Multi-criteria Detection of Bad Smells in Code with UTA Method

Bartosz Walter[1] and Błażej Pietrzak[1,2]

[1] Institute of Computing Science, Poznań University of Technology, Poland
{Bartosz.Walter,Blazej.Pietrzak}@cs.put.poznan.pl
[2] Poznań Supercomputing and Networking Center, Poland

Abstract. Bad smells are indicators of inappropriate code design and implementation. They suggest a need for refactoring, i.e. restructuring the program towards better readability, understandability and eligibility for changes. Smells are defined only in terms of general, subjective criteria, which makes them difficult for automatic identification. Existing approaches to smell detection base mainly on human intuition, usually supported by code metrics. Unfortunately, these models do not comprise the full spectrum of possible smell symptoms and still are uncertain. In the paper we propose a multi-criteria approach for detecting smells adopted from UTA method. It learns from programmer's preferences, and then combines the signals coming from different sensors in the code and computes their utility functions. The final result reflects the intensity of an examined smell, which allows the programmer to make a ranking of most onerous odors.

1 Introduction

Software, both while development and maintenance, require health-preserving activities. In Extreme Programming (XP) refactoring is the key practice addressing this issue [1]. The term denotes constant restructuring the program code in order to make it more readable and understandable while preserving its behavior [2].

Applying the treatment is just next to diagnosing the illness. Prior to performing an action, a programmer should find the suspected code and identify the problem to apply an appropriate solution. XP coined the term *code smells* to describe the common structures and constructs in the code that possibly require refactoring. There are over 20 different smells known, but their detection is still based on human intuition and experience. The lack of formal criteria for detecting smells effectively prevents it from being fully automated, which seriously affects the refactoring performance. Existing approaches to smells detection are based on a semi-automated analysis of metrics values, with a programmer making the final assessment and deciding whether the flawed code should be refactored or not.

Unfortunately, metrics fail to uncover many smell symptoms. They just deliver an aggregate measure of the code, whereas smells often reveal also by improper statements, dangerous program behavior or presence of other smells. Besides, metrics represent individual code properties, like cohesion or class size, while ignoring other aspects. Thus, there is a need for a detector that would simultaneously attain two goals: combine different signals indicating the smell presence and let the programmer to adjust the detection process to individual preferences.

In the paper we present a multi-criteria approach to smell detection. Data coming from six sources we identified are quantified and aggregated into a single value using a multi-criteria method UTA [3]. The method learns from the programmer's subjective preferences of smells intensity, and then creates a ranking of detected odors, which reflects the preferences. The stored preference model can be reused later.

The paper is structured as follows: In chapter 2 we present the identified sources of smell symptoms. Chapter 3 gives a short description of the UTA method with an example of defining utility functions for every data source for a *Large Class* smell. In chapter 4 we present the evaluation on selected classes taken from Jakarta Tomcat project [13], and conclude with a summary in chapter 5.

2 Data Sources for Smell Detection

In the real world, smells are detected with a single sense only: a nose. In programming, however, code smells usually are a combination of different subtle odors coming from various sources. Some of them, like publicly exposed attributes of a class, can be effectively localized with only one sensor, while others – the vast majority – require smarter detection. This comes from the high-level nature of smells: each of them actually discloses several symptoms. That yields several data sources that indicate smell presence.

As an example, consider the *Data Class* smell [2]. It deals with breaking the encapsulation and limited or no responsibility of a class. It is characterized by three violations that describe not-so-closely related programming faults:

- A class with no functionality (methods) but holding data,
- A public attribute of a class,
- A collection returned by class methods can be modified independently from the owning object.

Each of these violations has different symptoms and requires appropriate detectors: the number of methods in a class can be found with metrics, a public field – with analysis of a syntax tree, and an insecure collection – by running the code.

In general, we identified six distinct sources of data useful for smell detection:

- programmer's intuition and experience,
- metrics values,
- analysis of a source code syntax tree,
- history of changes made in code,
- dynamic behavior of code,
- existence of other smells.

