Evaluation of code smells and Refactoring

Sharath Badam

Abstract:

Refactoring, one of the methods to keep software maintainable, is covered in the paper. However, if we do not recognize the appropriate times to apply refactoring, refactoring will not yield its full benefits. Fowler & Beck (Fowler & Beck 2000) provide a list of bad code smells to help software developers determine whether a piece of software need refactoring or not. According to Fowler & Beck, unpleasant code smells serve as a more direct indicator of the need for refactoring than some notion of programming aesthetics. Fowler & Beck assert that it is impossible to determine whether refactoring is necessary using a specific set of metrics. As a result, bad code smells represent a sort of compromise between exact source code metrics and vague programming aesthetics.

For instance, the size of a class might be represented by a single characteristic, with the existing smell being referred to as Large Class on one end and Lazy Class on the other. Fowler also introduces a collection of refactoring (move methods, inline temp, etc.) with step-by-step instructions on how each undesirable code smell can be eliminated. The reader should therefore be aware that the refactoring concept also provides comprehensive guidelines for actually improving the source code. The 22 offensive code smells are explained in the next section with the aim of introducing them.

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

1. Introduction

Refactoring is an extremely new field of study, and as such, it lacks a clear definition. There are many definitions of refactoring, the majority of which are listed below:

1. It is a modification made to software's underlying structure to make it simpler to comprehend and less expensive to modify without altering its discernible behavior.
2. To perform a sequence of refactoring to software in order to restructure it without altering its observable behavior.
3. Refactoring is the process of changing an object design in many ways to make it more adaptable and/or reusable. Efficiency and maintainability are two of the many reasons you could want to do this, with those two factors likely being the most significant.
4. Refactoring software code entails rewriting it and "cleaning it up."
5. Refactoring is the process of moving functional units around in your software.
6. Getting every piece of functionality to reside in exactly one place in the software is the main goal of refactoring.

2. Related work

Bad Smells in Code

Finding areas of poor design in a system and knowing where to refactor can be difficult. Code refers to these problematic design elements as "Bad Smells" or "Stinks." These

locations can be found more by "human intuition" than through precise research. These "Bad Smells" are recognized using the developer's experience. However, if we do not recognize the appropriate times to apply refactoring, refactoring will not yield its full benefits. Fowler & Beck (Fowler & Beck 2000) provide a list of bad code smells to help software developers determine whether a piece of software need refactoring or not. Any sign that something is amiss is a "code smell." The phrase became more common once it was used in Refactoring. Refactoring can be used to address some of the qualities that bad code demonstrates. We refer to these as Bad Smells.

Types of Smell

There are 18 bad code smells and few are introduced in this part for the following reasons:

A **long method** is one that is challenging to comprehend, modify, or extend. Short methods are a principle that Fowler and Beck (Fowler & Beck 2000) firmly support. A procedure gets harder to understand the longer it is. Almost always, all you need to do to condense a method is to use Extract Method. If you attempt to utilize the Extract Method and find that you are passing a large number of parameters, you may frequently use Replace Temp with Query to get rid of the temps and condense the long list of parameters by using Introduce Parameter Object and Preserve Whole Object. Long Method can be condensed into:

Symptoms: A long method that is challenging to comprehend and reuse

Cyclomatic complexity is detected (polynomial metrics)

No correlation

Solutions: Replace Temp with Query, Decompose Conditional, Extract Method, Replace Method with Method Object, and

Identifications: Medium

Removal: Difficult

Impact: massive

A **large class** is a sign that it is trying to attempt too much. Due to the excessive number of instance variables or methods in these classes, duplicate code is inevitable. Duplication thrives in classes with excessive amounts of code. Both Extract Class and Extract Subclass will function in these scenarios. The smell is best described as follows:

Symptoms: A surplus of instance variables or methods

Lack of Cohesion Methods or Measurement Class Size

Relationship: Blob/God Object

Solutions: Extraction of Classes, Interfaces, Subclasses, and Inclusion of Foreign Method

