Leveraging Code Smell Detection with Inter-smell Relations

Błażej Pietrzak and Bartosz Walter

Institute of Computing Science, Poznań University of Technology, Poland
{Blazej.Pietrzak, Bartosz.Walter}@cs.put.poznan.pl

Abstract. The variety of code smells deserves a numerous set of detectors capable of sensing them. There exist several sources of data that may be examined: code metrics, existence of particular elements in an abstract syntax tree, specific code behavior or subsequent changes in the code. Another factor that can be used for this purpose is the knowledge of other, already detected or rejected smells. In the paper we define and analyze different relations that exist among smells and provide tips how they could be exploited to alleviate detection of other smells.

Keywords: Refactoring, bad code smells, inter-smell relations.

1 Introduction

The quality of source code is one of the factors affecting the software maintenance cost [1]. Poor quality results both in short term in increased fault ratio and on the long run in higher expenditure on modifications and further development of the product. Code quality is then a costly, although valued attribute of software, which gives a chance for savings and profits in further software maintenance, but requires considerable initial investments.

High quality source code is particularly important in agile methodologies. eXtreme Programming (XP) [2], the most popular among them, diminishes the importance of documentation in favour to the source code readability and comprehension. Any factors that do not contribute to these values are considered potential threats and are candidates for improvement. Although there exist numerous different source code flaws that can negatively affect the software quality, XP covers all of them by a vague term of *bad code smell* [2]. Smells are defined as constructs in the code that "suggest (sometimes scream for) the possibility of refactoring" [3]. This deliberate imprecision, which puts stress on the human judgment based on experience and the sense of aesthetics, leads to significant problems with automated detection and identification of smells. It is illustrated by the diversity of over 20 bad smells identified by Fowler, which differ in importance, complexity and localization. The range of code elements affected by them spans from entire modules or class hierarchies (*Parallel Inheritance Hierarchies*, *Message Chain*), through single classes and objects (*Feature Envy, Divergent Change*, *Large Class*), then methods (*Extract Method*, *Long Parameter List*), ending up with individual variables, statements and expressions (*Primitive Obsession*, *Temporary Field*). As a result, there exists no

P. Abrahamsson, M. Marchesi, and G. Succi (Eds.): XP 2006, LNCS 4044, pp. 75 – 84, 2006.

general method of smell detection. Each smell describes a distinct flaw, related to either improper structure, communication between objects, low readability and other aspects. In turn, each smell is revealed with multiple symptoms of various nature and require a unique mechanism of identification.

In attempt to capture the subtle, complex nature of smells, in [4] we proposed a multi-criteria, holistic model of smell detection, which combines various sources of information. We identified six such sources considered 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.

Apart from the programmer's intuition, another four data sources are measurable or at least intuitively comprehensible. The last one is special as it reuses information about the already discovered smells, so that they can be exploited again in further examination. It comes from the observation that smells are not independent, separated phenomena and their presence or absence often carries knowledge about other smells. Therefore, it is possible to support code smells detection process with already available information about the relations existing between smells. Our initial thoughts on the smell dependencies have been presented in [11].

In this paper we continue the research and examine some relations existing among code smells, presenting how they could be exploited for more effective smell detection.

The paper is structured as follows. Section 2 describes seven identified relations among bad code smells. It also suggests how the relations could be exploited in smell detection. In section 3 we attempt to evaluate the relevance of the relations on selected classes taken from Jakarta Tomcat project [5]. The paper is concluded with a summary presented in the section 4.

2 Inter-smell Dependencies

Even a superficial analysis of Fowler's bad smells descriptions reveals that most of them are related to each other: some appear in groups, while others exclude one another. In general, the already confirmed presence or absence of a particular smell may carry information about others. It is Fowler who noticed the existence of relations and dependencies between smells: “When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind” [3].

The nature of the relations varies: some smells share a common flaw as an origin, whereas others are revealed by similar symptoms or can be eliminated with a single transformation. The kind of relationship suggests also the way it could be exploited. We focus on the relations that (1) contribute to identification of other smells and (2) their elimination.

