A Taxonomy and an Initial Empirical Study of Bad Smells in Code

Mika Mäntylä, Jari Vanhanen, Casper Lassenius
Helsinki University of Technology
Software Business and Engineering Institute
P.O. Box 9600, FIN-02015 HUT, Finland
mika.mantyla@soberit.hut.fi, jari.vanhanen@hut.fi, casper.lassenius@hut.fi

Abstract

This paper presents research in progress, as well as tentative findings related to the empirical study of so called bad code smells. We present a taxonomy that categorizes similar bad smells. We believe the taxonomy makes the smells more understandable and recognizes the relationships between smells. Additionally, we present our initial findings from an empirical study of the use of the smells for evaluating code quality in a small Finnish software product company. Our findings indicate that the taxonomy for the smells could help explain the identified correlations between the subjective evaluations of the existence of the smells.

1. Introduction

Software system quality typically degenerates as the system is subjected to changes during the course of its lifetime. Successfully maintaining such a system demands that in addition to adding new functionality, existing code must be continuously refactored, i.e., improved without adding functionality.

The importance of refactoring has been recognized in the software product business. In [5] the researchers describe how Microsoft uses 20% of its development effort to re-develop the code base of its products. In [6] we can learn how Netscape's inability to refactor their code base hindered their software development, and how Microsoft's redesign efforts in Internet Explorer 3.0 project later paid off.

As an aid in identifying problematic code in object-oriented (OO) context, Fowler and Beck introduced 22 software structures as indicators of bad code, which they called "bad smells" [9]. These bad smells are supposed to help software developers in deciding when software needs refactoring.

To our knowledge no studies have been published in which bad smells would have been used as a basis for subjective code evaluation. Development level Anti-Patterns [2], which have some overlap with bad code smells, also seem to be lacking academic research.

In this paper we describe our initial efforts at empirically studying the bad smells. Thus, despite the very tentative findings of this paper, we hope that it helps stimulate more empirical research aiming at critically evaluating, validating and improving our understanding of subjective indicators of bad code quality.

2. Related work

Despite the fact that we found no studies directly studying the 22 bad smells presented in [9], earlier work has looked at several related topics. These are briefly summarized below.

The link between the structure of the software and maintainability is established in [14]. The study shows that software structure, which was measured using source code metrics, could predict maintainability of the software. Another study also shows that source code metrics and perceived maintainability have a correlation [10]. A study on measuring maintainability of OO systems shows that OO measures can predict maintainability [12]. Important work regarding software systems maintainability is also the construction of the maintainability index [3,4], which combines source code metrics and developers' opinions.

Some work has been done to automatically detect structures in the system that need refactoring. Some studies have focused strictly on clone detection or reduction, see [1,8] for more references. In [15] the researchers focused on automatically detecting and visualizing low cohesion in methods, attributes or classes, which act as a motivation for four refactorings. Katoka et al. [11] use an invariant detection tool to find the candidate spots for four possible refactorings. Touwré and Mens [17] use meta logic programing to find two code smells, which they called Obsolete Parameter and Inappropriate Interfaces. Some studies [7,13] have also used the historical data to identify the spots, where programmers have made changes or refactorings to the software.

3. Smell taxonomy

This chapter briefly introduces the bad code smells identified by Fowler and Beck [9] and proposes a higher level taxonomy for classifying them. The original authors present the 22 bad smells in a single flat list and do not provide any classification of the smells. Since several smells are closely related and the number of the smells is quite high, we feel that this taxonomy, which categorizes similar bad smells, is beneficial. We believe that the taxonomy makes the smells more understandable and recognizes the relationships between the smells. The classes we propose are: *bloaters, object-orientation abusers, change preventers, dispensables, encapsulators, couplers*, and *others*.

