Facilitating Software Refactoring with Appropriate Resolution Order of Bad Smells

Hui Liu[1,2], Limei Yang[1], Zhendong Niu[1], Zhiyi Ma[2,3] and Weizhong Shao[2,3]
[1]School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
[2]Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing 100871, China
[3]Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
{liuhui04, mzy,wzshao}@sei.pku.edu.cn, eicoyang@gmail.com, zniu@bit.edu.cn

ABSTRACT

Bad smell is a key concept in software refactoring. We have a bunch of bad smells, refactoring rules, and refactoring tools, but we do not know which kind of bad smells should be resolved first. The resolution of one kind of bad smells may have impact on the resolution of other bad smells. Consequently, different resolution orders of the same set of bad smells may require different effort, and/or lead to different quality improvement. In order to ease the work and maximize the effect of refactoring, we try to analyze the relationships among different kinds of bad smells, and their impact on resolution orders of these bad smells. With the analysis, we recommend a resolution order of common bad smells. The main contribution of this paper is to motivate the necessity to arrange resolution orders of bad smells, and recommend a resolution order of common bad smells.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*

General Terms

Design

Keywords

Software Refactoring, Resolution Order, Schedule, Quality

1. INTRODUCTION

One of the key issues in software refactoring is to determine what kind of source code should be refactored. Experts have already summarized some typical situations which may deserve refactoring. Kent Beck and Martin Fowler call them *Bad Smells* [1, Chapter 3], indicating that some part of the source code smells terrible. In other words, *Bad Smells* (e.g., *Duplicate Code*) are signs of potential problems in code. Definition and explanation of a bunch of bad smells can be found in the third chapter of [1]. It is also possible to find even more kinds of bad smells in other books [3] or on Internet[1]. These bad smells are usually linked to corresponding refactoring rules which can help to dispel the bad smells.

Different kinds of bad smells are not independent of each other, and the resolution of one kind of bad smells may influence (ease or complicate) the following resolution of other bad smells. But to date, we know little about the relationships among different kinds of bad smells, let along the impact that the resolution of one bad smell has on the following resolution of other (left) bad smells.

As a result of the impact, different resolution orders of the same set of bad smells may require different effort and/or lead to different results. Consequently, it is possible to ease the work of refactoring by arranging appropriate resolution orders of bad smells, taking full advantage of the impact. Likewise, it is also possible to maximize the effect of refactoring if bad smells are resolved in an appropriate order.

But to our knowledge, there is no published research in resolution orders of bad smells. One possible reason is that software refactoring, especially semi-automatic and bad smells centric refactoring, is an emerging research field. In this paper, we analyze the relationships among bad smells, and their influence on the resolution orders.

The paper is organized as follows. Section 2 analyzes resolution orders of different kinds of bad smells, and Section 3 recommends a resolution order for these bad smells. Section 4 makes a conclusion.

2. ANALYSIS

2.1 Overview

Software refactoring is not yet completely formalized. To date, both bad smells and refactoring rules are described in an informal or semi-formal way [1, 3]. As a result, it is hard, if not impossible, to find a formal method to analyze the impact of resolving one kind of bad smells on the resolution of other kinds of bad smells. Consequently, we have to analyze the impact in an informal way.

[1]http://wiki.java.net/bin/view/People/SmellsToRefactorings

ESEC-FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.

Table 1: Evaluated Bad Smells

	Names of Bad Smells
1	Duplicate Code
2	Divergent Change
3	Long Method
4	Large Class
5	Long Parameter List
6	Feature Envy
7	Useless Field
8	Useless Method
9	Useless Class
10	Primitive Obsession

For each pair of bad smells, we ask the following five questions:

- Will different resolution orders lead to different resulting systems (different effect)? If yes, why?
- Will the resolution of a bad smell ease the detection of the other (ease detection)? If yes, why?
- Will the resolution of a bad smell complicate the detection of the other (complicate detection)? If yes, why?
- Will the resolution of a bad smell ease the modification (resolution) on the other bad smell (ease modification)? If yes, why?
- Will the resolution of a bad smell complicate the modification on the other (complicate modification)? If yes, why? Here modification includes all operations on detected bad smells.

It is also possible that the resolution orders do not matter for two kinds of bad smells. For example, it does not matter *Simple Primitive Obsession* is resolved before *Long Parameter List* or not. For this case, we just leave them alone, and no resolution order is recommended.

The relationships among the evaluated bad smells are divided into six categories, and each of the following subsection concentrates on one of them. It is possible that a pair of bad smells appears in more than one subsection because the division is not exclusive: Different categories concern different aspects of the relationships, and a specific relationship may be investigated in more than one aspect.

