Bad Smelling Concept in Software Refactoring

Ganesh B. Regulwar[+] and Raju M. Tugnayat

Jawaharlal Darda Institute of Engineering & Technology, MIDC, Lohara, Yavaymal (MS), INDIA

Abstract: "This paper discusses refactoring, which is one of the techniques to keep software maintainable. However, refactoring itself will not bring the full benefits, if we do not understand when refactoring needs to be applied. To make it easier for a software developer to decide whether certain software needs refactoring or not, Fowler & Beck (Fowler & Beck 2000) give a list of bad code smells. Fowler & Beck's idea was that bad code smells are a more concrete indication for the refactoring need than some vague idea of programming aesthetics. Fowler & Beck also acclaim that no set of precise metrics can be given to identify the need of refactoring. Therefore, bad code smells are kind of a compromise between the vague programming aesthetics and precise source code metrics. With bad code smells the reader must bear in mind that some smells represent the opposite ends of the same attribute. For example, the size of a class could be a single attribute, and in one end of the attribute the existing smell is called Large Class and in the other it is referred to as Lazy Class. In addition, for each bad code smell Fowler (Fowler 2000) introduces a set of refactoring (move methods, inline temp, etc) with step wise instructions on how each smell can be removed. Therefore, the reader should realize that the refactoring concept also includes detailed instructions on how to actually improve the source code. The purpose of this section is to introduce the 22 bad code smells, which are listed further.

Keywords: Improves the Design of Software, Bad Smell, Software Maintainable

1. Introduction

The Definition of Refactoring**:**

Refactoring is relatively a new area of research and so is not well defined. There are a vast number of definitions for refactoring, most of them are mentioned below:

Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactoring (verb): to restructure software by applying a series of refactoring without changing its observable behavior.

- Refactoring is the process of taking an object design and rearranging it in various ways to make the design more flexible and/or reusable. There are several reasons you might want to do this, efficiency and maintainability being probably the most important.
- To refactor programming code is to rewrite the code, to "clean it up".
- Refactoring is the moving of units of functionality from one place to another in your program.
- Refactoring has as a primary objective, getting each piece of functionality to exist in exactly one place in the software

2. Bad Smells in Code

Knowing where to refactor within in a system is quite a challenge to identify areas of bad design. These areas of bad design are known as "Bad Smells" or "Stinks" within code. Finding these areas are more related to "human intuition" than an exact science. The developers experience is relied upon in identify these "Bad Smells".

However, refactoring itself will not bring the full benefits, if we do not understand when refactoring needs to be applied. To make it easier for a software developer to decide whether certain software needs refactoring or not, Fowler & Beck (Fowler & Beck 2000) give a list of bad code smells. **Code smell** is any symptom that indicating something wrong. It generally indicates that the code should be refactored or the overall design should be re-examined. The term appears to have been coined by Kent Beck .Usage of the

[+] ganeshregulwar@gmail.com; tugnayatrm@rediffmaill.com

term increased after it was featured in Refactoring. Bad code exhibits certain characteristics that can be rectified using Refactoring. These are called **Bad Smells**.

3. Types of Smell

The purpose of this section is to introduce those 18 bad code smells, which are listed below:

Long Method is a method that is too long, so it is difficult to understand, change, or extend. Fowler and Beck (Fowler & Beck 2000) strongly believe in short methods. The longer a procedure is the more difficult it is to understand. Nearly all of the time all you have to do to shorten a method is *Extract Method*. If you try to use *Extract Method* and end up passing a lot of parameters, you can often use *Replace Temp with Query* to eliminate the temps and slim down the long list of parameters with *Introduce Parameter Object* and *Preserve Whole Object* .Long Method can be summarized -

- Symptoms: Too long method that is difficult to understand and reuse
- Detection: Cyclomatic complexity (polynomial metrics)
- Relationship: N/A
- Solutions: Decompose Conditional, Extract Method, Replace Method with Method Object, and Replace Temp with Query
- Identifications: Medium
- Removal: Difficult
- Impact: Strong

Large Class means that a class is trying to do too much. These classes have too many instance variables or methods, duplicated code cannot be far behind. A class with too much code is also a breeding ground for duplication. In both cases *Extract Class* and *Extract Subclass* will work. This smell can be summarized as follows –