In [11] we proposed five coarse relations that describe dependencies between smells. The extended and updated list now contains six relations:

- Plain support,
- Mutual support,
- Rejection.
- Aggregate support,
- Transitive support,
- Inclusion.

In order to measure the effectiveness of the relations we need a metric reflecting their strength. Strength of the plain support relation, which also makes a basis for the other ones, can be measured with the certainty factor [12]. Certainty factor for the relation $r(A, B)$ is interpreted as a number of objects incriminated with the smell B in the set of objects featuring the smell A. The notion of the factor is used in the remaining relations respectively.

2.1 Plain Support

Plain support relation is the simplest relation that may be identified. A smell B is supported by A if the existence of A implies with sufficiently high certainty the existence of B. B is then a companion smell of A, and the program entities (classes, methods, expressions etc.) burdened with A also suffer from B. The relation makes a basis for many other relations analyzed below.

The importance of the relation comes from observation that in A is often an easy to detect smell with few symptoms, while B is a more complex one, embracing various aspects and showing up with different symptoms. Thus, A can be utilized for diagnosing B without delving into its complex nature.

As an example, let us consider the relation between *Data Class* and *Feature Envy*. A *Data Class* is a class inappropriately used as a data container [3], which may evince through one of the following:

- Class contains public fields,
- Class improperly encapsulates a collection,
- Class is structure equivalent and features with only getting and settings methods.

We only analyze the structure equivalent violations, because the other are not related to the *Feature Envy* smell. The exemplary structure equivalent symptom, taken from Tomcat's code base (*org.apache.catalina.deploy.FilterMap* class), is provided below.

```
public class FilterMap implements Serializable {
    ...
    private String filterName = null;
    public String getFilterName() {
        return (this.filterName);
    }
    public void setFilterName(String filterName) {
        this.filterName = filterName;
    }
    private String servletName = null;
    public String getServletName() {
```

```
        return (this.servletName);
    }
    public void setServletName(String servletName) {
        this.servletName = servletName;
    }
    ...
}
```

A method that is more interested in a class other than the one it actually belongs to, is an example of a *Feature Envy* smell [3]. It indicates that the responsibility is improperly distributed among classes. *Feature Envious* methods should be moved to the class that they reference the most. The exemplary *Feature Envious* method taken from Tomcat's *org.apache.catalina.core.ApplicationFilterFactory* class is presented below.

```
public final class ApplicationFilterFactory {
    ...
    private boolean matchFiltersServlet(
        FilterMap filterMap, String servletName) {
        if (servletName == null) {
            return false;
        } else {
            if (servletName.equals(
                filterMap.getServletName())){
                return true;
            } else {
                return false;
            }
        }
    }
    ...
}
```

The *matchFilterServlet()* method checks if the actual servlet name matches the filter's servlet name. It makes no use of any of its enclosing class' fields and methods. There are two objects referenced by it: *filterMap* and *servletName*, each of them referenced twice. Since *servletName* is of a standard type *java.lang.String* and cannot be modified, then *filterMap* object is considered the possible owner of the method. Thus, the method could be moved to the *FilterMap* class, which is a *Data Class*. As a side effect, the latter smell would be removed as well.

Of course, there exist several design patterns, like *Strategy* and *Visitor* [8], which are used primarily to combat the *Divergent Change* smell [3], that violate this rule. In this article we did not take these cases under consideration.

The conclusion is that the structure equivalent version of the *Data Class* smell is closely related to the *Feature Envy* smell. If there exist a *Data Class*, there is usually also another class that uses its data. The client almost certainly contains methods that are *Feature Envy* candidates.

2.2 Mutual Support

This relation is a symmetric closure of the plain support: both related smells support each other. It is not only simply equivalent to two plain support relations, but also suggests that the related smells share common roots and originate from the same code flaw. Removing the reason may result in reduction or even removal of both smells.

