develop Feature Envy. In the Memento Pattern some data of the Originator is stored in a separate data class called Memento. We separate behavior from the data, if we want to dynamically change it as we do by exchanging one ConcreteState for another in the State Pattern. The more data is left and accessed in the Context class, the stronger the Feature Envy will be. Finally, if we want to let Colleague classes interact with each other without knowing about each other the ConcreteMediator might operate on the data of a few of the Colleagues.

3 Logic based Code Analysis

In the following we want to introduce the overall architecture of our prototype.

3.1 Structures

Our implementation of the structures behind the metrics and of the structures of design pattern are similar. They consist of a set of predicates that generates - given an handle element - a labeled graph.

Table 1. Data-Behavior Separation: Roles in the pattern that can develop a smell as a consequence of strong data-behavior separation.

Pattern	Data Class	Feature Envy
Flyweight	ExtrinsicState	Flyweight
Interpreter	Context	Expression
Mediator		ConcreteMediator
Memento	Memento	Originator
State	Context	ConcreteState
Strategy		ConcreteStrategy
Visitor	Element	ConcreteVisitor

The smell *feature envy* analyses how strongly a method operates on fields of another class. This analysis involves can be seen as the analysis of a certain graph with nodes for the method and the own and foreign attributes and as well for the access relation.

The pattern *visitor* can be seen as graph with nodes for the single abstract visitor and the single abstract element and a few nodes for the concrete visitors and concrete elements. Access relations could be seen as part of this graph, but this is not necessary for our purpose.

Definition 1 (Structure Definition). *A structure definition consists of*

- *one unary predicate that tests whether a program element is the* handle *of a structure,*
- *some binary predicates that tests whether given the handle of a structure, another program elements plays a certain* role *in the structure with this handle,*
- *and some ternary predicates that tests whether given the handle, a* relation *between two other program elements that play a role in this structure holds true.*

Definition 2 (Structure). *Given a structure definition and a program element that fulfils the handle predicate, we call this program element the* handle of the structure*, where the structure consists of*

- role players*, which are all program elements annotated with the name of a role predicate of the structure definition, for which the role predicate is true, if we use the handle as a first argument and the role player as a second argument;*
- relation player duos*, which are all pairs of program elements annotated with the name of a relation predicate of the structure definition, for which the relation predicate is true, if we use the handle as a first argument, the first element of the pair as second argument, and the second element of the pair as a third argument.*

3.2 Fact Representation of Object-oriented Code

Our prototypes are implemented as plug-ins for development environment Eclipse. They contain a builder that runs everytime the builder of the Java Development Tool is run and translates the complete Java Abstract Syntax Tree (AST) of a program into a representation in Prolog facts. In this AST all references are resolved and all language elements of Java 5 are represented. Figure 2 shows the facts representing the first

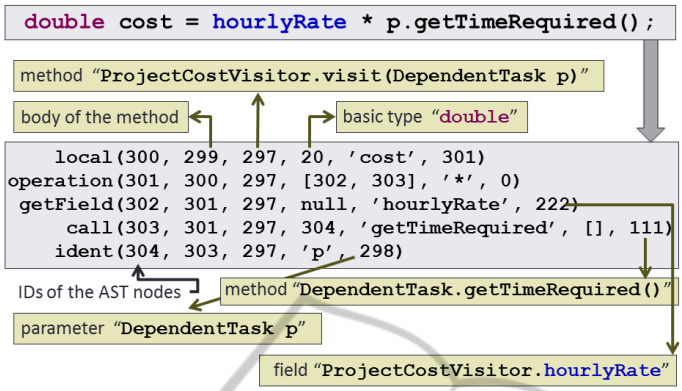


Fig. 2. Fact representation of the first line of the method `ProjectCostVisitor.visit(DependentTask)`.

statement in method `visit(DependentTask)` in `ProjectCostVisitor`. Each fact represents a node of the AST. The first parameter of a fact is a unique ID that is used to reference this node. The second parameter is a back reference to the parent node. The third parameter references the enclosing method. The remaining parameters contain some attributes as well as references to the child nodes. It is very easy to build on this fact representation more abstract predicates, as for example in listing 1.1:

```
method_contains_call(M, C) :- call(C, _, M, _, _, _, _).
call_calls_method(C, M)    :- call(C, _, _, _, _, _, M).
```

Listing 1.1. Derived Predicates.

In the following we will use many predicates of this kind without defining them. The names of the predicates should convey their meaning.