A situation when primitives are employed instead of small classes is represented by the smell of **primitive obsession**. To represent money, for instance, programmers utilise primitives rather than developing a unique class that could have the essential features, such as currency conversion. The traits of Primitive Obsession are as follows:

Symptoms: utilising primitives as opposed to small classes

Observing: N/A

No connection

Solutions: extract Class, Add Parameter Object, Substitute Object for Array, Data Value for Object, and Type Code for Subclass/State/Strategy

Identifications: Moderate Removal: Moderate

Impact: large

Long parameter lists are too long and hence challenging to comprehend. With objects, you only need to put in enough so that the method can access all it needs, rather than providing everything the method requires. This is fortunate because long parameter lists are confusing, inconsistent, and challenging to use because they are always being altered as new data becomes available. When you can obtain the data in one parameter by making a request of an object you are already familiar with, use Replace Parameter with Method. This smell can be summed up as

Symptoms: An overly complicated procedure with several parameters

detection: count up the parameters.

Solutions: Add a parameter object, swap out a method for a method object, and keep the whole object. Medium Identification

Removal: Moderate; impact: Moderate

Software with a "**Data Clumps** smell" has data elements that frequently occur together. Frequently, the same three or four data points will appear together in many places:

Fields in a few classes.

b) Multiple method signatures contain parameters:

When one of the group's data items is removed, the other elements, such as numbers specifying RGB colors, become meaningless. The following characteristics of data clumps

The data are always consistent with one another.

Detection: The data collection loses all meaning if one value is deleted.

Relationship: Magic String/Magic Numbers

Identifications: Medium

Solutions: Extract Class, Introduce Parameter Object, and Preserve Whole Object

Removal: challenging

Impact: Moderate

Since a switch operator does not always imply a smell, the term "**Switch Statements Smell**" may be slightly misleading. The usage of type codes or runtime class type detection in place of polymorphism is referred to as the "smell." This smell also manifests itself in type codes that are provided to methods. When you read a switch statement, you should usually take polymorphism into account. To get the switch statement into the class where the polymorphism is required, use the Move Method after using the Extract Method to get the switch statement out. Polymorphism is overkill if there are only a few circumstances that have an impact on one particular approach. Replace Parameter with Explicit Methods is a solid choice in this situation. Try introducing a null object in one of your conditional circumstances.

Switch Statements are outlined as follows:

Symptoms: Type codes or runtime class type detection to replace polymorphism

Detection: runtime detection

No connection

Solutions: Introduce Null Object, Polymorphism in place of Conditional, Explicit Method in place of Type Code, and Subclass/State/Strategy in place of Type Code

Identifications: Medium

Removal: Moderate

Impact: Minimal

Temporary Field smell denotes the presence of a variable in the class that is only used occasionally. You may occasionally come into an entity that only uses an instance variable under specific conditions. Because we typically anticipate an object to use all of its variables, this might make the code challenging to comprehend. By including all the code that use the variable in the component, use Extract Class to give these orphan variables a home. By utilising Introduce Null Object to construct a backup component for when the variables are invalid, you can also get rid of conditional code. The summary of Temporary Field is as follows:

Symptoms: A variable in a class is only ever utilised in certain circumstances.

Detection: Evaluation of different methods that access each field

Solutions: Introduce Null Object and extract the Class.

Identifications: Medium

Removal: Moderate

Impact: Minimal

A child class that emits the **Refused Bequest** smell does not completely support all the methods or data it inherits. When the class refuses to implement an interface, there is a severe case of this smell. Ensuring the Software's Maintainability, The Refused Bequest can be summed up as follows:

Symptoms: It was impossible for a class to support its inherited data or methods.

The phrase "**Alternative Classes with Different Interfaces Smell**" refers to situations where a class can work with two different classes but their interfaces are different. If a class wants to operate with a ball, it calls the playBall() method of the ball class, and if it wants to operate with a rectangle, it calls the playRectangle method ().