2.1 Intuition and Experience

At present stage the human factor in refactoring is unavoidable in smell detection. While applying refactorings is subject to partial automation, the identification of smells is a creative activity that cannot be fully formalized. According to Fowler "no set of metrics rivals informed human intuition" and the objective is to "give indications that there is trouble that can be solved by a refactoring" [2].

Attempts toward supporting smells detection [4, 5] are based on pointing at most distinct cases, but finally leaving the decision whether to refactor or not to the programmer. Smells are the matter of subjective aesthetics.

2.2 Metrics

Measuring the software code is a common method of taking quick, compact picture of its structure. It is also prevalently and primarily used for detecting smells (e.g. [4]). Most smells affect multiple code measures, and their impact is relatively easy to detect. The term 'metrics driven development', introduced by Simon et al. [5], describes a process, in which the programmer makes choice of code transformations to be done depending on the measurement outcome. Pieces of code with the most deviated metrics are refactored first.

Code metrics, while useful in many cases, cannot sense several smells. It should be noticed, however, that metrics produce mere numbers, not actual suggestions for refactoring. They cannot perform dynamic analysis or look for incorrect constructs. Effectively, doubtless following the metrics may lead to wrong decisions, so they need support from other sensors.

2.3 Syntax Trees

Analysis of abstract syntax trees (AST) is another frequently used source of smells indicators. It allows for finding inappropriate or deprecated statements, improper data types etc. In particular, it provides context information, like inherited members or access modifiers, that otherwise missing could affect the outcome from other sensors.

This method is often combined with metrics, because they are usually computed from the syntax trees. In this case the AST analysis plays an auxiliary role to metrics.

2.4 Code Behavior

There are a few smells that actually result from design flaws, but are difficult to find with a static analysis only. The smells are subject to dynamic analysis, requiring running the code. We propose to utilize unit tests [6] for that purpose. Unit test take a single code unit (usually a class) and examine if its methods react appropriately to the input values. For the sake of smell detection, unit tests simulate conditions in which a smell can be sensed. For example, to detect if a collection is well protected, they attempt to execute all its mutating methods, and check whether the collection actually changes [7]. The tests results indicate the smell presence.

Since the expected outcome is known in advance, the tests could be generated in semi-automated way from templates defined for the given smell (similar approach was applied in for verification of refactorings correctness [8]). In this case the cost of creating the test is significantly reduced. The prototype of such detector we built for *Data Class* smell [7] proved the applicability of the concept.

2.5 Code History

Some smells, like *Shotgun Surgery* or *Divergent Change*, are called maintenance smells [4]. They reveal the design flaws by analyzing how the code reacts to changes. The changes cannot be detected with analysis of a single piece of code, so they compare the subsequent code versions [9]. Configuration management system's entries make a good example for that.

2.6 Presence of Other Smells

Most smells co-exist with others, which means that presence of one smell may suggest presence of related ones. This is due the common origins of some smells: a single code blunder gives rise to many design faults, and the resulting smells are not independent of each other. For example, *Long Methods* often contain *Switch Statements* that decide on the control flow, or have *Long Parameter Lists*. The latter case indicates also the *Feature Envy*, since most of data must be delivered from outside. Therefore, if a method is considered long, the danger of *Long Parameter List* gets higher. However, there is a need for statistical analysis that would confirm or reject this hypothesis.

3 Multi-criteria Model of Smells

Multiple smell sensors give different, partial views of a single odor. To obtain a complex image of the smell, there is a need for a detector that both combines the heterogeneous signals and still preserves the initial programmer preferences concerning their impact on the resulting grade. This chapter illustrates the multi-criteria model of smells based on an example of detecting the *Large Class* [2] smell.

3.1 UTA Method

In general, there are two approaches to create the intensity function: traditional *aggregation* approach and *disaggregation-aggregation* approach [3]. In the traditional aggregation approach the utility function (intensity) is created first *a priori* and then it is evaluated.

