International Journal of Innovative Research in Computer Science & Technology (IJIRCST)
ISSN: 2347-5552, Volume-4, Issue-3, May 2016

A Preamble Study on Smell Detection in Software Refactoring Techniques

M.Sangeetha, Dr.V.Sangeetha

Abstract— **Refactoring is the progression of expounding and shorten the design of obtainable code, without changing its deeds. Basically code smells are prearranged distinctiveness of software that may signify a code or design problem. The concept of refactoring covers practically any revision or cleaning up of source code, especially Improving the Design of Existing Code. To make possible software refactoring, a quantity of tools have been anticipated for code smell detection and/or for automatic or semi-automatic refactoring. However, these tools are reflexive and human driven, thus making software refactoring dependent on developers' naturalness. While refactoring can be applied to any programming language, the majority of refactoring current tools has been developed for the Java language. This paper converses refactoring is one of the method to reside software maintainable and the proposed framework could help developers to avoid similar code smells through timely warnings at the early stages of software development, thus reducing the total number of code smells surveys.**

Index Terms— **refactoring, bad smells, code smell, design of code, detection, IDE, IDEA.**

I. INTRODUCTION

REFACTORING is moderately a passionate area of research and consequently is not fine distinct. It's the progression of enchanting an object intend and rescheduling it in different ways to compose the design more flexible and/or reusable. There are numerous rationales you potency desire to do this, efficiency and maintainability being perhaps the most significant. In the direction of refactor programming code is to rephrase the code, to "clean it up". In mathematics, factorization or factoring is the decomposition of an object into an expression of smaller objects, or factors, which multiplied together, give the original.

Refactoring is the pitiful of units of functionality from one consign to another in your program. Refactoring has as a principal intention, receiving each portion of functionality to subsist in exactly one consign in the software. Good programmers write code that human can understand; When refactoring is typically provoked by perceive a code smell. In case the technique at tender possibly will be awfully extensive, or it might be a close to reproduction of an additional in close proximity routine.

Formerly renowned, alike tribulations preserve to deal with refactoring the resource code, or renovating it hooked on a new-fangled outline that performs the same as earlier than although to facilitate nix longer "smells". On behalf of a extensive practice, more than one smaller subroutines can be pull out; or for replacement routines, the repetition could be detached and reinstate with one mutual function. Malfunction to carry out refactoring can result in accumulating insignificant debt; on the further dispense, refactoring is solitary of the key means of repaying supposed liability.

.

II. TYPES OF REFACTORING

There are two broad-spectrum categories of benefits to the commotion of refactoring

A. Maintainability

It is easier to stick bugs because the source code is easy to read and the intent of its author is easy to grasp.[4] This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments.

B. Extensibility

It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.

Before refactoring a section of code, a solid set of automatic unit tests is needed. The tests are used to demonstrate that the behavior of the module is correct before the refactoring. If it inadvertently turns out that a test fails, then it's generally best to fix the test first, because otherwise it is hard to distinguish between failures introduced by refactoring and failures that were already there. After the refactoring, the tests are run again to verify the refactoring didn't break the tests. Of course, the tests can never prove that there are no bugs, but the important point is that this process can be cost-effective: good unit tests can catch enough errors to make them worthwhile and to make refactoring safe enough.

III. CODE SMELL

Code smell, also known as bad smell, in computer programming code, refers to any symptom in the source code of a program that possibly indicates a deeper problem.[1]

Manuscript received on May 6, 2016

Ms. M. Sangeetha, Department of Computer Science, Shakthikailash College, Dharmapuri, India (phone: 04348 2 47733 Mobile : 9790425960)

Dr. V. Sangeetha, Assistant Professor, Dept of Computer Science, Periyar University Arts & Science College, Metturdam, India

According to Martin Fowler[2] "a code smell is a surface indication that usually corresponds to a deeper problem in the system". Another way to look at smells is with respect to principles and quality:[3] "smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality". Code smells are usually not bugs—they are not technically incorrect and do not currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. Bad code smells are an important reason for technical debt.[1] Robert C. Martin calls a list of code smells a "value system" for software craftsmanship.[4]

Often the deeper problem hinted by a code smell can be uncovered when the code is subjected to a short feedback cycle where it is refactored in small[6], controlled steps, and the resulting design is examined to see if there are any further code smells that indicate the need of more refactoring[5]. From the point of view of a programmer charged with performing refactoring, code smells are heuristics to indicate when to refactor, and what specific refactoring techniques to use. Thus, a code smell is a driver for refactoring..