3.3 Smell Detection Strategies

Smell Detection Strategies as defined by Marinescu and Lanza in [5] are elementary logical formulas build of comparisons of metrics with corresponding thresholds. To illustrate our approach we will explain the detection strategy for the smell *Feature Envy* top-down. The detection strategy reads

```
feature_envy(M) :-
    feature_envy_structure(M),
    access_to_foreign_data(M, ATFD),       ATFD >  2,
    locality_of_attribute_access(M, LAA), LAA  <  0.3,
    foreign_data_provider(M, FDP),         FDP  =< 5.
```

The first goal verifies that M is a method for which we can calculate the feature envy. It is build of three metrics *Access To Foreign Data* (The number of directly or indirectly accessed fields in a foreign class), *Locality of Attribute Access* (The ratio to which, the accessed fields are from the own class.) and *Foreign Data Provider* (The number of foreign classes from which the methods accesses a field.) are calculated.

These metrics simply count the number of certain role player in the structure, as the source code shows:

```
access_to_foreign_data(M, Value) :-
   count(F, method_accesses_foreign_field(M, M, F), V).
locality_of_attribute_access(M, V) :-
   count(F, method_accesses_own_field(M, M, F),     AOF),
   count(F, method_accesses_foreign_field(M, M, F), AFF),
   Value is AOF / (AOF + AFF).
foreign_data_provider(Method, Value) :-
   count(C, method_accesses_foreign_class(M, M, C), V).
```

The metrics build on the role (*own class*, *own field*, *foreign class*, *foreign field*) and relation (*method accesses own field*, *method accesses foreign field*) definitions of the feature envy structure. The first lines of the definition read like follows:

```
feature_envy_structure(S) :-
    source_method(S), not(abstract(S)).
method(S, M) :-
    feature_envy_structure(S), S = M.
own_class(S, C) :-
    method(S, M), method_is_in_type(M, C).
own_field(S, F) :-
    own_class(S, C), type_contains_field(C, F),
    modifier(F, private).
[...]
method_accesses_foreign_field(S, M, F) :-
   method(S, M),
   foreign_field(S, F),
   once(method_accesses_field(M, F)).
[...]
```

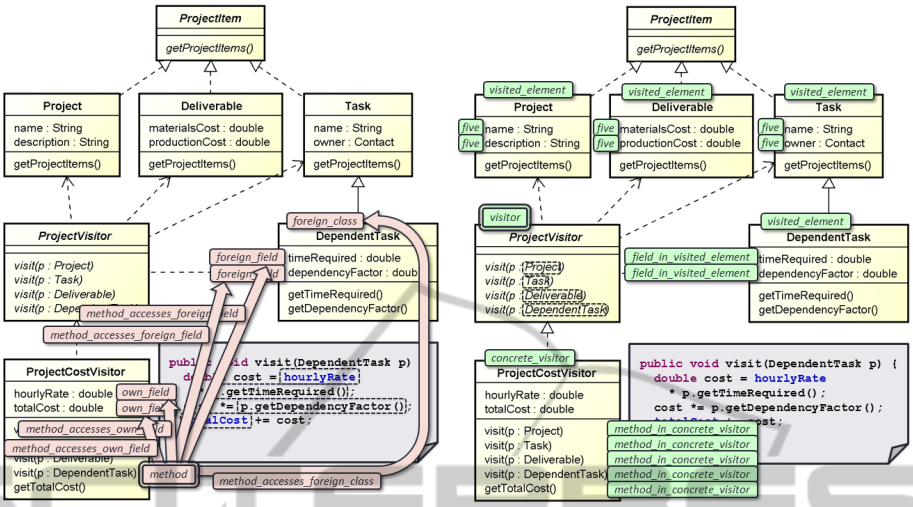
Listing 1.2. Feature Envy Structure (Excerpt).

We illustrated this structure in Figure 3. The complete structure consists of the roles and the relations.