In the latter approach the algorithm is split into two phases. The disaggregation phase aims at reconstructing a preference model from the decision maker's limited set of reference alternatives. Next, the aggregation phase, basing on the information induced from the disaggregation one, concludes to construct the measurable utility functions reflecting the verbal criteria used initially.

UTA [3] is a method for ranking a finite set of alternatives, evaluated by the finite set of criteria. The comprehensive preference model used by the method is an additive utility function. The decision maker creates a subjective ranking of small subset of reference alternatives. In UTA, the reference ranking involves two relations: strict preference and indifference. The main goal of the method is the construction of additive utility function compatible with the reference ranking. The construction proceeds according to ordinal regression approach. The utility function is then used on the entire set of alternatives, giving a ranking value to each alternative. The Kendall's coefficient describes the correlation between the reference ranking and the utility function. Values greater than 0.75 indicate that the utility function is compatible with reference ranking obtained from the decision maker. The maximum value for Kendall's coefficient is 1.0.

This method is highly interactive and permits the decision-maker to select the best solution according to his/her subjective point of view. It is also conducive to changes in product preferences and generally copes well with noisy or inconsistent data [10]

UTA is also a good solution in cases where there exist difficulties in obtaining values of the preference model directly from the user.

3.2 Multi-criteria Model for Detecting *Large Class*

According to Fowler, a *Large Class* is a class that is trying to do too much [2]. To be more specific, we decided that following symptoms indicate existence of a *Large Class* smell:

- The class has too much functionality,
- A *Temporary Field* smell [2] occurs in the class,
- *Data Clumps* or *Long Parameter List* smells [2] occur in the class,
- *Inappropriate Intimacy* [2] occurs in the class.

3.2.1 Measuring Class Functionality

The metrics used for measuring functionality offered by a class are presented in Table 1. The recommended threshold values for these metrics are taken from the Nasa Software Assurance Technology Center [11]. We assumed that a class has too much functionality when one of the metrics exceeds the accepted maximum.

Table 1. Metrics used for measuring functionality (source: [11, 12])

Metric	Description	Maximum accepted value
Number Of Methods (NOM)	Number of methods in the class.	20
Weighted Methods per Class (WMC)	Sum of cyclomatic complexities of class methods.	100
Response For Class (RFC)	Number of methods + number of methods called by each of these methods (each method counted once).	100
Coupling Between Objects (CBO)	Number of classes referencing the given class.	5

3.2.2 Detecting *Temporary Fields*

A field is considered temporary when it is set only in certain circumstances [2]. We decided not to detect this smell, because the detector gives many confusing results: e.g. setting methods in a *Data Class* [2] by design use only one field at a time. Such field could be falsely considered as temporary by the detector.

3.2.3 Detecting *Inappropriate Intimacy*

Over-intimate classes are classes that spend too much time delving in each other's private parts [2]. We assume that *Inappropriate Intimacy* is present when:

- There are bi-directional associations between classes, or
- Subclasses know more about their parents than they should.

To detect this smell the detector traverses the Abstract Syntax Tree and counts the number of bi-directional associations for every class (the NOB metric).

3.2.4 Finding *Data Clumps* and *Long Parameter Lists*

The *Data Clumps* smell [2] relates to a group of fields that appear in several classes in the same context. *Long Parameter List* smell [2] is a special case of *Data Clumps*, but dealing with methods: it is a group of method parameters that appears in several methods.

Since this smell requires knowledge of the code semantics (which is unavailable at the code level), we do not detect it.

4 Experimental Evaluation

The proposed approach to detecting *Large Class* [2] smell was evaluated on Jakarta Tomcat 5.5.4 [13] project. Tomcat is well known to the open-source community for its good code quality resulting from constant refactoring.

