Code Quality Cultivation

Daniel Speicher

University of Bonn, Computer Science III
dsp@acm.org

Abstract. Two of the meanings of the word "cultivation" that are rather unrelated show a strong dependency, when applied to the domain of code quality:
The existing code in an evolving software system could be seen as the soil in which new code and new functionality is growing. While working this "soil" developers benefit from unobtrusively presented automatic feedback about the quality of their code. There are tools that verify the correct usage of good code structures ("design pattern") and other tools that highlight improvement opportunities ("bad smells").
As design patterns and bad smells are usually presented and discussed separately it has not been observed, that they partially contradict with each other. We will show that even well chosen design patterns can lead to bad smells. Thus, design quality is relative, which does not mean that it is arbitrary. Design quality knowledge has to be rendered more precisely. We suggest to co-evolve the quality knowledge specifications together with the code in a process of cultivation. Bad smell definitions can then easily be extended by taking existing design patterns into account.
When the design knowledge is cultivated together with the code, specific knowledge like typical method names can be incorporated. A case study explored unjustified "intensive coupling"-smells in ArgoUML: While a previously suggested generic structural criterion identified 13% unjustified warnings, taking the specific names into account, identified 90%.

Keywords: Code Quality, Design Pattern, Bad Smell, Natural Odor, Logic Meta-Programming, Case Study, Knowledge Evolution

1 Introduction

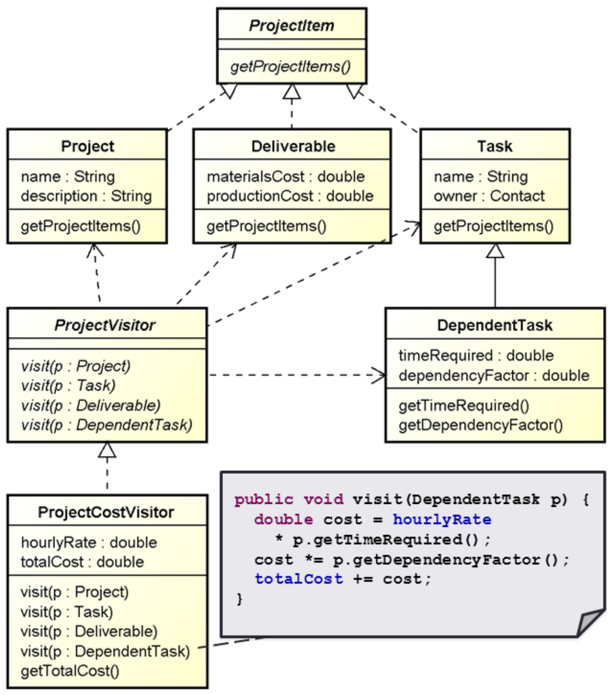
Good software design improves maintainability, evolvabilty and understandability. As any maintenance or evolution step requires the developer to understand the software reasonably good, understandability is the most crucial of these qualities. Therefore it can not be a goal to develop detection strategies 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") [6] developed by Beck and Fowler as well as the catalog of proven design solutions ("design pattern") [7] developed by Gamma, Helm, Johnson and Vlissides are common software engineering knowledge.

Marinescu devised in [16] a method to translate the informal descriptions of bad smells [6] and object-oriented design heuristics [21] 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 [15] 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 [2], [6], [12].

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 known 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 on 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 sketches the twofold "cultivation" process we envision. Quality knowledge specifications provide feedback to the developers and developers refine quality knowledge where precision is useful. 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. Section 6 collects some related work about design pattern definition, bad smell detection, logic meta-programming. Section 7 summarizes our contributions.



```
public void visit(DependentTask p) {
  double cost = hourlyRate
    * p.getTimeRequired();
  cost *= p.getDependencyFactor();
  totalCost += cost;
}
```

Fig. 1: A simple example of the *visitor* pattern: 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.

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 mainly has 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[1] 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

[1] This example was taken with some minor modifications from (accessed in 09/2011): `http://www.java2s.com/Code/Java/Design-Pattern/VisitorPattern1.htm`.

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*.

