Object Oriented Source Code Bad Smells Detection : A Critique

Abdullah Al-Shishani
Hashemite University, Jordan
Department of Software Engineering
abdullah.asendarz@gmail.com

***Abstract*—Software bad smells detection is very important in any software project, because it affects maintainability on the long run. This paper reviews bad smells identification and detection in object oriented software. An overview of current bad smells identification and detection tools and approaches of object oriented systems are presented and a critique is done. The focus is on object oriented systems because almost all of the programming languages are heading forward to object oriented methodologies.**

Keywords*—Software bad smells, bad smells detection, object oriented, bad smells critique.

I. Introduction

Software maintainability is one of the most important non-functional requirements in any software project, because maintenance typically consumes 40 to 80% (60% average) of software costs. On of the factors that affect maintainability is bad smells. The maintainability of any software can be effected by bad smells, because it makes the code more complex; making the act of maintaining more harder and costy. Thus, detecting bad smells in any software project can be really promising.

Code smells, as defined by Fowler [3] are symptoms of poor design and implementation choices. These symptoms, in some cases, may be the result of some activities performed by developers in a hurry *(e.g., implementing urgent features or some critical bugs hot-fixes)*.

Many previous studies and research have been done on bad smells identification and detection, and lots of bad smells identifications exist. However, in this paper, 3 papers that focus on bad smells and design defects are reviewed and a critique is done.

A. Detecting Bad Smells in Source Code Using Change History Information

First paper to review is Palomba et al. [6] "Detecting Bad Smells in Source Code Using Change History Information" in 2013. Palomba et al. presented a tool and approach to detect bad smells identified by Fowler [3] and Brown using change history information extracted from versioning systems.

Palomba et al. and based on change history information of the source code proposed an approach called HIST (**H**istorical **I**nformation for **S**mell de**T**ection). HIST can detect five smells identified by Fowler [3] and Brown[1], These smells are Divergent Change Shotgun Surgery, Parallel Inheritance, Blob and Feature Envy. It is good to note that Blob and Feature Envy can be detected using static detection approaches that already exist [5] (without making use of change history information).

B. Java Quality Assurance by Detecting Code Smells

The second paper to review is Eva van Emden et al. [2] "Java Quality Assurance by Detecting Code Smells" in 2002. Eva van Emden et al. presented an approach for the automatic detection and visualization of code smells. However, our focus will be on the bad smells detection rather than focusing on the visualization.

Eva van Emden et al. used static approach for bad smells detection. Characterizing each bad smell by it's aspects, thus, if the source code has all the aspects described, we say it has a bad smell of that particular type.

Eva van Emden et al. focused on many bad smells presented by Fowler, that can be categorized to simple and complex. Simple bad smells are bad smells that everyone discourages *(e.g. code duplication and long methods)*. Complex bad smells are bad smells that originate from object oriented design issues *(e.g. parallel inheritance hierarchies and message chains)*. Eva van Emden et al. noted that the list of source code bad smells can never be complete, as in each project's domain a different set of bad smells should be considered.

However, one should note that some of the bad smells can not be detected using a static approach, thus, Eva van Emden et al. proposed method can not detect these bad smells as they use a static approach.

C. Automatic Generation of Detection Algorithms for Design Defects

The third paper to review is Naouel Moha et al. [4] "Automatic Generation of Detection Algorithms for Design Defects" in 2006. Naouel Moha et al. presented an approach, a language and a framework to detect design defects. The presented approach tends to detect design defect on two levels, High Level and Low Level. However, they focus on the high level *(e.g, Spaghetti Code, Blob, .. etc)*.

Naouel Moha et al. presented an approach to detect design defects (which are considered bad smells on the low level) by generating detection algorithms. They define a systematic method to specify high-level design defects and to generate detection and correction algorithms from the specifications of the design defects.

It is more precise detecting bad smells especially on high level using algorithms generated using Naouel Moha et al. approach.

The remainder of this paper is organised as follows: Section 2 summarizes all the three researches mentioned; in section 3 a critical evaluation is made; section 4 describes conclusion.