In order to model the programmer preferences, we needed a learning ranking. It can be either constructed with a real variants or imaginary data, if it only reflects the programmer sense of smell. We chose the latter option, since the criteria for detecting *Large Class* smell had been already defined in Chapter 3. In such cases the learning set is included into the variants set, and then removed from final ranking.

The detection rules are presented in Table 2. Relation *S* denotes the strict preference relation and *I* stands for the indifference relation. We prefer excessive functionality to *Inappropriate Intimacy* smell, because the latter one not necessarily implies the *Large Class*. We also distinguish different aspects of over-functionality. Any metric describing functionality that is exceeding the accepted value is considered to be smelly, but classes with numerous methods stink more.

Table 2. Smell symptoms reference ranking for *Large Class*

Relation	Variant name	NOM	RFC	WMC	CBO	NOB
	Imaginary_NOM	21	21	21	0	0
S						
	Imaginary _CBO	1	1	1	6	0
I						
	Imaginary _RFC	1	101	1	1	0
I						
	Imaginary _WMC	1	1	101	0	0
S						
	Imaginary _NOB	1	2	1	1	1

Table 3 presents top ten smelly classes in the ranking generated by UTA method after analyzing 829 classes of Tomcat code base. The *U* column represents the aggregate utility value.

However, the resulting ranking is not satisfactory. It reflects the weak order of the learning ranking, but there are several doubtful cases. For example, analyzing the final ranking we noticed that the *Request* class smells more than *Digester* class. These classes have similar values on every criterion except for the NOB metric. The *Digester* class have over three times higher value on that criterion and is considered to be less smelly. We looked through the code to assess which class actually smelled more intensively and decided that the *Digester* class was a better candidate for refactoring. The resulting value of utility function for *Request* class was higher

Table 3. Smell intensity ranking

U	Class name (org.apache.*)	NOM	RFC	WMC	CBO	NOB
1.00	*catalina.core.StandardContext*	250	967	353	103	9
0.97	*catalina.connector.Request*	129	426	161	69	4
0.92	*tomcat.util.digester.Digester*	129	387	110	57	13
0.88	*coyote.tomcat4.CoyoteRequest*	113	333	111	62	2
0.82	*jasper.compiler.Parser*	57	407	272	60	1
0.81	*catalina.core.StandardServer*	58	379	224	75	2
0.80	*catalina.core.StandardWrapper*	79	337	109	71	4
0.78	*catalina.servlets.WebdavServlet*	29	441	302	57	0
0.77	*catalina.servlets.DefaultServlet*	36	402	202	62	0
0.76	*catalina.connector.Response*	72	280	109	52	3

than for *Digester* class due to compensation on other criteria. The other interesting aspect of intensity ranking is that the *Parser* class is preferred to the *StandardWrapper* class, although *Parser* class has many complex methods that could be extracted. This would result in higher number of methods (the NOM metric).

Luckily, these drawbacks can be easily fixed. The advantage of UTA method is that it can incrementally refine the preference model until the result is satisfactory. Of course, the modifications can affect the correspondence between learning and final ranking. We decided to extend the reference ranking by adding four relations: *Digester* is preferred to *Request, Request* is preferred to *Parser*, *Parser* is preferred to *StandardWrapper* and the latter one is preferred to the artificial *Imaginary_NOM* variant.

Table 4. Smell intensity ranking for modified reference ranking

U	Class name (org.apache.*)	NOM	RFC	WMC	CBO	NOB
0.96	*catalina.core.StandardContext*	250	967	353	103	9
0.84	*tomcat.util.digester.Digester*	129	387	110	57	13
0.82	*catalina.connector.Request*	129	426	161	69	4
0.74	*coyote.tomcat4.CoyoteRequest*	113	333	111	62	2
0.66	*jasper.compiler.Parser*	57	407	272	60	1
0.63	*catalina.core.StandardWrapper*	79	337	109	71	4
0.62	*catalina.core.StandardServer*	58	379	224	75	2
0.60	*catalina.loader.WebappClassLoader*	50	301	256	63	0
0.60	*catalina.connector.Response*	72	280	109	52	3
0.59	*catalina.servlets.WebdavServlet*	29	441	302	57	0

The intensity ranking for modified references (Table 4) reflects our expectations. The *Inappropriate Intimacy* smell is finally taken into consideration, *Digester* class is preferred to *Request* class and *StandardWrapper* is class preferred to *Parser* class. The ranking is still compatible with the reference ranking at acceptable level (the Kendall's coefficient is equal 0.98).