Before we continue with the discussion of other design patterns, we want to review this judgement again, as it is not impossible that the authors of the original pattern catalog suggested imperfect solutions and better design patterns would not have a flaw like *feature envy*. Indeed there have been thorough discussions about the *visitor* pattern and a variety of alternatives had been suggested [17], [9], [19]. Yet, the criticism was never about *visitors* accessing data of *elements* but only about the dependencies from the *abstract visitor* to *concrete elements* as it implies that all *visitors* need to be changed, once a *concrete element* is added. The *visitor* is a good choice, if the improved cohesion or maintainability of the functionality realized by the *visitor* outweighs the cohesion of parts of the functionality in the *visitor* with the *elements*.[2] That is: sometimes *feature envy* is rather a "natural odor" than a bad smell of a *visitor*.

Pattern	Data Class	Feature Envy
Flyweight	Extrinsic State	Flyweight
Interpreter	Context	Expression
Mediator		Concrete Mediator
Memento	Memento	Originator
State	Context	Concrete State
Strategy		Concrete Strategy
Visitor	Element	Concrete Visitor

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

There are variations of the same data-behavior separation in other design patterns, as listed in Tab. 1. We separate data into an extra object, if we want other objects to share this data. So does the *extrinsic state* in the *flyweight* pattern and the *context* in the *interpreter* pattern. As a result the classes which use this data (*flyweight* in *flyweight*, *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 change it dynamically as we do by exchanging one *concrete state* 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

[2] The trade-off is a direct consequence of the object-oriented principle that behavior belongs to exactly one object. The trade-off dissolves in languages that have other means to combine data and behavior: With AspectJ for example the cohesive behavior can be described in a "*visitor*"-aspect that introduces behavior to all *elements* by inter-type declarations [10].

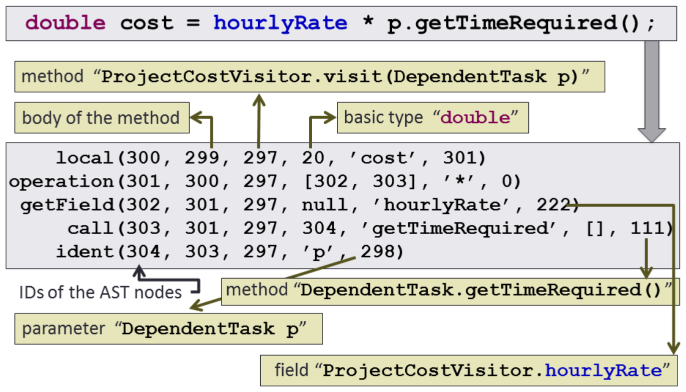
interact with each other without knowing about each other the *concrete mediator* might operate on the data of a few of the *colleagues*.

The separation of data and behavior is not the only possible consequence of design pattern. Some pattern give certain roles (*factory*, *builder*, *singleton*, *façade*, *mediator*) such a central responsibility that they naturally tend to have strong afferent coupling (*shotgun surgery* in the sense of [15]). Roles that are meant to coordinate or configure other objects (*concrete factory*, *builder* and its *director*, *façade*, *concrete mediator*, *concrete visitor*, *context* in *state*) naturally tend to have strong efferent coupling and might even attract high complexity. Types that are inserted into existing hierarchies for the purpose of a pattern (*adapter*, *composite*, *abstract decorator*, *concrete decorator*) tend to extend the interface disproportional or to ignore the inherited features. Finally the level of indirection that is added by some roles (*adapter*, *abstraction* in *bridge*, *abstract decorator*, *façade*, *proxy*, *concrete mediator*, *context* in *state*) would without the pattern be considered as the smell *middle man* defined in [6].

3 Logic Based Code Analysis

The specifications of quality knowledge are defined on a representation of Java source code as Prolog facts (3.1), on which we define graph specifications ("structures", 3.2) as a basis for smell (3.3) and pattern definitions (3.4). Finally these smell definitions can easily be adapted to take developer intentions expressed as patterns into account (3.5).

3.1 Fact Representation of Object-Oriented Code



```
double cost = hourlyRate * p.getTimeRequired();
```

```
    local(300, 299, 297, 20, 'cost', 301)
operation(301, 300, 297, [302, 303], '*', 0)
 getField(302, 301, 297, null, 'hourlyRate', 222)
     call(303, 301, 297, 304, 'getTimeRequired', [], 111)
    ident(304, 303, 297, 'p', 298)
```

method "ProjectCostVisitor.visit(DependentTask p)"
body of the method
basic type "double"
IDs of the AST nodes
method "DependentTask.getTimeRequired()"
parameter "DependentTask p"
field "ProjectCostVisitor.hourlyRate"

Fig. 2: Fact representation of the first line of the method `visit(DependentTask)` in `ProjectCostVisitor` and the referenced nodes of the AST.