Inheritance in parallel When two concurrent class hierarchies must be extended, the condition is referred to as a **hierarchy smell**. In reality, it is a unique instance of shotgun surgery. “Every time you make a subclass of one class, you also have to make a subclass of another”.

3. Background

Any symptom in a program's source code known as a "code smell" could point to a more serious issue. They are signs of subpar design and implementation decisions, which may hamper further software system evolution, maintenance, and development. Code smells can be eliminated once they have been identified in a system by refactoring the source code. However, human inspection is cumbersome and unreliable, and detection in big software systems is a resource- and time-intensive, error-prone operation.

Developers are assisted in identifying "smelly" entities by tools for automatic or semi-automatic code smell detection. The tools can highlight the entities that are more likely to present code smells thanks to the use of detecting algorithms. For finding code smells, there are fortunately lots of software analysis techniques accessible. In general, this fact shows that the software engineering community is aware of the significance of regulating the structural quality of features that are being developed. On the other hand, it presents a fresh problem for how to evaluate, compare, and choose the best tool for a given development setting.

In truth, there are several issues with evaluating how well code smell detection technologies work. For instance, numerous interpretations of same code smell result from the ambiguity and occasionally vagueness of definitions. Due to the lack of precise definitions, tools are required to use several detection methods for the same code smell. Additionally, because these methods typically rely on the computation of a specific set of combined metrics, ranging from

conventional object-oriented metrics to metrics constructed in ad hoc ways for the goal of smell detection, they produce a variety of results.

Finally, even if the same metrics are employed, the threshold values may differ since they are determined by taking into account many variables, including the size of the system domain, organizational procedures, and the expertise of the software engineers who design them. The amount of detected code smells is significantly affected by changing the thresholds.

It might be challenging to comprehend the outcomes of various methodologies when applied to the aforementioned issues. Indeed, early research (Mäntylä 2005; Moha et al. 2010; Murphy-Hill and Black 2010) attempted to solve some of these issues, but only for small systems and minimal code smells. For instance, Fontana et al. (2012) looked into six code smells in the GanttProject software system. Their research demonstrated that a deeper comprehension of the system and its code smells is necessary for a more accurate evaluation of code scent detection approaches. The trouble is not only in the various ways that code smells are interpreted, but it's also in manually identifying the code smells. Therefore, it is difficult to find open-source systems with validated lists of code smells to serve as a basis for evaluating the techniques and tools. The goal of the paper is to improve our understanding of code smells and the tools used to identify them. compared three detection tools—JDeodorant, PMD, and inFusion—in a software system. Additionally, we conducted a comparison of tools for two code smells: large classes and long methods.

Code smells and detection tools

In Fowler's book (Fowler 1999), Kent Beck introduced code smells as a way to identify symptoms that might point to a problem with the system code. They refer to any symptom in a program's source code that can point to a bigger issue. Code smell detection tools can assist programmers in maintaining the quality of their software by using a variety of methods for detecting code smells, including object-oriented metrics and program slicing.

Code smell definitions

Code smells are snippets of code that hint at potential refactoring. Fowler (1999) first listed 22 code smells along with suggested refactoring. The literature has also suggested other smells, like Swiss Army Knife and Spaghetti Code. Refactoring is the process of making changes to a software system that do not affect the code's external behavior while enhancing its internal structure (Fowler 1999). The existence of code smells is a sign that refactoring is required. Code smells can reduce the software system's quality in areas like maintainability and comprehensibility, making it more challenging to understand and, therefore, maintain. This slows down the software development process.

Three code smells—God Class, God Method, and Feature Envy—are the main topics of this paper. God Class defines a class that unifies the system's functions. "A class that knows or does too much," in other words. In other words, a God Class goes against the object-oriented programming principle that says a class should only have one responsibility, as stated by

DeMarco in 1979. When a method is given an excessive amount of functionality, it becomes unmanageable and challenging to maintain and expand (Fowler 1999). Therefore, God Method tends to centralize a class's functions in the same manner as God Classes prefer to do the same for a subsystem or even an entire system's functionalities.