II. Summary

In this section a summary of each one of the researches is presented, explaining the key points of each research.

A. Detecting Bad Smells in Source Code Using Change History Information

Even though there are many approaches to detect bad smells, most of these approaches rely on the structural information extracted from the code. Thus, these approaches can not detect many of bad smells described by Fowler [3]. One good example is detecting parallel inheritance, "Parallel Inheritance means that two or more class hierarchies evolve by adding code to both classes at the same time"[6], which can not be detected with a static analysis.

Palomba et al. approach uses change history information of the source code proposed to detect bad smells as the static analysis may be not useful for detecting some bad smells. The approach is called HIST.

HIST is used and evaluated on four bad smells (i.e, Divergent Change,Shotgun Surgery,Parallel Inheritance and Blob). These smells are not randomly selected, it is because of the need to have a benchmark including smells that can be identified using change history information and smells that do not necessarily require this type of information. The steps to detect bad smells using HIST :

Change History Extraction : HIST first extracts information needed to detect the bad smells from the versioning system *(e.g, SVN, CVS, or git)* using a component called *Change history extractor*. However, the logs extracted through this operation report code changes only at file level, which is not sufficient to detect some bad smells described, because many of them describe method-level behaviour *(e.g, Feature Envy and Divergent Change)*. Thus, the Change history extractor includes a code analyzer to capture changes at method-level. Using the code analyser and the extracted logs, the set of changes extracted from the versioning system includes:

- Classes added/removed/moved/renamed
- Class attributes added/removed/moved/renamed
- Methods added/removed/moved/renamed
- Changes applied to all the method signatures
- Changes applied to all the method bodies

using these extracted logs and changes, it is possible to detect bad smells that have method-level behaviour sufficiently.

Code Smells Detection : after extracting needed fine logs from the versioning system, HIST passes these information to the *Code Smell Detector*. One should note that HIST uses a custom heuristic for each bad smell to detect. As HIST relies on the change history information, it is possible that the system under examination had some of the bad smells in the past but do not anymore *(e.g., it has been refactored by the developers)*. Thus, HIST omits these bad smells as they do not exist anymore in the current version of the system.

As mentioned, HIST uses custom heuristics for each bad smell, In the following the heuristics to detect different kinds of bad smells are described:

- Divergent Change Detection : Divergent Change conjecture is as defined as *classes affected by Divergent Change that present different sets of methods each one containing methods changing together but independently from methods* [3] in the other sets. To detect Divergent Change, the dataset is composed of a sequence of change sets *(e.g., methodsthat have been committed (changed) together in a version control repository)*.
- Shotgun Surgery Detection : Shotgun Surgery conjecture is defined as *a class affected by Shotgun Surgery contains at least one method changing together with several other methods contained in other classes* [6] to detect Shotgun Surgery *Code Smell detector* looks-up for methods in deffirent classes that are changing together.
- Parallel Inheritance Detection : Parallel Inheritance conjecture is defined as *every time you make a subclass of one class, you also have to make a subclass of the other* [3] to detect Parallel Inheritance *Code Smell detector* looks-up for different classes that adds subclass each time that some other class adds a subclass in change history logs
- Blob Detection : Blob conjecture is defined as *a class that centralizes most of the systems behavior and has dependencies towards data classes* [1] to detect Blob *Code Smell detector* looks-up for classes that needs to be changed most-likely when a change is made on different artifacts, because that class is probably a *God Class* which has loosely coupling with different artifacts.
- Feature Envy Detection : Feature Envy conjecture is defined as *a method affected by feature envy changes more often with the envied class than with the class it is actually in* [6] to detect Feature Envy *Code Smell detector* looks-up for methods involved in commits with methods of another class of the system more than in commits with methods of their class.