Our prototypes are implemented as plug-ins for the development environment Eclipse. They contain a builder that runs everytime the Java builder 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. All language elements of Java 5 are represented. Fig 2 shows the facts representing the first statement in the method `visit(DependentTask)` in `ProjectCostVisitor`. Each fact represents a node of the AST. The first parameter of a fact is a unique ID of this node. The second parameter is a back reference to the parent node. The third parameter references the enclosing method. The remaining parameters contain attributes as well as references to the child nodes. It is very easy to build more abstract predicates based on this fact representation. We will use many predicates without defining them as we hope that the names of the predicates convey their meaning. Here are two example definitions:

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

3.2 Structures

The structure behind smells and metrics and the structure of design pattern can be implemented similarly. The smell *feature envy* analyses how strongly a method operates on fields of another class. This can be seen as the analysis of a graph (illustrated in Fig. 3 (a)) with nodes for the method, the own and foreign attributes and edges for the access relation. The pattern *visitor* can be seen as graph (partially illustrated in Fig. 3 (b)) with nodes for the single *abstract visitor* and the single *abstract element* and a few nodes for the *concrete visitors* and *concrete elements*. As there can be at least a few of these graphs in the program, we want to be able to distinguish them. Therefore we call one node of each graph which belongs to only one of these graphs the *corner* of it. The corner of the graph for *feature envy* would be the method under consideration. The corner of the graph for the *visitor* can be the *abstract visitor*.

Definition 1 (Structure Schema). *A structure schema consists of (1) one unary predicate that tests whether a program element is the* corner *of a structure, (2) some binary predicates that tests whether given the corner of a structure, another program elements plays a certain* role *in the structure with this corner, (3) and some ternary predicates that tests whether given the corner, a* relation *between two other program elements that play a role in this structure exists.*

Definition 2 (Structure Instance). *Given a structure schema and a program element that fulfils the corner predicate, we call this program element the* corner of the structure [instance]*, where the structure consists of (1)* role players*, which are all program elements annotated with the names of the role predicates of the structure schema, for which the role predicate is true, if we use the corner as a first argument and the role player as a second argument; (2)* relation player duos*, which are all ordered pairs of program elements annotated with the name of a relation predicate of the structure schema, for which the relation predicate is true, if we use the corner as a first argument, the first element of the pair as second argument, and the second element of the pair as a third argument.*

3.3 Smell Detection Strategies

Detection strategies as defined by Marinescu and Lanza in [15] are elementary logical formulas built of comparisons of metrics with corresponding thresholds. To illustrate their idea and our implementation we will explain the detection strategy for the smell *feature envy* top-down. The detection strategy reads:

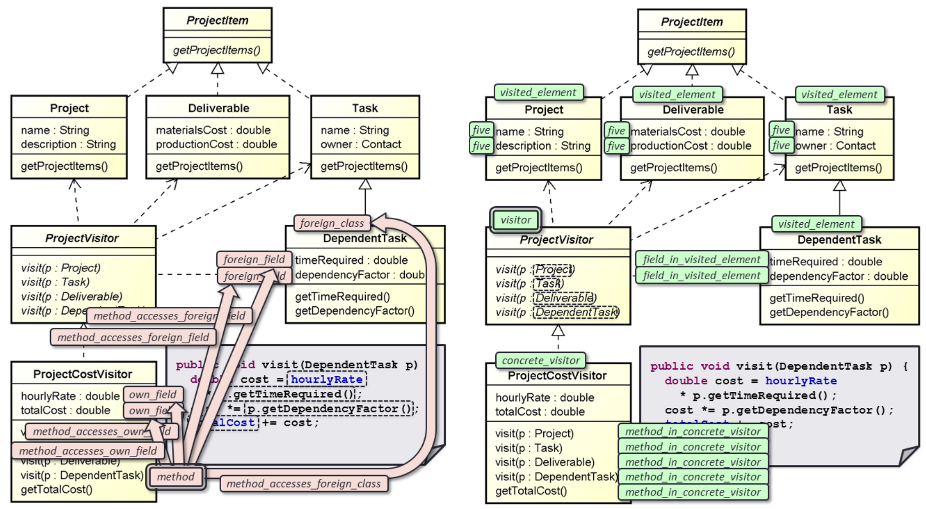
```
feature_envy(M) :-
    feature_usage_structure(M),
    access_to_foreign_data(M, ATFD),      ATFD >  2,
    locality_of_attribute_access(M, LAA), LAA  <  0.3,
    foreign_data_provider(M, FDP),        FDP  =< 5.
```