Seemingly, it gives a powerful ability to attain two goals with a single action. However, among the smells identified by Fowler there are no two odors mutually supporting each other with considerable certainty. That observation is justified, as different smells, although often related to each other, describe at least slightly, yet different anomalies. Therefore, even if a smell *A* supports smell *B*, the reversed relation (if exists) is weaker. Should any such smell be defined in future, it would resemble the existing ones so much, that the gain from removing it along with others would be negligible.

Unfortunately, we cannot provide any examples of the mutual support relation.

2.3 Rejection

Rejection yields the negative information about smells presence: a smell *B* is rejected by a smell *A*, if the presence of *A* excludes the existence of the smell *B*. Knowing that, we may restrict the exploration area to remaining smells and limit the computational complexity of the detection process.

Noticeably, this relation, unlike others, is symmetric: if *A* rejects *B*, then *B* rejects *A*. Presence or confirmed absence of any of smells participating in the relation carries information about the other one.

For example, a *Lazy Class*, which has no or only limited functionality, cannot be simultaneously an over-functional *Large Class*. *Lazy Classes* are relatively easy to identify, because there exist few symptoms of low functionality. Therefore, for classes diagnosed as lazy there is no need to look for *Large Class* signs. The latter smell embraces multiple subtle symptoms, which are much harder to detect than *Lazy Class*, like multiple interfaces, multiple instances, multiple subclasses, so the knowledge of the *Lazy Class* presence allows giving up further exploration towards *Large Class*.

2.4 Aggregate Support

Aggregate support generalizes the plain support and rejection relations to a case of multiple source smells. A finite sets of detected smells A_1, A_2, …, A_m and absent smells B_1, B_2, …, B_m support a smell *C* as an aggregate, if they all support the existence of the smell *C* with higher certainty than any of individual smells A_i does or the smell *C* rejects the existence of any of smells B_j. Colloquially speaking, it is the synergy of several source smells (both present and absent) that increases the probability of existence of the target smell.

Aggregate support in several cases provides a stronger premise for many smells to exist. Source smells usually combine a broader spectrum of symptoms, which gives higher accuracy of the final result. The price for that is higher complexity of the detection process, resulting from the necessity of analyzing multiple source smells.

As an example, let us consider the following relation: if the given class is simultaneously composed of setters and getters, is not *Inappropriately Intimate*, and is the target of *Move Method* performed to remove a *Feature Envious* method, then it is a *Data Class*. The certainty factor for that relation is then higher than it would be without some of the supporting symptoms.

2.5 Transitive Support

The relation is a specific example of aggregate support with source smells depending on each other. Provided that there exist two plain support relations *p*: *A* supports *B* and *q*: *B* supports *C*, we can deduce the presence of a relation *r*: *A* supports *C*.

As an example we found the chain *Data Class* supports *Feature Envy* supports *Large Class*. *Large Classes* are classes that bear too much functionality. The over-functionality may result from improper class abstraction and combining several classes together. Other reasons include the presence of *Feature Envious* methods or *Inappropriate Intimacy* with other classes. Such a class needs to be split into smaller classes. Therefore, *Data Class* suggests the presence of the *Large Class*, because *Data Class* is related to *Feature Envy* (see 2.1) and the *Feature Envy* is related to *Large Class*.

2.6 Inclusion

Inclusion is a directed relation between smells *A* and *B*, in which *A* is a particular case of *B*. It means that every symptom revealing the smell *A* is also a sign of *B*'s presence. Therefore, by detecting the smell *A* we always find also the smell *B*.

Inclusion is slightly related to plain support, with exception that the special smell entirely fulfills symptoms specific to the general one.

Fowler's catalog contains a few examples of included smells. For instance, *Parallel Inheritance Hierarchies* is a special case of *Shotgun Surgery* smell.

2.7 Common Refactoring

The relations presented above concentrate on direct dependencies between smells. There exist other relations, which connect smells indirectly. One of binding elements is a common refactoring that once applied, affects all smells involved, either removing them or removing some and introducing the other.

