Performance - Detection of Bad Smells In Code for Refactoring Methods

Karnam Sreenu [1], D. B. Jagannadha Rao[2]

[1], (Assistant Professor, Department of Information Technology, Sreenidhi Institute of Science and Technology, India

[2], (Associate Professor, Department of Computer Applications, Institute of Science and Technology, India

ABSTRACT: *Day by day the complexity levels of Software system increasing. Hence more effort is required for software organizations to develop new or rebuild existing system of high quality. Refactoring reduces the cost of software maintenance through changing the internal structure of the source-code to improve the overall design that helps the present and future developers to evolve and understand a system. This paper describes new refactoring methods and metrics along with the existing metrics to identify the characteristics of bad smells "Lazy Class" and "Temporary Field" through which the developer can be provided with significant guidance to locate bad smells. After identifying these bad smells, appropriate refactoring methods can remove them.*

Keywords: *Software Refactoring, Bad Smells, Software Metrics, Software Quality.*

I. INTRODUCTION

The design of source-code has become an increasingly important part of the overall development of software. Refactoring changes the internal code structure of an Object-Oriented (O-O) system without affecting the overall behavior of the system to improve the quality of the design [1]. Refactoring is a process of making semantic-preserving transformations of code into a form that the software engineer finds easier to understand.

Refactoring or the restructuring of a software system without changing its behavior is necessary to remove quality defects that are introduced by quick and often unsystematic development.

Refactoring is starting to become an integrated part of other software development processes to improve the design, help make design changes, integrate new functionality, and help understand the underlying design concepts.

The process of refactoring has three distinct stages to its application: identify where to apply a refactoring, choose an appropriate refactoring as a solution and apply the refactoring. Current software tools and Fowler's description of refactorings only consider the final stage of applying refactoring methods automatically and manually. Knowing where an appropriate place and which refactorings to apply in a system is arguably quite difficult.

One particular motivation is to improve the design of a software system through locating problems in the design and using refactoring as a solution.

Fowler and Beck [1] defined bad smells that describe a design problem that have a number of related Refactorings that can change the structure of a system to help improve the design. However locating bad smells currently involves manually inspecting source-code, which quickly becomes unfeasible as the size of the system Increases. Providing an automatic support for the detection of bad smells becomes quite appealing.

The motivation for this paper is to enhance the well established refactoring process of identifying where to apply refactorings in a system. The focus will be on automatically identifying bad smell design problems in Java source code. To achieve the goal a prototype tool is developed that applies a set of software metrics on Java systems and the results are interpreted to identify problems in the design (i.e. bad smells).

II. BACKGROUND

2.1 Bad Smells

Some of the bad smells from Fowler's book [1] are summarized below:

- **Duplicate Code**: The same code structure in two or more places is a good sign that the code needs to be refactored: if you need to make a change in one place, you will probably need to change the other one as well, but you might miss it.
- **Long Method:** Long methods should be decomposed for clarity and ease of maintenance.
- **Long Parameter List:** Long parameter lists are hard to understand. You don't need to pass in everything a method needs, just enough so it can find all it needs.
- **Shotgun Surgery:** If a type of program change requires lots of little code changes in various different classes, it may be hard to find all the right places that do need changing. May be the places that are affected should all be brought together into one class.
- **Feature Envy:** This is where a method on one class seems more interested in the attributes (usually data) of another class than in its own class, May be the method would be happier in the other class.
- **Large Class:** Classes that are trying to do too much often have large numbers of instance variables.
- **Data Class:** Classes that just have data fields, and access methods, but no real behaviour. If the data is public, make it private.
- **Lazy Class:** Classes that are not doing much useful work should be eliminated.
- **Temporary Field:** It can be confusing when some of the member variables in a class are only used occasionally.

2.2 Refactoring Methods

- **Push down method:** 'Behavior on a super class is relevant only for some of its subclasses'. The method is moved to those subclasses.
- **Pull up Method:** 'You have methods with identical results on subclasses'. In this case, the methods should be moved to the super class.

- **Pull up field:** 'Two subclasses have the same field'. In this case, the field in question should be moved to the super class.
- **Move field:** 'A field is, or will be, used by another class more than the class on which it is defined'.
- **Rename Method:** A method is renamed to make its purpose more obvious.
- Rename Field: A field is renamed to make its purpose more obvious.
- **Move Method:** 'A method is, or will be, using or used by more features of another class than the class on which it is defined'.