The term "feature envy" describes a method that appears more interested in the features of a different class than the one it actually belongs to (Fowler 1999). These bits of code make direct or indirect access to various pieces of data belonging to other classes. It can mean that the method is incorrectly placed and has to be moved to a different class. These three code smells were chosen because they are the most common code smells in the target systems, they can be detected without the aid of tools, which is necessary to compile the code smell reference list, (ii) they can also be detected without the aid of tools, allowing us to compare the tools recall, precision, and agreement, and (iii) they are among the most frequently detected code smells by code smell detection tools.

Detection tools

In this study, four code smell detection tools—inFusion, JDeodorant, PMD, and JSpIRIT—that may be downloaded for free or with a trial version are assessed. These tools were chosen because they analyse Java programs, they can be set up and installed using the downloaded files that were provided, they find the analysed smells in both target systems, and they produce a list of instances of code smells that can be used to calculate recall, precision, and agreement. Other tools were thrown away for various reasons. For instance, Stench Blossom was disqualified because it did not offer a specific list of code smells, while Checkstyle was disqualified because it failed to find any instances of smells in any of the target systems. Instead, it just offers features for visualisation.

God Class, God Method, and Feature Envy are three of the 22 code smells that inFusion, a commercial standalone tool for Java, C, and C++, finds. InFusion is no longer accessible for download as a commercial product. Nevertheless, the tool's iPlasma open source variant is still accessible. The detection methodologies established by Lanza and Marinescu (2006) served as the foundation for all smell detection techniques, which were further improved using source code from a variety of open source and for-profit systems. Four code smells are identified by the Java Eclipse plugin JDeodorant Footnote2: God Class, God Method, Feature Envy, and Switch Statement.

The detection methods are founded on the recognition of refactoring possibilities of Extract Class for God Class, Extract Method for God Method, and Move Method for Feature Envy. Two of the code smells of importance to us, God Class and God Method, are both detected by the open source Java tool and Eclipse plugin PMD Footnote3. Metrics are the foundation of the detection techniques. God Method uses just one metric, LOC, while God Class relies on the detection mechanisms outlined by Lanza and Marinescu (2006). Last but not least, the Java Eclipse plugin JSpIRIT Footnote4 recognizes and ranks ten code smells, including the three that are of particular relevance to us: God Class, God Method, and Feature Envy.

Tool	**Version**	**Type**	**Languages**	**Refactoring**	**Export**	**Detection Techniques**
inFusion	1.8.6 2015	Standalone	Java, C, C++	No	Yes	Software Metrics
JDeodorant	5.0.0 2015	Eclipse Plugin	Java	Yes	Yes	Refactoring opportunities
PMD	5.3.0 2015	Eclipse Plugin	Java, C, C++ and others	No	No	Software Metrics
JSpIRIT	1.0.0 2014	Eclipse Plugin	Java	No	No	Software Metrics

Table 1

The essential details of the tools under evaluation are compiled in Table 1. The names of the studied tools, as provided on the websites for those tools, are included in the column Tool. The version of the tools used in the experiments is indicated in the column Version. If a tool is a standalone application or a plugin for the Eclipse IDE, it is indicated in the column Type. The programming languages in the source code that can be analysed by the tools are listed in the column Languages, with Java serving as their common language.

The Refactoring column indicates whether the tool has the JDeodorant-exclusive ability to refactor detected code smells. Only inFusion and JDeodorant have the feature of exporting the results to a file, which exports the results as an XML file and an HTML file, respectively. The column Export indicates whether the tool permits exporting the results to a file. The last section, Detection Techniques, provides an overview of the methods each instrument employs, with software metrics being the most popular.

Methodology

Research Questions

Four code smells detection tools—inFusion, JDeodorant, JSpIRIT, and PMD—are examined and compared in this study. The aim is to assess the accuracy and consistency of these tools in identifying code smells by providing answers to two study objectives. The following presents and discusses research-related concerns.

RQ1. How effective is each tool at pointing out pertinent code smells?