5 Conclusions

Detecting bad smells in the code is a complex problem. The high-level and vague description given by Fowler makes it difficult to define strict and clear rules of what is and what is not a smell. Programmers can merely observe and measure smell symptoms, which are ambiguous and may lead to wrong conclusions. Effectively it is the programmer who makes the decision basing on intuition and experience.

The multi-criteria model of smells that we proposed, based on the UTA method, responds to the problems mentioned above. It aggregates multiple sources of smell symptoms, not only the metrics, which broadens the spectrum of potential sensors. We determined six such sources, and most of them can be objectively examined and measured.

Every smell is defined by a set of measurable symptoms that suggest its presence. Our model utilizes UTA method to combine these measures and concludes with a ranking of smell intensity. It is important that the method learns from the programmer preferences and constructs the ranking according to them.

The experiment with detecting *Large Class* smell showed that multi-criteria approach allows for more thorough representation of programmer preferences than individual metrics. A combination of various symptoms provides more versatile insight into the nature of a smell. The proposed approach is highly interactive and permits the programmer to define smells according to his/her subjective point of view.

Acknowledgements

This work has been supported by the Polish State Committee for Scientific Research as a part of the research grant KBN/91-0824.

References

1. Beck K.: Extreme Programming Explained. Embrace Change. Addison-Wesley, 2000.
2. Fowler M.: Refactoring. Improving Design of Existing Code. Addison-Wesley, 1999.
3. Jacquet-Lagreze E., Siskos J.: Assessing a Set of Additive Utility Functions for Multicriteria Decision-making, the UTA Method. European Journal on Operational Research, Vol. 10, No. 2, 1982, 151-164.
4. van Emden E., Moonen L.: Java Quality Assurance by Detecting Code Smells. In: Proceedings of the 9th Working Conference on Reverse Engineering. IEEE Computer Press, 2003.
5. Simon F., Steinbrueckner F., Lewerentz C.: Metrics Based Refactoring. In: Proceedings of CSMR Conference. Lisbon, 2001.
6. JUnit, http://www.junit.org, January 2005.
7. Pietrzak B., Walter B.: Automated Detection of Data Class smell. Inżynieria Oprogramowania. Nowe wyzwania. WNT, 2004, 465-477 (in Polish).
8. Walter B., Pietrzak B.: Automated Generation of Unit Tests for Refactoring. Lecture Notes in Computer Science, Vol. 3092. Springer Verlag, Berlin Heidelberg New York, 2004, 211–214.
9. Ratju D., Ducasse S., Gybra T., Marinescu R.: Using History Information to Improve Design Flaws Detection. In: Proceedings of 8th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, Washington D.C., 2004, 223-232.
10. Beuthe M., Scannella G.: Comparative Analysis of UTA Multicriteria Methods. European Journal of Operational Research, Vol. 130, No. 2, 2001, 246-262.
11. NASA Software Assurance Technology Center: SATC Historical Metrics Database, http://satc.gsfc.nasa.gov/metrics/codemetrics/oo/java/index.html, January 2005.
12. Chidamber S.R., Kemerer C.F.: A Metrics Suite from Object-Oriented Design. IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994, 476-493.
13. The Apache Jakarta Project: Tomcat 5.5.4, http://jakarta.apache.org/tomcat/index.html, January 2005.
14. Pietrzak B.: XSmells: Computer Aided Refactoring of Software. M. Sc Thesis, Poznań University of Technology. Poznań, Poland, 2003.