The relationships are represented as '*Bad Smells A* → *Bad Smells B*', indicates that *Bad Smells A* had better be resolved before *Bad Smells B*.

2.2 Evaluated Bad Smells

As an initial study, we take 10 kinds of bad smells for evaluation. The evaluated bad smells are listed in Table 1. Detailed explanation of the bad smells can be found in the book by Kent Beck and Martin Fowler [1, Chapter 3].

Bad smell *Primitive Obsession* is further divided into three subcategories: *Simple Primitive Obsession*, *Simple Type Code*, and *Complex Type Code* [1]. The division is necessary because different refactoring rules should be applied depending on whether the primitive data are type code, and whether the type code influences the behaviors of the enclosing class.

2.3 Side Effect

The resolution of one kind of bad smells may remove other kinds of bad smells as a side effect.

Duplicate Code* → *Long Method Resolution of *Duplicate Code* may remove *Long Method* as a side effect.

Duplicate Code* → *Large Class Large classes usually come from pool distribution of responsibilities. But sometimes, duplicate code may also lead to large classes. In this case, no mater which kind of bad smells are resolved first, the work is the same: Remove duplicate code.

If we resolve *Duplicate Code* first, we can detect duplicate code with professional detection tools, e.g. CCFinder [2], and remove the duplication. As a side effect, the size of the enclosing class is reduced, and bad smell *Large Class* is dispelled.

However, if we resolve *Large Class* first, things may be more complicated. We have to find the reason (duplicate code in this case) manually why the class is so large and how to break it down. It is not easy, especially when duplicate code involves more than one file.

Divergent Change* → *Large Class/Long Method *Divergent Change* indicates that different parts of a class are commonly changed for different reasons. The solution to *Divergent Change* is to decompose the class, and sometimes involved methods may also be decomposed. Of course, the decomposition will reduce the size and complexity of the decomposed classes and methods, which may dispel bad smells *Large Class* and *Long Method*.

Long Method* → *Long Parameter List As a solution to *Long Method*, some parts of the method may be extracted as new methods. Usually the extracted new methods should be called within the old one, and thus do not help to reduce the parameter list. But sometimes the responsibility of the old method may be redefined (responsibilities of the extracted parts of the method are excluded), and it never calls the extracted new methods again. In this case, the parameter list would be reduced.

Divergent Change* → *Long Parameter List Since the resolution of *Divergent Change* may involve decomposition of methods, it may reduce parameters of the decomposed methods as a side effect: A method may be decomposed into two independent methods with fewer parameters.

Complex Type Code* → *Long Method/Large Class The resolution of *Complex Type Code* involves decomposition of classes and methods. The decomposition of course helps to reduce the length and complexity of the methods and classes, and thus helps to dispel bad smells *Long Method* and *Large Class*.

Complex Type Code* → *Divergent Change *Complex Type Code* indicates that relatively independent classes are combined together with a type code. It is very possible that different parts of the composed class change independently for different reasons, which is known as *Divergent Change*. Fortunately, if *Complex Type Code* is cleaned, *Divergent Change* caused by it will be removed as a side effect.

Feature Envy* → *Long Method During the resolution of *Feature Envy*, some parts of the involved method may be extracted and moved to other class. It is in fact a special kind of decomposition of methods which helps to dispel *Long Method*.

Feature Envy* → *Large Class Since the resolution of *Feature Envy* involves redistribution of computation, it may help to dispel *Large Class*. Furthermore, it may also results in *Large Class* by the redistribution. Consequently, if *Feature Envy* is resolved after *Large Class*, large classes caused by the resolution of *Feature Envy* may be missed.

Feature Envy* → *Long Parameter List The resolution of *Feature Envy* may involve method extraction. During the extraction, it is possible to redefine the involved method so as that it does not call the extracted new methods. In this case, the resolution of *Feature Envy* reduces parameters of the decomposed methods. In turn, the reduction of parameters may dispel bad smells *Long Parameter List*.

Feature Envy* → *Divergent Change Some computation may be more interested in another class than the enclosing class, which is known as *Feature Envy*. It may lead to *Divergent Change*: The computation changes with the interested class instead of the enclosing class. As a result, different parts of the enclosing class change independently. Fortunately, if *Feature Envy* is removed, *Divergent Change* caused by it may be removed as a side effect.

Useless Field* → *Simple Primitive Obsession If a primitive field is useless, you do not have to replace it with an object. Instead, you should just remove it, and you do not have to worry about primitive obsession with the field anymore.