A 2015 empirical study[1] of half a million code commits to 200 open source software projects found that

- most bad smells affecting a piece of code are already present since its creation, rather than being introduced later via evolutionary code changes
- while most code smells are introduced while adding new features and enhancing existing ones, refactoring activities can also add bad smells
- "Newcomers are not necessary responsible for introducing bad smells, while developers with high workloads and release pressure are more prone to introducing smell instances", indicating a need for increased code inspection efforts in such stressful work situations.

IV. Common Code Smells

A. Application-level smells

1) Duplicated code: identical or very similar code exists in more than one location.
2) Contrived complexity: forced usage of overcomplicated design patterns where simpler design would suffice.

B. Class -level smells

1) Large class: a class that has grown too large. See God object.
2) Feature envy: a class that uses methods of another class excessively.
3) Inappropriate intimacy: a class that has dependencies on implementation details of another class.
4) Refused bequest: a class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class. See Liskov substitution principle.
5) Lazy class / Freeloader: a class that does too little.
6) Excessive use of literals: these should be coded as named constants, to improve readability and to avoid programming errors. Additionally, literals can and should be externalized into resource files/scripts where possible, to facilitate localization of software if it is intended to be deployed in different regions.
7) Cyclomatic complexity: too many branches or loops; this may indicate a function needs to be broken up into smaller functions, or that it has potential for simplification.
8) Downcasting: a type cast which breaks the abstraction model; the abstraction may have to be refactored or eliminated.[7]
9) Orphan Variable or Constant class: a class that typically has a collection of constants which belong elsewhere (typically a problem when using a Constants class) where those constants should be owned by one of the other member classes.

C. Method -level smells

1) Too many parameters: a long list of parameters is hard to read, and makes calling and testing the function complicated. It may indicate that the purpose of the function is ill-conceived and that the code should be refactored so responsibility is assigned in a more clean-cut way.
2) Long method: a method, function, or procedure that has grown too large.
3) Excessively long identifiers: in particular, the use of naming conventions to provide disambiguation that should be implicit in the software architecture.
4) Excessively short identifiers: the name of a variable should reflect its function unless the function is obvious.
5) Excessive return of data: a function or method that returns more than what each of its callers needs.

V. Detection of Bad Smell

Tool support is crucial for the success of bad smell detection and resolution because of the following reasons. First, uncovering bad smells in large systems necessitates the use of detection tools because manually uncovering these smells is tedious and time-consuming, especially those involving more than one file or package, e.g., duplicated code. The tools are expected to detect bad smells automatically or semi automatically. Clone detection is an excellent example, and researchers have proposed detection algorithms and developed tools for clone detection in the last decades. Second, software engineers need tools to automatically or semi automatically carry out refactoring to clean bad consuming and error prone. For example, renaming a variable requires revising all references to that variable[8].

Manually identifying all references is challenging—an issue that detection tools based on program analysis seeks to address. Most mainstream integrated development environments (IDE), such as Eclipse Microsoft Visual Studio and IntelliJ IDEA support software refactoring. Professional refactoring tools have also been developed. Rich tool support, in turn, accelerates the popularization of software refactoring.

Human intervention, however, remains indispensable to bad smell detection and resolution because of the following reasons.

First, most bad smells automatically detected should be rechecked manually because 100 percent precision cannot be guaranteed by detection tools. Second, it is up to software engineers to determine how to restructure bad smells in terms of refactoring rules that should be applied, and arguments of the rules.

VI. REFACTORING ETHICS

A. Code Hygiene

A popular metaphor for refactoring is cleaning the kitchen as you cook. In any kitchen in which several complex meals are prepared per day for more than a handful of people, you will typically find that cleaning and reorganizing occur continuously. Someone is responsible for keeping the dishes, the pots, the kitchen itself, the food, the refrigerator all clean and organized from moment to moment. Without this, continuous cooking would soon collapse. In your own household, you can see non-trivial effects from postponing even small amounts of dish refactoring: did you ever try to scrape the muck formed by dried Cocoa Crispies out of a bowl? A missed opportunity for 2 seconds worth of rinsing can become 10 minutes of aggressive scraping.