It is good to note that using change history information to detect bad smells can be the optimal approach in some cases *(e.g, Parallel Inheritance Detection)*, but in other cases it may be better to use a static approach instead *(e.g, Blob)*

B. Java Quality Assurance by Detecting Code Smells

Eva van Emden et al. used static approach for bad smells detection. Basically characterizing each bad smell by it's aspects, thus, if the source code has all the aspects described, we say it has a bad smell of that particular type. However, these aspects are distinguished in to two types :

- Primitive Aspects (can be observed directly in the source code)
- Derived Aspects (inferred from other aspects)

example of a Primitive Aspect is a method that contains a switch statement, which can be observed directly in the source

code. And example of a Derived Aspect is a class that is not using any of it's super class methods.

The process of detecting bad smells using Eva van Emden et al. approach is separated into the following steps:

1) Find all entities of interest in the code.
2) Inspect them for primitive smell aspects.
3) Store information about entities and primitive smell aspects in a repository.
4) Infer derived smell aspects from the repository

the approach is straight forward. However, it is not possible to detect some bad smells, using this static approach, that depends on the change history information *(e.g, Parallel Inheritance)*.

C. Automatic Generation of Detection Algorithms for Design Defects

To specify design defects Naouel Moha et al. use *Rule Cards*, which describes a particular design defect. After generating a rule card for a design defect, the rule card is parsed, then a model representing that particular rule card is built and finally the detection algorithm is generated. The generated algorithm is used to detect design defects, and the results are validated.

The detection of design defects is done by defining these defects synthetically and by generating detection algorithms automatically. The detection of design defects are enhanced with structural and semantic properties to increase precision in design defects detection. Using a method to describe design defects presented by Naouel Moha, the first language is built to specify the design defects in their basic characteristics using rule cards, such as metrics, structural relationships, and semantic and structural properties. Then, using this language and their framework dedicated to the analysis of programs,detection algorithms are generated automatically from the specifications of design defects.

The focus was on rule cards to generate design defects detection algorithms. A rule card describes a design defect, its code smells and the relationships among code smells. Rule cards are formalised with a BNF grammar, which determines the exact syntax for a language.

What makes this approach special is automatic generation of design pattern detection algorithms and being more precise.

III. Critical Evaluation

The base on evaluating approaches under study is how good are these approaches in detecting bad smells and the covered bad smells in the detection process. The basic evaluation points are:

- Approach logic.
- Bad smells that the approach can detect
- Level of automation
- Accuracy

All of approaches are evaluated and tested on high quality real life evolving systems *(e.g, Apache Ant, Apache Tomcat, Apache Xerces, Apache Lucene, JEdit, Log4J)*.

A. Detecting Bad Smells in Source Code Using Change History Information

The usage of change history information in detecting bad smells is very helpful in many cases and detecting some bad smells *(e.g, Parallel Inheritance)*, which can be undetectable or at least inaccurate in many static bad smell detection approaches. However, for detecting some other bad smells *(e.g, Blob)* static approaches make more sense, but still doable using change history information. In many cases, a bad smell is refactored because it may cause troubles in the future, thus, these bad smells that do not exist in the code anymore are omitted. A drawback is having to define custom logic for each bad smell to detect manually. To sum up, the idea of using change history information is really helpful in some cases and does not make sense in many others.

The focus of this approach was mainly on bad smells that can be detected using change history information such as Parallel Inheritance, Divergent Change, Blob and Shotgun Surgery.

The approach is semi-automated, as the *Change history extractor* refines the logs to extract the information before passing these logs to *Code Smell Detector*, which is responsible for bad smells detection process.However, defining logic for each bad smell to detect is manual.

Evaluation and testing done on this approach showed precision between 61% and 80%, and its recall is between 61% and 100%.

(-EXTRA- i personally liked this approach, even though it does not make much sense in some cases, like detecting blob ? as explained it does make sense using change history information, but the way i see it, it is not necessary to use change history information to detect bad smells such as blob)

B. Java Quality Assurance by Detecting Code Smells

The act of detecting bad smells by searching for the artifacts of the bad smell under study is static and straight forward approach, basically searching for primitive smell aspects that can be observed directly in the code *(e.g, Switch Statement)*, and derived smell aspects that are inferred from other aspects *(e.g, a class that do not use any methods offered by its superclasses)* . This approach can be useless in detecting some bad smells *(e.g, Parallel Inheritance)*, it is more focused on more simple bad smells *(e.g, Switch Statements, Code Duplicate)*. To sum up, it is not efficient to detect bad smells by searching for their artifacts with complex bad smells.