Useless Class → If a class is useless, just remove it. The removal would remove all bad smells associated with the class at the same time.

Useless Method → If a method is useless, just remove it. The removal would remove all bad smells associated with the method at the same time.

2.4 Ease Detection

The detection of bad smells is difficult, even with professional detection tools. Consequently, if the resolution of one kind of bad smells can ease the detection of other kinds of bad smells, it had better be resolved before others.

Useless Method* → *Useless Field Removal of useless methods would ease the detection of useless fields that used only by useless methods. After the removal of *Useless Method*, we do not have to analyze whether the enclosing methods of fields are useless or useful while detecting *Useless Field*.

2.5 Complicate Detection

Duplicate Code* → *Simple Primitive Obsession *Replace Primitive Data with Objects*, as a solution to *Simple Primitive Obsession*, would modify clones. To be worse, it is very possible that not all duplicate fragments are modified in the same way (the primitive data in different clones may be replaced with different objects). But any literal difference, even the difference in names of objects, would complicate the detection of duplications.

Duplicate Code* → *Feature Envy *Move Features between Classes*, as a solution to *Feature Envy*, would potentially modify duplications. As a result, it had better be carried out after duplications are removed.

2.6 Different Results

For some bad smells, different resolution orders may lead to different results.

Feature Envy* → *Large Class A large class may contains some computation which is more interested in features of another class than those of the enclosing class (*Feature Envy*). To reduce coupling between classes, the computation should be moved to the class it is interested in.

If *Large Class* is resolved before *Feature Envy*, the contained computation interested in another class (*RefClass*) may be decomposed and redistributed into different classes when the large class is decomposed into small classes (as a solution to bad smell *Large Class*). As a result, each new small class refers a few features of *RefClass*. But the amount of features referred by each new class is small enough that it will not be treated as *Feature Envy* any more. Consequently, the computation interested in features of *RefClass* will not be moved to *RefClass*.

If *Feature Envy* is resolved before *Large Class*, however, the computation interested in features of *RefClass* will be moved to *RefClass*, as a solution to *Feature Envy*.

Feature Envy* → *Divergent Change The resolution of *Divergent Change* is similar to that of *Large Class*: Breaking the class into small ones. However, *Feature Envy* may be concealed by the decomposition if *Divergent Change* is resolved first. To clear both of them properly, we should resolve *Feature Envy* before *Divergent Change*.

Long Method* → *Large Class In order to dispel bad smells *Large Class*, we should decompose the large class into a few small classes, which is in fact a redistribution of responsibilities in class level.

The resolution of bad smells *Long Method* would redistribute responsibilities of the long method into a few reusable short methods. It is in fact a redistribution of responsibilities in method level.

If the redistribution of responsibilities could be done from the bottom up, the redistribution could be more reasonable and throughout. Suppose that a long method could be decomposed into three independent ones, and the enclosing large class should be decomposed into two. If the long method is decomposed first, the three new methods could be redistributed into different classes in the decomposition of the enclosing class. But if it is done after the class level decomposition, it has no chance to be redistributed across classes.

2.7 Complicate Modification

It is not only time-consuming, but also error-prone to modify complicate source code. So, if the resolution of a kind of bad smells may complicate the modification of following refactorings, we should resolve them as late as possible.

Large Class/Divergent Change/Complex Type Code* → *Long Parameter List *Complicate Modification* is intensively associated with *Long Parameter List* and *Class Decomposition*. If a parameter list contains many items (fields) of a specific class, we had better replace the items with an instance of the class in order to reduce the amount of the parameters. But after that, if the class has to be decomposed for some reason (as a solution to bad smells *Large Class*, *Divergent Change*, or *Complex Type Code*), the instance of the class in the parameter list introduced while resolving *Long Parameter List* should be replaced. To be worse, the replacement is quite difficult: We have to check the method's body carefully to tell which fields and methods of the instance have been accessed by the method.

However, if we clean *Long Parameter List* after the large class is decomposed, things can be much easier. We can easily tell which classes the method is really interested in by checking the parameter list instead of the complex body of the method. The parameter list itself tells us which fields the method is interested in.

2.8 Ease Modification

In contrast to *Complicate Modification*, *Ease Modification* indicates that resolving one kind of bad smells may ease the modification of following refactorings.

Duplicate Code* → *Simple Primitive Obsession/Simple Type Code/Complex Type Code *Ease Modification* is intensively associated with bad smell *Duplicated Code* because the resolution of *Duplicated Code* replaces duplicate (or similar) fragments with a single copy of the fragment and a few method calls, and thus reduces the amount of occurrences of bad smells in the fragments. As a result, less code needs modification, and thus modification required by refactoring becomes easier.