3.1. Bloaters

Bloaters represent something in the code that has grown so large that it cannot be effectively handled. The smells in the Bloater category are: *Long Method*, *Large Class*, *Primitive Obsession*, *Long Parameter List*, and *Data Clumps*. In general it is more difficult to understand or modify a single long method than several smaller methods. The same kind of argument holds also for Long Parameter List and Large Class. Primitive Obsession does not actually represent a bloat, but is a symptom causing bloats, because it refers to situations in which the logic handling the data appears in large classes and long methods. For Data Clumps we could also argue that it should be in the Object-Orientation Abusers, because in theory a class should be created from each Data Clump. However, since Data Clumps often appear with the Long Parameter List smell we have decided to include it in this category.

3.2. Object-Orientation Abusers

The smells in the Object-Orientation Abuser category are: *Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces,* and *Parallel Inheritance Hierarchies*. This category of smells is related to cases where the solution does not fully exploit the possibilities of OO design. In Switch Statements smell type codes are used and detected using switch statements. In OO software design the need for type codes should, however, be handled by creating subclasses. The Parallel Inheritance Hierarchies and Refused Bequest smells lack proper inheritance design, which is one of the key elements in OO programming. The Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse. The Temporary Field smell means a case where a variable is in the class scope, when it should be in the method scope. This violates the information hiding principle.

3.3. Change Preventers

The third category of smells refers to code structures that considerably hinder the modification of the software. The smells in the Change Preventers category are: *Divergent Change* and *Shotgun Surgery*. The key is that according to [9] the classes and the possible changes need to have a one-to-one relationship, e.g., one class that is modified when a database is changed, another class which is modified when new sorting algorithms are added. The smells in this category violate this principle. The Divergent Change smell means that we have a single class that is modified in many different types of changes. The Shotgun Surgery smell is the opposite. There we need to modify many classes when making a single change to a system.

3.4. Dispensables

The smells in the Dispensables category are *Lazy Class, Data Class, Duplicate Code,* and *Speculative Generality*.
These smells represent something unnecessary that should be removed from the code. Classes that are not doing enough need to be removed or their responsibility needs to be increased. Data Class and Lazy Class represent such smells. Also unused or redundant code needs to be removed, which is the case with Duplicate Code and Speculative Generality.

Interestingly, Fowler and Beck [9] do not present a smell for dead code. We find this quite surprising, since in our experience it is a quite common problem. Dead code is code that has been used in the past, but is currently never executed. Dead code hinders code comprehension and makes the current program structure less obvious.

3.5. Encapsulators

The Encapsulators deal with data communication mechanisms or encapsulation. The smells in the Encapsulators category are *Message Chains* and *Middle Man*. The smells in this category are somewhat opposite, meaning that decreasing one smell will cause the other to increase. Removing the Message Chains smell does not always cause the Middle Man smell and vice versa, since the best solution is often to restructure the class hierarchy by moving methods or adding subclasses. Naturally, one could argue that the Message Chains smell belongs in the Couplers group and that the Middle Man smell belongs in the Object-Orientation Abusers. We believe that in order to get a better understanding of these smells they should be introduced together, because they both deal with the way objects, data, or operations are accessed.

3.6. Couplers

There are two coupling related smells, which are *Fea-*

ture Envy and *Inappropriate Intimacy*. The Feature Envy smell means a case where one method is too interested in other classes, and the Inappropriate Intimacy smell means that two classes are coupled tightly to each other. Both of these smells represent high coupling, which is against the OO design principles. Of course, here we could make an argument that these smells should belong in the Object-Orientation Abusers group, but since they both focus strictly on coupling, we think it is better if they are introduced in their own group.

3.7. Others

This class contains the two remaining smells *Incomplete Library Class*, and *Comments* that do not fit into any of the categories above.

4. Empirical study

This section describes an initial empirical study on the use of the bad smells for evaluating code quality, and shows how the results support the presented taxonomy.

4.1. Description of the survey

We tested the use of the smells in practice by performing a survey directed at the developers of a small Finnish software product company. In the survey we asked the developers to evaluate the degree to which they thought the smells existed in the different modules of the company's software products. The size of the software modules varied between 15 and 80 KLOC, their age was 0-8 years and the language used was Delphi/Kylix. We used a seven point Likert scale, with one indicating that a particular smell did not exist in the module at all and seven representing a lot of the smell in the module. Of 18 developers, 8 regular developers and 4 lead developers responded to the survey. We received totally 37 module-smell evaluations, i.e., data points where a developer had evaluated a module. The average number of modules evaluated by a developer was 3,08, varying from 2 to 6.