3.4 Lightweight Pattern Definition

To make our smell detection pattern aware, we need only a very lightweight pattern definition. The structure for the pattern consists of the roles *visitor*, *concrete visitor*, *method in concrete visitor*, *visited element* and *field in visited element*. The complete definition reads like follows:

```
visitor_pattern(P) :-
     declared_as_visitor(P).
visitor(P, V) :-
     visitor_pattern(P), P = V.
concrete_visitor(P, C) :-
    visitor(P, V), sub_type(V, C), not(interface(C)).
method_in_concrete_visitor(P, M) :-
```



(a) Feature Envy Structure (b) Lightweight Visitor Pattern

Fig. 3. The instance of the visitor pattern overlayed with: (a) The feature envy structure for the method `visit(DependentTask)` in the `ProjectCostVisitor`. (b) The lightweight pattern definition for the visitor pattern with the handle `ProjectVisitor`. "five" is an abbreviation for "field in visited element".

```
    concrete_visitor(P, C), type_contains_method(C, M).
visited_element(P, E) :-
    visitor(P, V), type_contains_method(V, M),
    method_has_parameter(M, R), parameter_has_type(R, E).
field_in_visited_element(P, F) :-
    visited_element(P, E), type_contains_field(E, F).
```

Listing 1.3. Visitor Pattern.

Note that we impose very little constraints on the elements and relations within the pattern. The idea is to focus in a first step on the identification of the elements and relations, describing the extension of the design pattern. The predicates are means to capture the intended extension of the developer under the assumption that he expressed it well enough. Typically developers use standard names or name parts at least for some of the role players in a pattern and we found that all other role players can be identified starting from one of these. Alternatively one could require that one or a few role players are annotated with the role.

To let the developers tie their class via a naming convention to the pattern, there should be predicate like:

```
declared_as_visitor(V) :- class_name_ends_with(V, 'Visitor'),
    not(subclass(V, P), class_name_ends_with(P, 'Visitor')).
```

To tie it to the pattern via an annotation, another predicate should be used:

```
declared_as_visitor(V) :- class_annotated_with(V, 'Visitor').
```

Defining what it actually means for these elements to form a design pattern is a second step. This distinction allows us to say that the role players implement a design pattern incorrectly in contrast to just saying, that the pattern is not there. And we may even evolve our knowledge about the *intension* of a design pattern (What it means to be a design pattern) separate from the knowledge about the *extension* of a design pattern (Which elements of the program a meant to form the design pattern.) The goal that a developer wants to achieve, the *intention* of the pattern, should be seen as part of the intension.

3.5 Intention Aware Smell Detection

To make the smell detection aware of the possible intentions, we add a check into the respective predicates at the earliest possible place, i.e. as soon as the variables are bound. For this purpose we define predicates `intended_field_access`, `intended_method_call`, `natural_odor`. Here is the adapted code for the relation *method accesses foreign field* of the *feature envy structure* and the *dataclass*.

```
dataclass(C) :-
    named_internal_type(C),
    not(natural_odor(dataclass, C)),
    weight_of_class(C, WOC), WOC < 3.34,
    [...]
method_accesses_foreign_field(S, M, F) :-
   method(S, M),
   foreign_field(S, F),
   not(intended_field_access(M, F)),
   once(method_accesses_field(M, F)).
```

Listing 1.4. Intention aware relation and smell.

Given this adaptation and the definition of the pattern, it is easy to make the smells ignore the intended field access and the natural odor:

```
natural_odor(dataclass, Element) :-
   concrete_element(_, Element).
intended_field_access(M, F) :-
   visitor_pattern(P),
   method_in_concrete_visitor(P, M),
   field_in_visited_element(P, F).
```

Listing 1.5. 'Publishing intentions and natural odors'.

The smell dataclass will now ignore any class that plays the role of a concrete element in the visitor pattern. In the calculation of the feature envy structure all accesses from a method in a concrete visitor to a field in a visited element will be ignored. That is, feature envy towards other classes will still be detected.

As a side note: Having to adapt all the different relations and smell definitions is not desirable. An aspect-oriented adaptation would be very helpful here and Prolog is very well suited to be enhanced with Aspects.

Another way to adapt the smells is to use thresholds that depend as well on the roles an element plays. We will not discuss this option although it is obviously considerable.

4 Conflicts Stimulate Knowledge Evolution

We suggested to use our technology for an alternating process of code review (quality improving) and code documenting (quality knowledge improving). For the process of quality improvement [5] give an excellent guideline. For the process of improving the explicit quality knowledge, we gave a first suggestion here. Currently we expect the strongest stimulus to increase the design knowledge in unjustified smell warnings. Making the design explicit allows the smell to ignore it, while providing much more information than a single declaration that just asks to ignore this smell instance (like with the @SuppressWarnings in Java 5). Of course in the longer run, this design knowledge needs reviewing as well. We would suggest to make further expectations explicit and verify them. For example there should be no dependencies from any Concrete Element in the Visitor Pattern to any Concrete Visitor. Another expectation is, that the Element classes can stay much more stable than the visitors do. Making this expectation explicit and executable, will allow for a third feedback cycle.