Here the first goal verifies that M is a method for which we can calculate the *feature envy*. The rest is built 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 relations in the structure, as the source code shows:

```
access_to_foreign_data(M, V) :-
   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(M, V) :-
   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 as illustrated in Fig. 3. The first lines of the definition read like follows:

```
feature_usage_structure(S) :-
    source_method(S), not(abstract(S)).
method(S, M) :-
    feature_usage_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), private(F).
[...]
method_accesses_foreign_field(S, M, F) :-
   method(S, M),
   foreign_field(S, F),
   method_accesses_field(M, F).
[...]
```



```
public void visit(DependentTask p) {
  double cost = hourlyRate
    * p.getTimeRequired();
  cost *= p.getDependencyFactor();
  totalCost += cost;
```

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

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

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(C, V), not(interface(C)).
method_in_concrete_visitor(P, M) :-
    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).
```

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[3] 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(sub_type(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) separately 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 usage structure* and the *data class*:

```
data_class(C) :-
    named_internal_type(C),
    not(natural_odor(data_class, 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)),
   method_accesses_field(M, F).
```

[3] It is reasonable to use the naming conventions used in [7] even if identifiers are not in English. At least, one can expect some consistency in the naming of types and methods if design patterns are used intentionally. This names are meant to be configured by predicates like the ones we give here.

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

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

The smell *data class* will now ignore any class that plays the role of a *concrete element* in the *visitor* pattern. In the calculation of the *feature usage structure* all accesses from a method in a *concrete visitor* to a *field in visited element* will be ignored. That is, feature envy towards other classes will still be detected.[4][5]

4 Conflicts Stimulate Knowledge Evolution

We suggest to use our technology for an alternating "cultivation" process of code reviews (quality improving) and code structure documentation (quality knowledge improving). For the process of quality improvement with respect to generic criteria a detailed guideline can be found in [15]. For the process of improving the explicit quality knowledge, we gave a first suggestion here. Currently we expect unjustified smell warnings to be the strongest stimulus to increase the design knowledge.[6] In addition pattern mining tools can be used to identify pattern instances [13]. Making the design explicit allows to ignore the smells on certain elements, to use specific thresholds or to adapt the metrics to it. It provides the additional benefit of a defined structure that can be verified.

The ad-hoc solution to avoid unjustified warnings by manually excluding them e.g. with an annotation[7] is not desirable: Many locations would need to be annotated. The trade-off that led to the decision is still implicit. Every time the code changes, the reasons for ignoring the smells would need to be revisited. Such an annotation could be seen as a dependency of the general and rather stable

[4] 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.

[5] 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 obviously considerable option.

[6] This guarantees that the design knowledge is relevant to the developer and not only specified because of an ideal of completeness. Exman discussed for the area of software reuse from Web sources ("Websourcing") in [5] that knowledge is necessary incomplete. The incompleteness is even desirable as completeness is computational expensive and the ideal of completeness might lead to inconsistencies, especially as the specified knowledge has to be interpreted. He demonstrates that relevance can be a good replacement for completeness.

[7] Similar to the Java 5 annotation `@SuppressWarnings` to suppress compiler warnings

definition of the smell to the concrete and relatively unstable code. Dependencies towards unstable concrete code increase the burden of maintenance.

In the longer run, the 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*. Such an constraint can be added as predicate to the pattern schema. Another expectation is, that the *element* classes can stay much more stable than the *visitors* do. This expectation could be added to the structure schema as an absolute or relative upper bound on the change frequency of the artifact in the repository. The development environment could regularly check the change frequencies such providing an third feedback cycle.