4.2. Correlations between smells

It seems natural that the existence of some smells would correlate positively with some other smells while others would have a negative correlation. In our small-sample study we found negative correlations only with the Primitive Obsession smell. Figure 1 shows only the strongest (r > 0,575) and the most significant (p < 0,01) correlations between the smells. Figure 1 also maps the proposed taxonomy to the correlations.

It seems that Inappropriate Intimacy hooks the Change Preventer smells together, since it has a strong correlation with both of them. Change Preventers are also correlated with each other, but the correlation is not as strong as the correlation between them and the Inappropriate Intimacy smell.

The Message Chains smell correlates with its opposite smell Middle Man. This could be due to that both smells indicate encapsulation or lack of it between objects and can therefore be easily confused.

The Object-Orientation Abusers have high in-group correlation. It would seem natural that Parallel Inheritance Hierarchies causes Refused Bequest (a smell in which a class does not support everything it has inherited), which again explains the need to detect type codes with Switch Statements. However, we can offer no explanation for the very high correlation between Refused Bequest and Primitive Obsession.

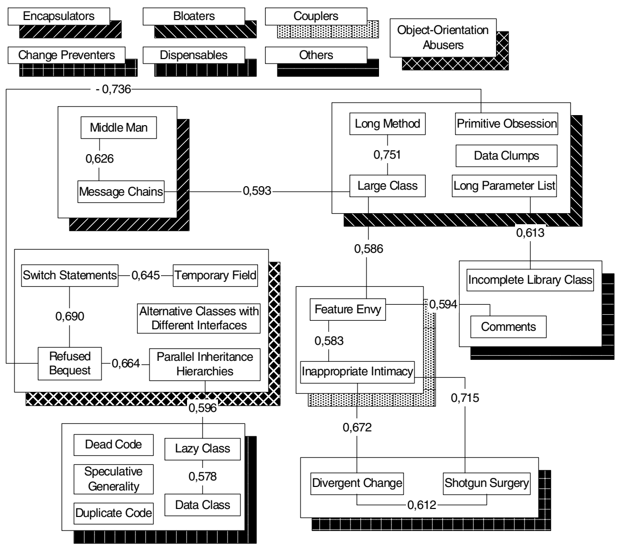

Figure 1. Spearman correlations between smells (r > 0,575 and p < 0,01) and the taxonomy

No support for the claims in [9] about the correlation of the Large Class and Duplicate Code smells was found. However, we must bear in mind that Duplicate Code is difficult to spot and that the developers' opinions might be biased.

In Figure 1, we can identify two more distinct groups. One is formed around the Couplers and the Large Class smell. The other group seems to be formed around the OO abusers and dispensable classes. One could speculate that large classes and an inability to minimize coupling could cause the first group. The second group seems to focus around poor inheritance usage and dispensable classes. This indicates that understanding when and how to use inheritance and remembering the two basic principles of minimizing coupling and maximizing cohesion [16] helps prevent smells.

It also appears that the taxonomy helps to capture strong correlations within groups. Figure 1 shows that 8 correlations are within the proposed groups but there are also 8 correlations between groups. Since we had 23 smells in our survey (22 from [9] and Dead Code that was discussed in Section 3.4), this results to 253 correlations between all the smells. The total number of correlations within all groups is 34. The total number of between group correlations is naturally 253-34= 219. This means that 23,53% (8/34) of the total amount of within-group correlations are strong, whereas only 3,65% (8/219) of between-group correlations are among the strongest. This result seems to indicate that the theoretical taxonomy is also supported by the correlations between the code smells.