For example, a *Move Method* applied to a *Lazy Class* may result in *Feature Envy* smell, because *Move Method* transfers the envious method outside, possibly reducing responsibility carried by that class.

3 Evaluation

To evaluate impact of our findings, we performed experiment on 830 classes coming from Apache Tomcat 5.5.4 [5] codebase. The project was selected for evaluation due to its high quality source code [9].

In subsequent sections we provide examples of how the information about smells could be exploited to detect other smells.

3.1 *Data Class* and *Feature Envy* Plain Support

In order to select *Data Class* candidates, we employed a simple getter/setter measure. We assumed that a class is a structure equivalent if the ratio of such methods is at least 80%. Other symptoms (improper encapsulation of fields and collections) were ignored. Candidates were then manually inspected to determine actual *Data Class* smell representatives. We also considered a method to be *Feature Envious* if it referenced other classes more frequently than its own class methods.

During inspection we found 26 classes, which had at least 80% of setter/getter methods, and as such were identified as *Data Classes*. Among them, 24 were referenced in *Feature Envious* methods. Therefore, it yields a high certainty factor (equal to 92%), which strongly suggests that the relation exists.

3.2 Plain Support of *Large Class* for *Feature Envy*

We analyzed the plain support relation between *Large Class* and *Feature Envy*. To measure class functionality we adopted four popular object-oriented metrics [6,7]. Their definitions and accepted thresholds taken from NASA's historical metrics database [10] are presented in Table 1.

Table 1. Metrics used for measuring functionality and their accepted thresholds (source: [6,10])

	Description	Max. accepted
NOM	Number of methods in the class	20
WMC	Sum of cyclomatic complexities of class methods	100
RFC	Number of methods + number of methods called by each of these methods (each method counted once)	100
CBO	Number of classes referencing the given class	5

We assumed that a class is considered large if at least one metric value exceeds the accepted threshold. Moreover, we also experimentally found that a *Large Class* has at least one *Feature Envious* method. Table 2 depicts the results of the evaluation. There exist 230 classes classified by common detectors as large. Out of these, 205 referenced *Feature Envious* methods. As we supposed, it turns out that most *Large Classes* have at least one *Feature Envious* method (certainty factor is equal to 89%), which helps in detecting the smell.

Table 2. Analysis of *Large Class*, *Inappropriate Intimacy* and *Feature Envy* smell relations (source: [11])

Metric	Value
Total number of analyzed classes	830
Number of classes with *Feature Envious* methods	463
Number of *Inappropriately Intimate* classes	159
Number of *Large Classes* found with common detectors	230
Number of *Large Classes* found exploiting relations between smells	501

3.3 Rejection

The rejection relation was analyzed with *Inappropriate Intimacy* and *Data Class* smells. *Inappropriately Intimate* classes "spend too much time delving in each other private parts" [3]. There are two violations covered by this smell:

- Bi-directional associations between classes, and
- Subclasses knowing more about their parents than their parents would like them to know.

Data Classes are mere data holders and thus do not have bi-directional associations with other classes. In other words, if a class is *Inappropriately Intimate*, then it cannot simultaneously be a *Data Class*.

Due to difficulties with automatic detection of the latter symptom of *Inappropriate Intimacy*, we considered only bi-directional associations between classes. Even a single association was considered to be smelly. The evaluation revealed 159 of 830 inspected classes to have such association. The number of possible checks for the *Data Class* smell was therefore reduced by 19%, because *Inappropriate Intimacy* excludes that smell.

3.4 Aggregate Support

As an example of this relation we evaluated *Data Class* structure equivalent smell [3]. A simple detector based on the setter/getter ratio found 66 candidates, out of which, after manual verification, only 26 have been found actually smelly (39% of accuracy).

We used this result to verify a hypothesis that information about support and rejection relations of other smells with *Data Class* smell may increase the accuracy of the detector, leaving the programmer with the smaller list of refactoring candidates to manual assessment. Therefore we evaluated the following aggregate relation: if a class has at least 80% of getter/setter methods, and is not *Inappropriately Intimate* smell, and is the target of *Move Method* refactoring of the *Feature Envy* method, then it is a *Data Class*.