- Symptoms: Too many instance variables or methods
- Detection: Lack of Cohesion Methods or measuring class size
- Relationship: Blob/God Object
- Solutions: Extract Class, Extract Interface, Extract Subclass, and Introduce Foreign Method
- Identifications: Medium
- Removal: Difficult
- Impact: Strong

Primitive Obsession smell represents a case where primitives are used instead of small classes. For example, to represent money, programmers use primitives rather than creating a separate class that could encapsulate the necessary functionality like currency conversion. Primitive Obsession has following features –

- Symptoms: Using primitive instead of small classes
- Detection: N/A
- Relationship: N/A
- Solutions: Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Data Value with Object, Replace Type Code with Subclass/State/strategy
- Identifications: Medium
- Removal: Medium
- Impact: Strong

Long Parameter List is a parameter list that is too long and thus difficult to understand. With objects you don't need to pass in everything the method needs, instead you pass in enough so the method can get to everything it needs. This is goodness, because long parameter lists are hard to understand, because they inconsistent and difficult to use because you are forever changing them as you need more data. Use *Replace Parameter with Method* when you can get the data in one parameter by making a request of an object you already know about. This smell can be summarized as –

- Symptoms: A method with too many parameters that is difficult to understand
- Detection: Count the number of parameters
- Relationship: N/A
- Solutions: Introduce Parameter Object, Replace Method with Method Object, and Preserve Whole Object
- Identifications: Medium

- Removal: Medium,
- Impact: Medium

Data Clumps smell means that software has data items that often appear together. Often you will see the same three or four data items together in lots of places:

a) Fields in a couple of classes

b) Parameters in many method signatures:Removing one of the group's data items means that the those items that are left make no sense, e.g., integers specifying RGB colors. Data clumps has the following features-

- Symptoms: Data is always coherent with each other.
- Detection: If one value is removed, the data set will be meaningless.
- Relationship: Magic Numbers/Magic String
- Solutions: Extract Class, Introduce Parameter Object, and Preserve Whole Object
- Identifications: Medium
- Removal: difficult
- Impact: Medium

Switch Statements smell has a slightly misleading name, since a switch operator does not necessarily imply a smell. The smell means a case where type codes or runtime class type detection are used instead of polymorphism. Also type codes passed on methods are an instance of this smell. "Most times you see a switch statement you should consider polymorphism". Use *Extract Method* to extract the switch statement and then *Move Method* to get it into the class where the polymorphism is needed. If you only have a few cases that affect a single method then polymorphism is overkill. In this case *Replace Parameter with Explicit Methods* is a good option. If one of your conditional cases is null, try *Introduce Null Object*. Switch Statements can be summarized as shown below –

- Symptoms: Replacing polymorphism with type codes or runtime class type detection
- Detection: runtime detection
- Relationship: N/A
- Solutions: Introduce Null Object, Replace Conditional with Polymorphism, Replace Method with Explicit Method, Replace Type Code with Subclass/State/Strategy
- Identifications: Medium
- Removal: Medium
- Impact: Weak

Temporary Field smell means that class has a variable which is only used in some situations. Sometimes you will see an object in which an instance variable is set only in certain circumstances. This can make the code difficult to understand because we usually expect an object to use all of its variables. Use *Extract Class* to create a home for these orphan variables by putting all the code that uses the variable into the component. You can also eliminate conditional code by using *Introduce Null Object* to create an alternative component for when the variables are not valid. Temporary Field can be summarized as shown below –

- Symptoms: A class has a variable that is only used in some situations.
- Detection: Comparing different methods that access each field
- Solutions: Extract Class and Introduce Null Object.
- Identifications: Medium
- Removal: Medium
- Impact: Weak

Refused Bequest smell means that a child class does not fully support all the methods or data it inherits. A bad case of this smell exists when the class is refusing to implement an interface. Keeping the Software Maintainable. Refused Bequest can be summarized as shown below –

- Symptoms: A class could not support its inherited methods or inherited data
- Detection: N/A
- Relationship: N/A
- Solutions: Replace Inheritance with Delegation
- Identifications: Medium
- Removal: Medium

- Impact: Medium

Alternative Classes with Different Interfaces smell means a case where a class can operate with two alternative classes, but the interface to these alternative classes is different. For example, a class can operate with a ball or a rectangle class, and if it operates with the ball, it calls the method of the ball class playBall() and with the rectangle it calls playRectangle().