5. Conclusions and future work

This paper makes two contributions. First, it proposes a subjective taxonomy that categorizes similar bad smells. We feel that this taxonomy makes the smells more understandable than the single flat list of 22 bad code smells that was presented in [9]. The taxonomy also helps to recognize the relationships between smells. This taxonomy is initial, so it probably has weaknesses and needs to be improved in the future. For instance, one could group the smells based on the structure that the smells effect, i.e., some smells exist on the methods (e.g. Long Method, Feature Envy), while others exist in classes (e.g. Large Class, Lazy Class), and some smells appear in the relationships between classes (Message Chains, Inappropriate Intimacy). This type of approach would probably result in a different taxonomy than the one proposed here.

The second contribution comes from the empirical study, which provides initial correlations between the smells. These correlations can help us understand how different smells are connected to each other. Also, since some smells can be measured with tools, the correlations could be useful in indicating the presence of the smells that cannot be discovered automatically. The taxonomy for the bad code smells, which is based on subjective but logical groups, seems to be useful in analyzing the smell correlations. This paper also shows that a much greater deal of within group correlations are strong, when compared to between-group correlations.

This paper has described ongoing research on bad code smells. In the future we plan to study how the demographic data explains the smell evaluations, and the correlation between various source code metrics and smell evaluations. We feel that controlled experiments are also needed with smaller and pre-examined software modules, because in some cases there were great fluctuations between the smell evaluations of different people for the same module.

References

[1] Balazinska, M., Merlo, E., Dagenais, M., Lague, B., and Kontogiannis, K., "Advanced clone-analysis to support object-oriented system refactoring", IEEE, *Proceedings of Seventh Working Conference on Reverse Engineering*, 23.11.2000, pp. 98-107.

[2] Brown, W. J., Malveau, R. C., McCormick, H. W., and Mowbray, T. J., *AntiPatterns Refactoring Software, Architectures, and Projects in Crisis*, New York: Wiley, 1998.

[3] Coleman, D., Ash, D., Lowther, B., and Oman, P. W., "Using Metrics to Evaluate Software System Maintainability," *Computer*, vol. 27, no. 8, 1994, pp. 44-49.

[4] Coleman, D., Lowther, B., and Oman, P. W., "The Application of Software Maintainability Models in Industrial Software Systems," *Journal of Systems and Software*, vol. 29, no. 1, 1995, pp. 3-16.

[5] Cusumano, M. A. and Selby, R. W., *Microsoft Secrets*, USA: The Free Press, 1995.

[6] Cusumano, M. A. and Yoffie, D. B., *Competing on Internet Time*, New York, USA: The Free Press, 1998.

[7] Demeyer, S., Ducasse, S., and Nierstrasz, O., "Finding refactorings via change metrics", ACM Press, *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, 15.10.2000, pp. 166-177.

[8] Ducasse, S., Rieger, M., and Demeyer, S., "A language independent approach for detecting duplicated code", *Proceedings of the International Conference on Software Maintenance*, 30.8.1999, pp. 109-118.

[9] Fowler, M. and Beck, K., "Bad Smells in Code," *Refactoring: Improving the Design of Existing Code* Addison-Wesley, 2000, pp. 75-88.

[10] Kafura, D. G. and Reddy, G. R., "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 13, no. 3, 1987, pp. 335-343.

[11] Kataoka, Y., Ernst, M. D., Griswold, W. G., and Notkin, D., "Automated support for program refactoring using invariants", IEEE, *Proceedings of International Conference on Software Maintenance*, 7.11.2001, pp. 736-743.

[12] Li, W. and Henry, S. M., "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software*, vol. 23, no. 2, 1993, pp. 111-122.

[13] Maruyama, K. and Shima, K., "Automatic method refactoring using weighted dependence graphs", IEEE, *Proceedings of the 1999 International Conference on Software Engineering*, 16.5.1999, pp. 236-245.

[14] Rombach, D. H., "Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, vol. 13, no. 3, 1987, pp. 344-354.

[15] Simon, F., Steinbruckner, F., and Lewerentz, C., "Metrics based refactoring", IEEE, *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, 14.3.2001, pp. 30-38.

[16] Stevens, W., Myers, G., and Constantine, L., "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, 1974, pp. 115-139.

[17] Touwré, T. and Mens, T., "Identifying refactoring opportunities using logic meta programming", IEEE, *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, 2003*, 26.3.2003, pp. 91-100.