5 Evaluation

5.1 Smell in Pattern

Sebastian Jancke implemented the detection strategies defined in [15] as well as a few more as part of his diploma thesis [11]. He analyzed different open source projects and conducted a successful user study. Tab. 2 lists some of the instances of natural odors he found.

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]

Table 2: Natural odors found in open source software projects

Although we assume that the idea to adapt smell definitions based on knowledge about design intentions is convincing, we are planning larger code studies to explore the relevance of our observations further. The following case study focuses on the aspect of our approach that we consider to be least convincing: To identify structure instances we suggest to refer to design knowledge that the developer has made explicit in names of methods, classes or packages or with annotations. As we elaborate in [22] the meaning of some names can be seen as the structure in which they are used.

5.2 Taking advantage of expressed intentions: Creation Methods

In [15, Ch. 6] "Collaboration Disharmonies" Lanza and Marinescu reference design knowledge in a way, that is unique in the book. The two strategies *intensive coupling* (many dependencies to a few other classes) and *dispersed coupling* (dependencies to many other classes) contain besides conditions about the coupling an additional condition "Method has few nested conditionals" measured by MAXNESTING > 1, where MAXNESTING is the maximal nesting depth of blocks in the method. 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 [. . .]." [15, 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. We wanted to test this statement with a little case study using the current version of the same source code that was used in [15][8]. To discuss this statement we call methods with MAXNESTING = 1 *flat* and present three hypotheses:

$$\text{Flat methods are configuration methods.} \tag{1}$$

$$\text{Coupling in flat methods is no design problem.} \tag{2}$$

$$\text{Coupling in configuration methods is no design problem.} \tag{3}$$

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 instead. Exploring twenty randomly and a few systematically chosen methods we found that in ArgoUML methods with names starting with "init", "create", "build" or "make" are configuration methods and methods with names starting with "test" or containing the term "Test" in their name or in the name of the enclosing class are test methods. 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.

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 ($1090/1242 = 88\%$) 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. 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

[8] `http://argouml.tigris.org/`, accessed in 06/2011. The 13 projects listed in the file "argouml-core-projectset.psf" were used and all 13297 non abstract methods of the 2083 named classes were analyzed. [15] used the version from 10/2004. An spreadsheet with our data can be downloaded from our website `http://sewiki.iai.uni-bonn.de/private/daniel/public/ic3k2011`

	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

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*.

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 it 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.[9]

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[10]. 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. Here is how we would implement the corresponding adaptation:

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

[9] Even if the responsibility for configuration is "extracted" into XML configurations.
[10] Indeed half of them turned out to be a sort of configuration method again. So restricted (!) to the methods with smells the nesting condition is a reasonable heuristic.

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

6 Related Work

Wuyts coined the term "logic meta-programming" in his Ph.D. thesis [23]. He presented the Smalltalk Open Unification Language ("SOUL"), a programming language for logic meta-programming on Smalltalk programs. His thesis introduced the term "co-evolution" for the consistent evolution of program elements, supported by specified consistency conditions. We suggest to consider the evolution of the meta-programs as well, as shortcomings in the meta-programs may be only observed over time.

Hajiyev, Verbaere, and de Moor presented in [8] the source code querying tool CodeQuest. This system uses Datalog queries that are guaranteed to terminate. These queries are translated into efficient SQL queries. As far as we presented our queries they are very close to Datalog queries and our suggestions could directly be used with CodeQuest. In particular we did not have any negation in a recursive cycle so that our code is stratified. The reasons why we currently prefer Prolog over Datalog - like the meta-circularity of Prolog - are beyond the scope of the paper. Kniesel, Hannemann, and Rho [14] showed that that the performance of this and our platform is comparable.

As the structures we define can be seen as typed graphs, approaches analysing the code as graph structures are as well appropriate implementation means. An example would be the TGraph approach by Ebert, Riediger, Winter, and Bildhauer [4] with its mature query language GReQL [3].

Riehle suggested to model design pattern as systems of roles and relations in Ph.D. thesis [20]. As a case study he documented all the design pattern in JHotDraw 5.1.[11] The role modeling approach could be seen as the current default approach for design pattern modeling and is used in [13] as well.