5 Evaluation

5.1 Smell in Pattern

Sebastian Jancke implemented the detection strategies as well as a few more as part of his diploma thesis [3]. The instances of Natural Odors he found are listed in table 2.

Table 2. Natural Odors Found in Open Source Software Projects.

Smell	Role/Concept	Source Code
Feature Envy	Concrete Visitor	JRefactory 2.6.24, SummaryVisitor
Feature Envy	Concrete Strategy	JHotDraw 6, ChopBoxConnector
Middleman	Abstract Decorator	JHotDraw 6, DecoratorFigure
Law of Demeter Violation	Embedded DSL	Google Guice 2.0
Shotgun Surgery	(stable) API	[everywhere]

5.2 Taking Advantage of Expressed Intentions: Creation Methods

In [5, Ch. 6] ”Collaboration Disharmonies” Lanza and Marinescu reference design knowledge in a way, that is unique within their detection strategies. The two strategies *Intensive Coupling* and *Dispersed Coupling* contain besides conditions about the coupling an additional condition ”Method has few nested conditionals” measured by $\mathrm{MAXNESTING} > \mathrm{SHALLOW}$, where $\mathrm{MAXNESTING}$ is the maximal nesting depth of blocks in the method and $\mathrm{SHALLOW}$ is 1. The authors motivate this condition as follows:

> “Additionally, based on our practical experience, we impose a minimal complexity condition on the function, to avoid the case of configuration operations (e.g., initializers, or UI configuring methods) that call many other methods. These configuration operations reveal a less harmful (and hardly avoidable) form of coupling [. . .].” [5, p.121]

This motivation references the concept of operations (methods) for configuration. Although this concept is not precisely defined, the description gives every experienced OO programmer a first operational impression about it. The condition is interesting because it makes an exception with the reference to this explicit concept that is not just a language level concept.

Therefore we wanted to test this statement with a little case study using the current version of the same source code that was used in [5][1]. To discuss this statement we call methods with $\mathrm{MAXNESTING} = 1$ *flat* and present the claim in form of three hypotheses:

Flat methods are configuration methods. (1)

It is safe to ignore the coupling in flat methods. (2)

It is safe to ignore the coupling in configuration methods. (3)

A quick view at the source code showed that the hypothesis (1) is wrong. Many of the flat methods are obviously test methods and some are neither test nor configuration methods. It turned out that the code was well designed enough, so that we could rely on naming conventions. Exploring twenty randomly and a few systematically chosen methods following these naming conventions we were convinced that the two following statements were true for the code under consideration:

In ArgoUML methods with names starting with “init”, “create”, “build” or “make” are configuration methods. (4)

In ArgoUML methods with names starting with “test” or containing the term “Test” in their name or in the name of the enclosing class of the method are test methods. (5)

Given this two name based rules and the detection strategies for *Intensive Coupling* and *Dispersed Coupling* (without the condition of the methods being flat) we were able to classify the methods. The result is presented in Table 3.

Table 3. Methods in ArgoUML: The maximal nesting within the method and the classification into configuration, test and other methods influences whether the method has *Intensive* or *Dispersed Coupling*.

	All Methods				Intensive Coupling				Dispersed Coupling			
Max. Nesting	config	test	other	Σ	config	test	other	Σ	config	test	other	Σ
1	772	880	6416	8068	19	57	26	102	27	143	32	202
2	318	110	2468	2896	88	97	19	204	60	24	213	297
> 2	152	53	2128	2333	209	234	29	472	52	34	568	654
Σ	1242	1043	11012	13297	316	388	74	778	139	201	813	1153

On the first view Hypothesis (1) seems to be backed by the data, as many ($772/1242 = 62\%$) configuration methods are indeed flat and most ($1090/1242 = 88\%$)

[1] http://argouml.tigris.org/, accessed in June 2011. The 13 projects referenced in the team project set file “argouml-core-projectset.psf” were used and all 13297 non abstract methods of the 2083 named classes were analyzed. For [5] the version from October 2004 was used.

have nesting not bigger than 2. Unfortunately these configuration methods build only a small fraction ($772/8068 = 10\%$ or $1090/10964 = 10\%$) of the methods with limited nesting, so that hypothesis (1) is not true, as we already said. Still, there are many ($8068/13297 = 61\%$) flat methods, so that excluding them from further analysis can improve performance. We further observe, that the major part of the methods with intensive coupling are configuration methods ($316/778 = 41\%$) and test methods ($388/778 = 50\%$). The major part ($781/1153 = 68\%$) of the methods with dispersed coupling are other methods with a nesting of at least 2.