Our objective is to assess the effectiveness of each tool in identifying code smells that have already been recognized by experts and included to each system's code smells reference list. To assess the precision and recall of each tool in identifying God Class, God Method, and

Feature Envy, the output from all tools evaluating the source code of both systems was compared against a reference list.

RQ2. Do many detection methods identify the same code smell?

Different tools use various detecting methodologies. various thresholds as a result, several tools are anticipated to recognize various classes and functions as code smells. When classifying a class or method as a code smell, our objective is to assess how well the tools match.

Experimental Results

This study evaluates code smell detection tools and compares their code smell detection accuracy and concordance to the same system. Although we rely on recall and precision to measure accuracy, agreement is measured by calculating overall agreement and AC1 statistics.

Project	# of Classes	# of Methods	LOC
1	24	124	1159
2	25	143	1316
3	25	143	1364
4	30	161	1559
5	37	202	2056
6	46	238	2511
7	50	269	3015
8	50	273	3167
9	55	290	3216
10	n/a	n/a	n/a
11	77	401	5996
12	80	424	6369
13	80	424	6369
14	92	566	7046
15	93	581	7231
16	97	406	7293
17	99	606	7316
18	103	611	7355
19	115	659	8800
20	118	671	8702
21	81	425	6350
22	83	432	6421
23	85	451	6608
24	87	467	6690
25	89	480	6700
26	90	505	6900
27	78	420	6010
28	80	424	6369
29	82	450	6615
30	85	451	6608

Table 2: Metrics for 30 Subject Programs having bad smells

Recall and Precision

We took into account true positives, which are instances (classes or methods) existing in the code smell reference list and also identified by the tool being evaluated, for calculating recall and precision. False positives are instances that the tool reported but were not in the reference list. False negatives are occurrences in the reference list that the tool did not flag as positive.

True negatives, on the other hand, are occurrences that neither appear in the reference list nor were detected by the tool. Recall is calculated by dividing the number of true positives by the instances in the reference list (true positives plus false negatives), whereas precision is calculated by dividing the number of true positives by the instances reported by the tool (true positives plus false positives).

Overall agreement

The proportion of examples (classes or procedures) that were categorized in the same category by both tools, for the overall agreement between pairs, or by all tools, in the case of overall agreement between many tools, is known as the overall agreement or percentage agreement (Hartmann, 1977). There are two categories that can be applied to each instance of a target system: smelly or non-smelly. As a result, the overall agreement is calculated by dividing the total number of instances in the target system by the number of instances categorized by the pair or set of tools as belonging to the same category (smelly and non-smelly). The total number of God Class instances is correlated with the system's total number of classes, whereas the total number of God Method and Feature Envy instances is correlated with the system's total number of methods.

AC1 statistic

AC1 statistic (Gwet 2001), or first-order agreement factor, adjusts the overall percent agreement of concordant agreements. In other words, by classifying the same case as smelly or not smelly at the same time, chances of tool matching are minimized. The AC1 statistic is therefore the relative number of instances the tool would expect to match from a set that has already had instances that happen to be in the same category removed. The AC1 statistic is computed as $(p - e(\gamma))/(1 - e(\gamma))$. where p is the overall match rate and $e(\gamma)$ is the random match probability (Gwet 2001). A complete calculation and explanation of $e(\gamma)$ can be found in the book by Gwet (2001).

Code Smell	**inFusion**		Jdeodarant		Jspirit		PMD		Reference List	
	Project 1	Project 2	Project 1	Project 2	Project 1	Project 2	Project 1	Project 2	Project 1	Project 2
God Class	3	0	85	98	9	20	8	33	47	12
God Method	17	0	100	599	27	30	16	13	67	60
Feature Envy	8	48	69	90	74	111	–	–	19	0
Total	28	48	254	787	110	161	24	46	133	72

Table3

Comparative study of code smell detection tools

Table 3 shows the total number of code smells identified by each tool in Project1, Project2

Project1:

JDeodorant reports the most classes for God Class (85), the reference list includes 47 classes, and the other tools report fewer than 9 classes. JDeodorant reports 100 techniques for the God Method, compared to 67 methods in the reference list. However, whereas PMD and inFusion both list 16 and 17, respectively, JSpIRIT lists 27 God Methods. The biggest number of methods reported for Feature Envy is 74 by JSpIRIT, followed by 69 by JDeodorant, and 19 by the reference list. Last but not least, inFusion only lists 9 cases of Feature Envy. In terms of the total number of reported smells, inFusion and PMD report comparable totals. However, with a total of 28 instances of God Class, God Method, and Feature Envy, inFusion is the most conservative tool. Compared to inFusion's detection of 20, PMD is less cautious, finding a total of 24 instances of God Class and God Method. The PMD doesn't pick up on Feature Envy. In comparison to the earlier tools, JSpIRIT and JDeodorant report more smells. By reporting 254 cases, JDeodorant is by far the most aggressive in its detection techniques. In other words, JDeodorant detects more than nine times as many odours as the two most traditional tools, PMD and inFusion. The tool that reports a total number of code smells, however, that is closer to the actual number of 133 instances of the reference list, is JSpIRIT.

CONCLUSIONS AND FUTURE WORK

Comparing smell detection tools in the code is a difficult task because these tools are based on informal definitions of smell. Differences in the interpretation of code smells by researchers and developers lead to different detection techniques, results, and consequently tools with the time required for verification. This article evaluates the accuracy and consistency of the inFusion, JDeodorant, JSpIRIT, and PMD tools. Accuracy was measured by calculating the recall and precision of the tool when detecting code smells from a reference list. Agreement was measured by calculating overall agreement and AC1 statistics, considering all tools simultaneously and between tool pairs. Given the accuracy of the tool, we found a trade-off between the number of correctly identified entities and the time spent on validation. For all smell in both systems, JDeodorant identified the most correct entities, but reported many false positives. Lower precision and higher recall mean more validation effort, but capture most of the affected entities. On the other hand, inFusion, JSpIRIT, and PMD were more accurate and reported more accurate instances of smelly entities. A low capture rate but high accuracy means that the tool does not report all of the affected entities. As the system evolves, it can become more complex and difficult to modify smells. Therefore, smell should be detected as early as possible to facilitate refactoring activities. In this case, using a more recognizable tool will give you better results. We also analyzed tool agreement by calculating overall agreement and chance-corrected agreement using AC1 statistics for all tools and tool pairs.

References:

[1] W. Welf Löwe, T. Panas, Rapid construction of software comprehension tools, International Journal of Software Engineering and Knowledge Engineering 15 (06) (2005) 995–1025.

[2] A. Telea, L. Voinea, Visual software analytics for the build optimization of large-scale software systems, Computational Statistics 26 (4) (2011) 635–654.

[3] B. F. Webster, Pitfalls of object oriented development, M and T Books, New York, NY, USA, 1995.

[4] W. H. Brown, R. C. Malveau, H. W. M. III, T. J. Mowbray, AntiPatterns: Refactoring software, architectures, and projects in crisis, John Wiley and Sons, Inc, Canada, 1998.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, AddisonWesley, 1999.

[6] M. Misbhauddin, M. Alshayeb, UML model refactoring: a systematic literature review, Empirical Software Engineering 20 (1) (2015) 206–251. doi:10.1007/s10664-013-9283-7.

[7] V. Garousi, M. V. Mäntylä, A systematic literature review of literature reviews in software testing, Information and Software Technology 80 (2016) 195 – 216.

[8] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola,F. Shull, C. Seaman, Identification and management of technical debt: A systematic mapping study, Information and Software Technology 70 (2016) 100–121. doi:10.1016/j.infsof. 2015.10.008.

[9] T. Besker, A. Martini, J. Bosch, Managing architectural technical debt: A unified model and systematic literature review, Journal of Systems and Software 135 (2018) 1–16. doi:10.1016/j.jss.2017.09.025.

[10] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, Journal of Systems and Software 101 (2015) 193–220. doi:10.1016/j.jss.2014.12.