B. Specific "Refactorings"

Refactorings are the opposite of fiddling endlessly with code; they are precise and finite. Martin Fowler's definitive book on the subject describes 72 specific "refactorings" by name (e.g., "Extract Method," which extracts a block of code from one method, and creates a new method for it). Each refactoring converts a section of code (a block, a method, a class) from one of 22 well-understood "smelly" states to a more optimal state. It takes awhile to learn to recognize refactoring opportunities, and to implement refactorings properly.

C. Refactoring to Patterns

Refactoring does not only occur at low code levels. In his recent book, Refactoring to Patterns, Joshua Kerievsky skillfully makes the case that refactoring is the technique we should use to introduce Gang of Four design patterns into our code. He argues that patterns are often over-used, and often introduced too early into systems. He follows Fowler's original format of showing and naming specific "refactorings," recipes for getting your code from point A to point B. Kerievsky's refactorings are generally higher level than Fowler's, and often use Fowler's refactorings as building blocks. Kerievsky also introduces the concept of refactoring "toward" a pattern, describing how many design patterns have several different implementations, or depths of implementation. Sometimes you need more of a pattern than you do at other times, and this book shows you exactly how to get part of the way there, or all of the way there.

D. The Flow of Refactoring

In a Test-First context, refactoring has the same flow as any other code change. You have your automated tests. You begin the refactoring by making the smallest discrete change you can that will compile, run, and function. Wherever possible, you make such changes by adding to the existing code, in parallel with it. You run the tests. You then make the next small discrete change, and run the tests again. When the refactoring is in place and the tests all run clean, you go back and remove the old smelly parallel code. Once the tests run clean after that, you are done.

E. Refactoring Automation in IDEs

Refactoring is much, much easier to do automatically than it is to do by hand. Fortunately, more and more Integrated Development Environments (IDEs) are building in automated refactoring support. For example, one popular IDE for Java is eclipse, which includes more auto-refactorings all the time. Another favorite is IntelliJ IDEA, which has historically included even more refactorings. In the .NET world, there are at least two refactoring tool plugins for Visual Studio 2003, and we are told that future versions of Visual Studio will have built-in refactoring support.

- To refactor code in eclipse or IDEA, you select the code you want to refactor, pull down the specific refactoring you need from a menu, and the IDE does the rest of the hard work. You are prompted appropriately by dialog boxes for new names for things that need naming, and for similar input. You can then immediately rerun your tests to make sure that the change didn't break anything. If anything was broken, you can easily undo the refactoring and investigate.

VII. 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.

REFERENCES

[1] Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D. (2015-05-01). "When and Why Your Code Starts to Smell Bad" (PDF). 2015 IEEE/ACM 37th IEEE International

Conference on Software Engineering (ICSE) 1: 403–414. doi:10.1109/ICSE.2015.59.
[2] Fowler, Martin. "CodeSmell". http://martinfowler.com/. Retrieved 19 November 2014. External link in |website= (help)
[3] Suryanarayana, Girish (November 2014). Refactoring for Software Design Smells. Morgan Kaufmann. p. 258. ISBN 978-0128013977. Retrieved 19 November 2014.
[4] Martin, Robert C. (2009). "17: Smells and Heuristics". Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.ISBN 978-0-13-235088-4.
[5] Fowler, Martin (1999). Refactoring. Improving the Design of Existing Code. Addison-Wesley. ISBN 0-201-48567-2.
[6] Binstock, Andrew (2011-06-27). "In Praise Of Small Code". Information Week. Retrieved 2011-06-27.
[7] Miller, Jeremy. "Downcasting is a code smell". Retrieved 4 December 2014.
[8] Dr. V. Sangeetha, M. Sangeetha, "Fascinating Perspective of Code Refactoring", International Journal of Advanced Research in Computer Science and Software Engineering(IJARCSSE), Volume 6, Issue 1, ISSN: 2277 128X, pp. 164-168, January 2016.

AUTHORS BIOGRAPHY

Ms. M. Sangeetha received her B.Sc. degree in computer science from Bharathidasan University, Trichy in 1997 and M.Sc. degree in computer science from Bharathidasan University, Trichy in 1999. She also received M.Phil. degree in Computer Science from Periyar University, Salem in 2007. Currently she is working as Assistant Professor & Head. in the Department of Computer Science, Shakthikailash Womens College, Dharmapuri, India. She guided more than 15 M.Phil Scholars in the area of Computer Science. She is pursuing her Ph.D in the area Software Engineering at Periyar University. Software Refactoring, Agents, Data mining and Big Data Analytics are her current interests and research focus also.