We still have to discuss whether this is safe to ignore flat methods, configuration methods and test methods. The coupling in configuration methods is indeed hardly avoidable as all the decoupled classes need to be instantiated and connected somewhere. That is, the coupling in some specific methods is a natural consequence of the overall decoupling effort across the system. Even if the responsibility for configuration is "extracted" into XML configuration file, the coupling is still part of the system although not part of the source code in the original programming language.

To test Hypothesis (3) that coupling in configuration methods can be ignored, we reviewed the 13 configuration methods with the highest coupling intensity ($22 - 116$) that have one of the smells: Even they are clearly understandable and the coupling is not harmful. The same is true for the 13 test methods with the highest coupling intensity ($28 - 68$).

To challenge the nesting condition, we reviewed the 13 methods with highest coupling intensity ($10 - 16$) within the other methods with no nesting, but with one of the smells. Our impression was not that clear as in the two cases before, but still the coupling did not require any refactoring[2]. Therefore we see no reason to reject Hypothesis (2).

To summarize, we expect all smells in configuration methods and all test methods to be false positives. The same is true for all methods with no nesting, i.e. Hypothesis (2) and (3) are plausible. The nesting condition reduces the smell results by $102/778 = 13\%$ or $202/1153 = 18\%$ while ignoring configuration and test methods reduces the results by $704/778 = 90\%$ or $340/1153 = 29\%$. So, if smell detection can use other information than structure (e.g. naming conventions) to identify configuration methods and test methods, the number of false positives can be strongly reduced.

```
configuration_method(M) :-
    declared_as_configuration_method(M).
natural_odor(intensive_coupling, E) :-
    configuration_method(E).

declared_as_configuration_method(M) :-
    source_method(M), method_name(M, N),
    member(P, ['init', 'create', 'build', 'make']),
    starts_with(N, P).
```

[2] Indeed half of them turned out to be a sort of configuration method again. So restricted (!) to the methods with smells the nesting condition could be a reasonable heuristic to detect configuration methods, i.e. (1) is true.

6 Contributions

This paper showed that automated bad smell detection should take developer intentions into account, as different structural quality criteria are appropriate, depending on these intentions. To illustrate this point we discussed a well known design pattern and how it is still good design even if it shows a smell. These intentions are often already expressed in the code, but not yet available to the automated analysis. We presented a technological and conceptual framework that allows to combine the perspectives of structures that should be avoided (bad smells) and structures that are useful building blocks (design pattern). We presented our approach to implement structures based on logic meta-programming and explained how smell detection can be made aware of existing structures in code. We suggested to use our technology for an alternating process of code review (quality improving) and code documenting (culture improving). A case study showed that the precision can be increased if the design knowledge that developers already made explicit is utilized instead of guessing developer intentions from structural properties.

Acknowledgements

The author wants to thank everyone who made this research possible. Sebastian Jancke developed the “smell detection in context” plug-in, an earlier version of the prototype presented here, and evaluated the usability benefits of this approach. My colleague Jan Nonnen, current and former diploma students as well as former participants of our labs contribute to the evolution of “Cultivate”, our platform code quality evaluation and coding culture evolution. Tobias Rho, Dr. G“unter Kniesel and further students develop “JTransformer” our platform that makes logic meta programming for Java possible and comfortable. Finally the author wants to thank Prof. Dr. Armin B. Cremers for letting the author work on this interesting topic.

References

1. Martin Fowler. Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley, Boston, MA, January 1995.
3. Sebastian Jancke. Smell detection in context, diploma thesis. University of Bonn, 2010.
4. Joshua Kerievsky. Refactoring to Patterns. Pearson Higher Education, 2004.
5. Michele Lanza and Radu 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.
6. Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In ICSM ’04: Proceedings of the 20th IEEE International Conference on Software Maintenance, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.
7. Arthur J. Riel. Object-Oriented Design Heuristics. Addison-Wesley Professional, 5 1996.
8. O. Nierstrasz S. Demeyer, S. Ducasse. Object Oriented Reengineering Patterns. Morgan Kaufmann, July 2002.