2.3 REFACTORING PROCESS

Refactoring can be divided into a number of steps as shown below [2]:

1. Identify where the software needs to be refactored.
2. Determine which refactorings need to be applied to the identified places.
3. Guarantee that the applied refactoring preserves behavior.
4. Apply the refactoring.
5. Assess the effect of the refactoring on the quality characteristics of the software or the process.
6. Maintain the consistency between the refactored program code and other software artifacts.

III. PROPOSED WORK

This paper provides two new refactoring methods called Merge Class Refactoring and Replace Temp Refactoring, also describes two metrics called Number of Methods (NOM) and Instance Variable per Method in a Class (IVMC). These two refactoring methods are mainly used to reduce the lines of source code.

Also provides a bad smell description framework and bad smell interpretation framework to collect the information regarding bad smells. These frameworks mainly contain three parts.

Bad Smell Description Framework:

- Bad Smell Name: It is the description of the bad smell which is proposed by Fowler and Beck's.
- Characteristics of bad smell: Identifying main characteristics from description of the above bad smell.
- Identifying any design heuristics from the characteristics.

Bad Smell Interpretation Framework:

- Bad Smell Name: It is the description of the bad smell which is proposed by Fowler and Beck's.
- Measurement Process: Describe possible measurement techniques that when applied to Java source-code can help identify the design problem.
- Interpretation Rules: The interpretation indicates a set of rules on how the metrics can be used to identify possible candidates.We are using conventional metrics and new metrics to identify bad smells "Lazy Class" and "Temporary Field".

3.1 REFACTORING MODEL:

Figure 1 describes the detail process about how the bad smells are identified in the source code and determining which refactoring can be applied with the help of metrics values, then we can apply the appropriate refactoring method on the source code.

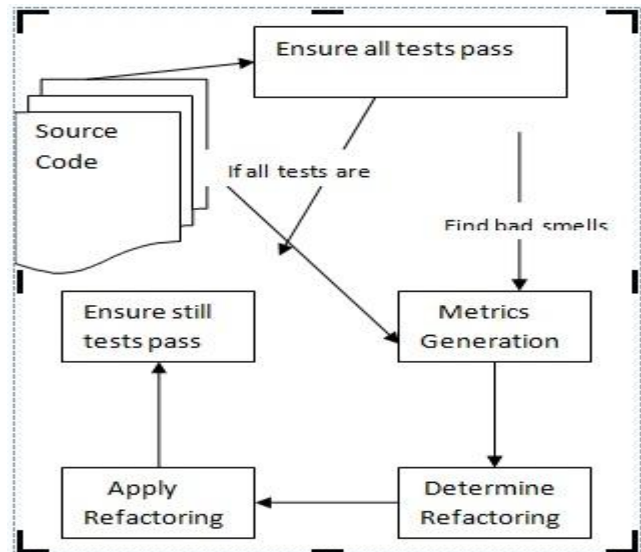


Figure 1: Describes the detail process

3.1.1. MERGE CLASS REFACTORING:

To apply this refactoring on the source code, first we have to identify the class to merge with the targeted class. To do this refactoring we are calculating some metric values to find out the lazy class which is not doing much work.

Lazy Class: To identify this bad smell the following are possible interpretation rules:

if NOM = 0
if (LOC < LOCThreshold)
if (DIT > 1)

LOC: Lines of Code
DIT: Depth of Inheritance

If the above rules are true then we can directly apply the merge class refactoring. That is the class which satisfies the above conditions can be merged with the targeted class.

Example:

```
Class Person
{
        String name;
        int getTelNumber();
}
Class TelNumber
{
        int areacode;
        String number;
        int getTelNumber();
        int getAreaCode();
}
```

On the above two classes we can apply the metrics to find out the lazy class, in which the lazy class is class "Person". So, it is merged with targeted class name called "Person".

```
Class Person
{
        String name;
        int areacode;
        String number;
        int getTelNumber();
```

```
    int getAreaCode();
}
```

Conditions:

I. If some classes have the same number of methods (NOM) then we should calculate Lines of Code for (LOC) those classes to know or decide which class has to be merged with targeted class.

II. The Merge Class Refactoring Method can be directly applied on the source code by calculating the Number of Methods (NOM) metric when there is no inheritance mechanism in the source code.