Brichau, Kellens, Castro, and D'Hondt describe in [1] IntensiVE a tool suite build on SOUL. They demonstrate that structural regularities like design pattern as well as smell definitions can easily be defined as queries (here called "intensions"). Again, we would expect it to be straightforward to implement our approach with this tool.

Moha, Guéhéneuc, Le Meur, and Duchien present in [18] the Software Architectural Defects Specification Language ("SADSL"). They analyse the design defects "Swiss army knife", "Functional decomposition", "Blob", and "Spaghetti code" and demonstrate that they can specified with SASDL. Detection algorithms are generated from this specification.

[11] Versions of JHotDraw from the last ten years can be found online starting from `http://www.jhotdraw.org/`. As this code was implemented to demonstrate the use of design pattern and is considered as well designed it is very often used in reverse engineering case studies.

7 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 "cultivation" process of code review (quality improving) and code documenting (quality knowledge improving). A case study showed that the precision of smell detection can be increased if the design knowledge that developers already made explicit is utilized instead of guessing developer intentions from structural properties.

Acknowledgments 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. 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 for code quality evaluation and coding culture evolution. Tobias Rho, Dr. Günter 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 detailed discussions on earlier versions of this work and for supporting it.

References

1. Johan Brichau, Andy Kellens, Sergio Castro, and Theo D'Hondt. Enforcing Structural Regularities in Software using IntensiVE. *Sci. Comput. Program.*, 75(4):232–246, 2010. 347
2. Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns.* Morgan Kaufmann, July 2002. 335
3. Jürgen Ebert and Daniel Bildhauer. Reverse Engineering Using Graph Queries. *Graph transformations and model-driven engineering*, pages 335–362, 2010. 347
4. Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph Technology in Reverse Engineering. The TGraph Approach. In *Proc. 10th Workshop Software Reengineering. GI Lecture Notes in Informatics*, 2008. 347
5. Iaakov Exman. Knowledge Incompleteness Considered Desirable. In Anabel Fraga and Juan Llorens, editors, *Proc. of 2nd Workshop on Knowledge Reuse (KREUSE 2009) hosted by ICSR 2009*, Falls Church, VA, USA, 2009. 343
6. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, June 1999. 335, 338

7. Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, January 1995. 335, 342
8. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *CodeQuest:* Scalable Source Code Queries with Datalog. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006. 347
9. Martin E. Nordberg III. Default and Extrinsic Visitor. In *Pattern languages of program design 3*, pages 105–123. Addison-Wesley Longman Publishing Co., Inc., 1997. 337
10. Martin E. Nordberg III. Aspect-Oriented Dependency Management. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 557–584, Boston, 2005. Addison-Wesley. 337
11. Sebastian Jancke. Smell Detection in Context. Diploma Thesis, University of Bonn, March 2010. 344
12. Joshua Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, August 2004. 335
13. Günter Kniesel and Alexander Binun. Standing on the Shoulders of Giants - A Data Fusion Approach to Design Pattern Detection. In *17th International Conference on Program Comprehension (ICPC 2009)*, Vancouver, Canada, 2009. 343, 347
14. Günter Kniesel, Jan Hannemann, and Tobias Rho. A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution (LATE 2007)*, New York, NY, USA, 2007. 347
15. Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice.* Springer-Verlag, August 2006. 335, 338, 340, 343, 344, 345
16. Radu Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, Washington, DC, USA, 2004. 335
17. Robert C. Martin. Acyclic Visitor. In *Pattern languages of program design 3*, pages 93–103. Addison-Wesley Longman Publishing Co., Inc., 1997. 337
18. Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien. A Domain Analysis to Specify Design Defects and Generate Detection Algorithms. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2008. 347
19. Jens Palsberg and C. Barry Jay. The Essence of the Visitor Pattern. In *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 9–15. IEEE, 1998. 337
20. Dirk Riehle. Framwork Design, A Role Modeling Approach. Dissertation, ETH Zürich, 2000. 347
21. Arthur J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996. 335
22. Daniel Speicher, Jan Nonnen, and Holger Mügge. How many realities fit into a program? - Notes on the meaning of meaning for programs. In *Proceedings of the Second International Workshop on Software Knowledge (SKY 2011)*, Paris, France, 2011. 344
23. Roel Wuyts. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. PhD thesis, Vrije Universiteit Brussel, 2001. 347