It is good to note as this approach is simple at some level, it has a high performance. The extraction of this approach on CHARTOON system that consists of 46000 LOC (without comments or empty lines) and 147 classes took roughly 30 seconds on a computer with an AMD Athlon processor (1.2 Ghz) and 512 Mb main memory running linux 2.4.9-12.

The focus of this approach is on bad smells that are less complex and can be detected using static analysis such as Instanceof, Typecast and Data Classes.

The approach is automated, as it depends on the static analysis of finding bad smell's primitive and derived artifacts.

It is obvious that searching for bad smells in source code by looking up for its artifacts can be precise.

(-EXTRA- i loved the idea and the simplicity of this approach, but it is useless with complex bad smells which occurs a lot in the source code,, so it is not as good as other approaches)

C. Automatic Generation of Detection Algorithms for Design Defects

The act of detecting source code design defects and bad smells using automatically generated algorithms is really precise and efficient. Design defects are specified synthetically using rule cards, which are the description of the design defects. This approach can detect complex bad smells or design defects *(e.g, Design AntiPatterns, Blob)*. Spaghetti Code *(i.e, procedural thinking in object-oriented programming)* is also considered, which gives a plus for this approach. To sum up, it is efficient to detect bad smells and design defects using automatically generated algorithms.

The focus of this approach is on complex bad smells and design defects such as Spaghetti Code, Swiss Army Knife and Functional Decomposition.

The approach is automated, as the algorithms are generated automatically.

Evaluation and testing done on this approach showed precision of 64%.

(-EXTRA- i didnt understand this approach very well because i was new to all that approach and stuff they used in it, however, i found it really tempting and couldnt resist reading it. As it looked at bad smells from "design pattern" point of view, and these problems always occur in variant projects)

IV. Conclusion

This paper reviewed three papers: Detecting Bad Smells in Source Code Using Change History Information [6], Java Quality Assurance by Detecting Code Smells [2], and Automatic Generation of Detection Algorithms for Design Defects [4], for detecting bad smells and design defects. The overall evaluation of the approaches under study were as follows:

For the first approach, Detecting Bad Smells in Source Code Using Change History Information, we found that using change history information extracted from a versioning system can be efficient in detecting bad smells, because some bad smells need these information to insure precise results with no false-positives or false-negatives. However, it can be replaced when detecting other bad smells that rely more on static analysis. A combination of static analysis and using change history information would be perfect.

For the second approach, Java Quality Assurance by Detecting Code Smells, we found that it is simple and straight forward detecting bad smells by searching for its artifacts. However, this approach can not detect complex bad smells, because they may not be able to consider as artifacts, such as Blob, which has no artifacts to detect using this approach.

For the third approach, Automatic Generation of Detection Algorithms for Design Defects, we found that it is precise to detect bad smells and design defect using automatically generated algorithms. As each design defect has its own rule card, which specifies that design defect, accurate results are guaranteed. This approach considers complex bad smells and design defects such as Spaghetti Code, Swiss Army Knife and Functional Decomposition.

References

[1] William J. Brown et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st ed. Wiley, Apr. 1998. ISBN: 0471197130.

[2] E. van Emden and Netherlands L. Moonen. "Java Quality Assurance by Detecting Code Smells". In: *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002.

[3] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 1st ed. AddisonWesley, July 1999. ISBN: 0201485672.

[4] Naouel Moha, Yann-Gal Guhneuc, and Pierre Leduc. "Automatic Generation of Detection Algorithms for Design Defects". In: *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*. IEEE, 2006.

[5] Naouel Moha et al. "Decor: A method for the specification and detection of code and design smells". In: *IEEE Transactions on Software Engineering* 36.1 (2010), 2036.

[6] Fabio Palomba et al. "Detecting bad smells in source code using change history information". In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013.