IV. We should calculate the Depth of Inheritance (DIT) metric to apply the Merge Class Refactoring Method on the source code when the inheritance mechanism will present in the source code.

3.1.2. Replace Temp Refactoring

We have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactoring's. Replace all references to that temp with the expression.

To apply this refactoring on the source code, first we have to identify the instance variable, that is not important in the source code. To do this the following interpretation rule is used.

if IVMC<=2

The above statement will be true when the instance variables which are declared in the code are not used more than two times. If the above rule is satisfied then we are ready apply the replace temp refactoring on the source code. By applying this refactoring on the source code we can remove the unused or unimportant instance variables so that the lines of the source code will be decreased.

Example1:

```
double random = ran.number();
return (random<=1);
```

In the above code "random" variable is treated as temporary variable, without its presence also the program is working correctly without changing its external behaviour. So, after applying the refactoring method the code will modified as following:

```
return (ran.number()<=1);
```

Example2:

The following code snippet also shows how the replace temp refactoring will occur to remove the instance variables which are not used more than two times.

Before Refactoring

```
int a=10, b=20, c=0;
c=a + b;
```

After First Refactoring

```
int a=10, b=20, c=0;
c=10 + b;
```

After Second Refactoring

```
int a=10, b=20, c=0;
c=10 + 20;
```

Conditions:

I. To apply this refactoring on the source code, first we have to identify the instance variable, that is not important in the source code.

II. The IVMC<=2 condition will be true when the instance variables which are declared in the code are not used more than two times.

V. IMPLEMENTATION

We have implemented the above mentioned metrics (NOM, LOC, DIT, and IVMC) in java [5] to find out the bad smells in the source code. If number of methods are equal in "Class A" and "Class B" then we have to calculate the Lines of Code for "Class A" and "Class B", based on these two values we have applied the appropriate Refactoring method. By using these implemented metric values we have applied the appropriate Refactoring method on the source code to remove the bad smell from the existing code or to improve the structure of the existing source code.

VI. CONCLUSION

The Merge Class Refactoring method and Replace Temp Refactoring method will be identified by providing metric values based on Number of Methods (NOM), Instance Variable per Method in Class (IVMC) and some existing metrics like Lines of Code (LOC), Depth of Inheritance (DIT) [3]. By identifying these metric values we can apply the above two refactoring methods directly on the source code to reduce the total number of lines of code (LOC) and to improve the structure of existing code.

REFERENCES

[1] Fowler M., Refactoring: Improving the Design of Existing Code, 1st ed: Addison-Wesley, ISBN 0-201-48567-2, 1999.

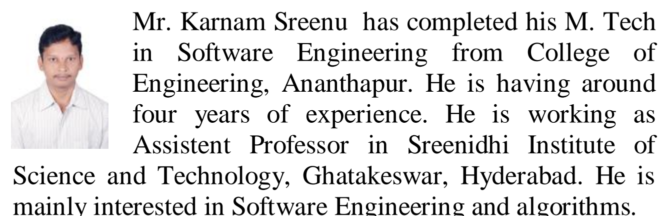
[2] T. Mens, T. Tourwe. "A Survey of Software Refactoring," IEEE Transactions on Software Engineering, Vol. 30, No.2, February 2004.

[3] Shyam R. Chidamber and Chris F. Kemerer "A Metrics Suite for Object Oriented Design"IEEE Transactions on Software Engineering, vol. 20, no. 6, June 1994.

[4] Rapu D., Ducasse S., Girba T., and Marinescu R., "Using history information to improve design flaws detection," presented at Eighth European Conference on Software Maintenance and Reengineering, Tampere, Finland, 2004, pp. 223-232.

[5] Naughton Schildt, The Complete Reference Java2 Third Edition.

AUTHOR'S PROFILE



Mr. Karnam Sreenu has completed his M. Tech in Software Engineering from College of Engineering, Ananthapur. He is having around four years of experience. He is working as Assistent Professor in Sreenidhi Institute of Science and Technology, Ghatakeswar, Hyderabad. He is mainly interested in Software Engineering and algorithms.

Mr. D. B. Jagannadha Rao has completed his M. Tech in Computer Science from College of Engineering, Ananthapur. He is having around nine years of experience. He is working as Associate Professor in Sreenidhi Institute of Science and Technology, Ghatakeswar, Hyderabad. He is especially interested in Mobile Ad-hoc Networks, Software Engineering and its routing algorithms.