Simple Type Code and *Complex Type Code* are also possible to appear in duplicate fragments, and the resolution of *Duplicate Code* can help to reduce the occurrences of the bad smells. Consequently, they had better be resolved after *Duplicate Code*.

3. RESOLUTION ORDER OF BAD SMELLS

With the analysis results in Section 2, this section tries to recommend a resolution order of the evaluated bad smells in the following two steps.

First, we represent the analysis results with a directed graph (the graph is too complicated to be presented here, but it can be downloaded from [2]). Each vertex of the directed graph represents a category of bad smells whose name is presented in the label of the node. Each directed edge represents a preferred resolution order of the two kinds of bad smells represented by adjacent vertexes of the edge, i.e., edge $\langle A, B\rangle$ indicates that A had better be resolved before B. The reason why a resolution order is better than the other (the reversed) is presented in the label of the edge.

The graph is in fact a multigraph containing parallel edges, which suggests that one resolution order may be justified by more than one reason. However, the graph is too complicated to read.

Second, we apply an extended topological sort algorithm to the directed graph to get a clear resolution order of different kinds of bad smells. The only difference between the extended and traditional topological sort algorithms is that when there are two or more vertexes available (whose indegree is zero) in each iteration, the extended algorithm selects all of them as candidates for the next execution step (branches). For example, once *Duplicate Code* is resolved, we can begin to deal with anyone of the following four kinds of bad smells: *Simple Primitive Obsession*, *Feature Envy*, *Simple Type Code*, and *Complex Type Code*. It is valuable to leave the decision to software engineers. In contrast, traditional topological sort algorithms produce a single order, leaving no choice to software engineers.

The resulting resolution order is also presented as a directed graph as shown in Fig.1. All edges of the graph of Fig.1 are downwards, which suggests that bad smells in the hierarchy could be handled from the top down. Edges of the graph in Fig.1 are directed but not labeled. In other words, they just indicate resolution orders among different kinds of bad smells, but reasons are omitted for clarity.

[2]http://sei.pku.edu.cn/∼liuhui04/ResolutionOrder.bmp

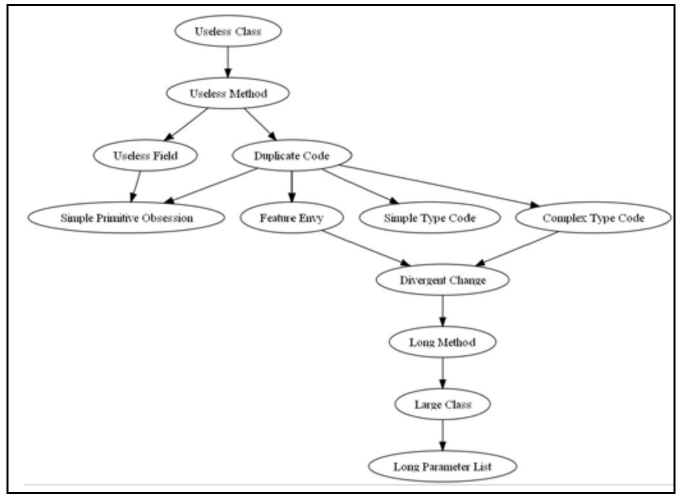

Figure 1: Resolution Order of Bad Smells

4. CONCLUSION AND FUTURE WORK

In this paper, we illustrate how resolution orders of bad smells influence the activities of refactoring, and how to ease the work of refactoring by arranging appropriate resolution orders of bad smells. We analyze the relationships among different kinds of bad smells. And then, we recommend a resolution order of these bad smells. Tool support for resolution orders of bad smells, similar to that for software development processes, is also under investigation.

5. ACKNOWLEDGEMENTS

The authors would like to say thanks to the students in the laboratory. They did most of the work in the evaluation. The authors thank the anonymous reviewers for their valuable comments and suggestions.

The work is funded by the National Natural Science Foundation of China No.60773152, the National Grand Fundamental Research 973 Program of China No.2005CB321805, the National High-Tech Research and Development Plan of China No.2007AA01Z127 and 2007AA010301, Open Research Fund Program of Key Laboratory of High Confidence Software Technologies, Ministry of Education No. HCST200802, Basic Research Found of Beijing Institute of Technology.

6. REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[2] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[3] W. C. Wake. *Refactoring Workbook*. Addison Wesley, August 2003.