Among 26 actual smell classes from 66 candidate classes we found 24 *Data Classes* referenced by *Feature Envy* methods and simultaneously being not *Inappropriately Intimate*. Another 12 were *Data Classes* referenced by *Inappropriately Intimate classes*. Therefore, there are only 30 classes left (out of 66) for manual inspection. The certainty factor for the analyzed aggregate support relation is then 92% (24 out of 26 candidate classes featured that smell).

3.5 Relations with a Common Refactoring

The knowledge about the relations between smells may be helpful also while removing them, i.e. at refactoring. We evaluated *Feature Envy* smell removal with *Move Method* transformation. Moved methods targeted also 21 *Data Classes* and simultaneously minimized the number of these smelly classes from 26 to 7. More details can be found in [11].

4 Conclusions

Every code smell is characterized by a different set of symptoms. To alleviate smell detection, we exploit the fact that some of them are related to others and carry information about them. The existence of already discovered smells becomes then a valuable indicator of other flaws. Whereas it infrequently plays a primary role in smell detection, it could be successfully utilized as an auxiliary source of smell-related data.

In the paper we identified six distinct inter-smell relations that appeared useful for smell detection. Another one relates smells through a common refactoring. The experiment showed that the use of the knowledge about already identified smells in Jakarta Tomcat code supports the detection process. We found examples of several smell dependencies, including simple, aggregate and transitive support and rejection relation. The certainty factor for those relations in that code suggests the existence of correlation among the dependent smells and applicability of this approach to smell detection.

Several activities benefited from the dependency analysis: in most cases it improved effectiveness and efficiency of the smell detection process; in others it suggested a single refactoring to remove several smells at once. Therefore, there are multiple applications of the inter-smell relations.

Future research plans include examination of other smells and their relations, and development of a tool for assisting a programmer in smell detection utilizing the presented approach.

Acknowledgements

The work has been supported by the Rector of Poznań University of Technology as a research grant BW/91-429.

References

1. Pearse T., Oman P.: Maintainability Measurements on Industrial Source Code Maintenance Activities. In: Proceedings of International Conference of Software Maintenance 1995, Opio (France), pp.295-303.
2. Beck K.: Extreme Programming Explained. Embrace Change. Addison-Wesley, 2000.
3. Fowler M.: Refactoring. Improving Design of Existing Code. Addison-Wesley, 1999.
4. Walter B., Pietrzak B.: Multi-criteria Detection of Bad Smells in the Code. In: Proceedings of 6th International Conference on Extreme Programming, 2005, Lecture Notes in Computer Science 3556, pp.154-161.
5. The Apache Jakarta Project: Tomcat 5.5.4, http://jakarta.apache.org/tomcat/index.html, January 2005.
6. 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.
7. Marinescu R., Using Object-oriented metrics for Automatic Design Flaws Detection in Large Scale Systems. ECOOP Workshop Reader 1998, Lecture Notes In Computer Science; Vol. 1543, pp.252-255.

8. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
9. Tomcat Defect Metric Report, http://www.reasoning.com/pdf/Tomcat_Metric_Report.pdf, visited in April 2005.
10. NASA Software Assurance Technology Center: SATC Historical Metrics Database, http://satc.gsfc.nasa.gov/metrics/codemetrics/oo/java/index.html, January 2005.
11. Pietrzak B., Walter B.: Exploring Bad Code Smells Dependencies. In: Zielinski K., Szmuc T. (eds.): Software Engineering: Evolution and Emerging Technologies. Frontiers in Artificial Intelligence and Applications, Vol. 130, pp.353-364.
12. Łukasiewicz J.: Die logischen Grundlagen der Wahrscheinilchkeitsrechnung. Kraków, 1913, in: L. Borkowski (ed.), Łukasiewicz J.: Selected Works. North Holland Publishing Company, Amsterdam, London, Polish Scientific Publishers, Warsaw, 1970.