Parallel Inheritance Hierarchies smell means a situation, where two parallel class hierarchies exist and both of these hierarchies must be extended. It is really a special case of shotgun surgery. "Every time you make a subclass of one class, you also have to make a subclass of another". The general strategy for eliminating the duplication is to make sure that instances of one hierarchy refer to instance of another. If you use *Move Method* and *Move Field*, the hierarchy on the referring class disappears. Parallel Inheritance Hierarchies can be summarized as shown below

- Symptoms: Existing parallel class hierarchies
- Detection: N/A
- Relationship: N/A
- Solutions: Move Field and Move Method
- Identifications: Medium
- Removal: Medium
- Impact: Strong

Lazy Class: Each class you create costs money and time to maintain and understand. Lazy class is a class that is not doing enough and should therefore be removed. "A class that isn't doing enough to pay for itself should be eliminated". If you have subclasses that are not doing enough try to use *Collapse Hierarchy* and nearly useless components should be subjected to *Inline Class*. This smell can be summarized as shown below –

- Symptoms: A class having little functions
- Detection: Measuring the number of fields and methods in conjunction with cyclomatic complexity.
- Relationship: Poltergeist/Lava flow
- Solutions: Collapse Hierarchy and Inline class
- Identifications: Medium
- Removal: Easy
- Impact: Weak

Data Class is a class that contains data, but hardly any logic for it. This is bad since classes should contain both data and logic. "These are classes that have fields, getting and setting methods for fields, and nothing else". These are classes that have fields, getting and setting methods, and nothing else. Such methods are dumb data holders and are manipulated in far too much detail by other classes. If in a previous life the classes were public fields, apply *Encapsulate Field*. If you have collection fields, check to see if they are properly encapsulated and apply *Encapsulate Collection* if they are not. Use *Remove Setting Method* on any field that should not be changed. Look for where these getters and setters are used by other classes and try to use *Move Method* to move behavior into the data class. If you can't move a whole method, use *Extract Method* to create a method that can be moved.

Duplicate code: According to Fowler and Beck (Fowler & Beck 2000), redundant code is the worst smell. We should remove duplicate code whenever we see it, because it means we have to do everything more than once. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them. The simplest duplicated code problem is when you have the same expression in two methods of the same class. Perform *Extract Method* and invoke the code from both places. Another common duplication problem is having the same expression in two sibling subclasses. Perform *Extract Method* in both classes then *Pull Up Field.* If you have duplicated code in two unrelated classes, consider using *Extract Class* in one class and then use the new component in the other. Duplicate code can be summarized as follows –

- Symptoms: Redundant code
- Detection: Percentage of duplicate code lines in the systems
- Relationship: Cut and Paste
- Solutions: Extract Class, Extract Method, Form Template Method, and Pull Up Method
- Identifications: Easy
- Removal: Medium

- Impact: Medium

Message Chains smell occur when you see a client that asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, etc. This smell may appear as a long line of get. This methods, or as a sequence of temps. Navigating this way means the client is structured to the structure of the navigation. The move to use in this case is *Hide Delegate* at various points in the chain. Message chains smell means a case, where a class asks an object from another object, which then asks another and so on. The problem here is that the first class will be coupled to the whole class structure. To reduce this coupling, a middle man can be used. This smell is summarized as follows:

- Symptoms: classes asking object from one to another
- Detection: Measuring the couplings of a method
- Relationship: N/A
- Solutions: Hide Delegate
- Identifications: Medium
- Removal: Medium
- Impact: Strong

Middle Man: One the major features of Objects is encapsulation. Encapsulation often comes with delegation. Middle Man smell means that a class is delegating most of its tasks to subsequent classes. Although this is a common pattern in programming, it can hinder the program, if there is too much delegation. The problem here is that every time you need to create new methods or to modify the old ones, you also have to add or modify the delegating method. Sometimes delegation can go too far. For example if you find half the methods are delegated to another class it might be time to use *Remove Middle Man* and talk to the object that really knows what is going on. If only a few methods are not doing much, use *Inline Method* to inline them into the caller. If there is additional behavior, you can use *Replace Delegation with Inheritance* to turn the middle man into a subclass of the real object. Middle Man is summarized as –

- Symptoms: A class delegating most of its tasks to subsequent classes
- Detection: Many methods coupled to one class with a low cyclomatic complexity
- Solutions: Inline Methods, Replace Delegation with Inheritance, and Remove Middleman
- Identifications: Medium
- Removal: Difficult
- Impact: Strong

Divergent Change smell means that one class needs to be continuously changed for different reasons, e.g., we have to modify the same class whenever we change a database, or add a new calculation formula. Another example is, if you have to change 4 different methods every time the database changes you might have a situation where two objects are better than one. To clean this up you identify everything that changes for a particular cause and use *Extract Class* to put them all together.

Comments are not necessarily a bad smell, but they can be misused to compensate poorly structured code. Often used as deodorant for other smells.

Dead Code: The smell has the following features –

- Symptoms: code never process at running time
- Detection: No reference to a method or a class
- Relationship: Boat Anchor
- Solutions: Collapse Hierarchy, Inline Class, Rename Method, and Remove Parameter
- Identifications: Easy
- Removal: Easy
- Impact: Medium

4. Advantages

Here we saw the bad code smells that tell the programmer when refactoring is needed. These bad code smells can be considered a measure of software maintainability, because removing them will make the software more maintainable.

5. Disadvantages

Critique on Bad Code Smells:

In the previous section, the bad code smells by Fowler and Beck (Fowler & Beck 2000) were introduced. In this section, there are some problems related to the bad code smells.

The **First problem** comes from the way the bad code smells are presented in Fowler & Beck's work. The smells are presented as a single flat list, which makes it quite difficult to get an overview of them, because the number of the smells is so high. For human mind it is quite difficult to remember 22 separate smells. The number of the smells and the way they are presented also hinders the ability to understand the smells themselves and the relationships between them. To summarize this, when presenting a concept as complex and varying as bad code smells, a single flat list with 22 entries should not be used, but instead a more hierarchical view should be provided.

The **Second critique** is directed at the comment where Fowler & Beck say that *In our experience no set of metrics rivals informed human intuition*. Here the authors wish to say that human judgment should always be the ultimate authority when assessing whether the smell spotted in the source code needs to be refactored out or not.

6. Conclusion

There are two points which can be added - First of all, that comment does not make automatic smell measurement obsolete. Automatic measurement can actually help a human to make a more informed and better decision on the smells and the possible need for refactoring. And second point on this issue is that relying strictly on human intuition might also be dangerous, because different people can have different opinions on when a smell needs refactoring. For instance, one developer can prefer really tiny methods, while the other developer might think that method should have at least 100 lines of code.

The problem with the bad code smells is that they lack empirical academic research. This final critique against the smells is the one that actually motivates research. Currently the bad smells are just concepts created by famous (and most likely talented) individuals of software engineering community, but nobody has tried to actually test and investigate the smells more thoroughly.

7. References

[1] FOWLER, M., BECK, K., BRANT, J., OPDYKE, W. & ROBERTS, D. (1999) Refactoring: Improving the Design of Existing Code, Addison Wesley. ZHANG, M., HALL, T., WERNICK, P. & BADDOO, N. (2008).Code Bad Smells: A Review of Current Knowledge. Techincal Report No. 468, Hatfield, STRI, University of *Hertfordshire.*

[2] FOWLER, MARTIN: A list of refactoring tools for several languages, http://www.refactoring.com/tools.html

[3] Source: Martin Fowler, *Refactoring - Improving the Design of Existing Code.* Addison Wesley, 1999.

[4] http://books.google.co.in/books?id=1MsETFPD3I0C&dq=bad+smell+coding+in+software+refactoring+by+martin+fowler+%26+kent+beck&printsec=frontcover&source=in&hl=en&ei=nrVHTO-JDIimvQOcxc3LAg&sa=X&oi=book_result&ct=result

[5] Refactoring : Improving the Design of existing Code :- Martin Fowler, Object Technology international, Inc , addison villey , Pearson education.

[6] A quantitative evaluation of maintainability enhancement by Refactoring .Yoshio kataoka ,Tetsuji fukaya Proceedings of the international conference on software Maintenance(ICSM'02)IEEE

[7] Refactoring :- M.J.Munro , Technical Report,EFoCS-56-2005,Dept.of computer & Information science ,University of strathclyde ,December 2002.

[8] HELSINKI UNIVERSITY OF TECHNOLOGY, Department of Computer Science and Engineering, Software Business and Engineering Institute-Mika Mäntylä - Bad Smells in Software – a Taxonomy and an Empirical Study