The International Journal Of Engineering And Science (IJES)
|| Volume || 5 || Issue || 6 || Pages || PP -14-20 || 2016 ||
ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805

Analysis of Various Software Metrics Used To Detect Bad Smells

Sukhdeep Kaur[1], Dr. Raman Maini[2]
Department Of Computer Engineering, Punjabi University, Patiala-147002 (India)

ABSTRACT

Software metrics are evolving successfully and are used to measure the object oriented properties such as size, complexity, cohesion and coupling of software code. In different organization, software metrics are applied to evaluating the internal code quality, productivity as well as maintainability of software. Software metric are calculated to detect bad smells in source code. In any program, bad smell or code smell demonstrates the debilitate architecture design of software that makes it rigid to maintain in the future. In this paper, static and dynamic metrics are discussed to detect a handful bad smells like long method, feature envy, large class, long parameter list, lazy class, data class etc. From the analysis, it has been observed that dynamic metrics have more unambiguous results than static metrics because it depends upon the running or working environment.

Keywords: *Software Metrics, Measurement Process, Static Metrics, Dynamic Metrics, Bad Smell.*

Date of Submission: 26 May 2016 Date of Accepted: 05 June 2016

I. INTRODUCTION

Software metric is a numerical degree used to measure the given attribute that is possessed by software system, process or component. Software metric is a critical feature of software engineering that acts like an indicator for software attribute and used to measure the performance of software or its components. The software metric name is incorporated with distinct measurement process of software and all phases of system development [1]. Software measurement process is helpful to estimate the better cost, complexity, scheduling and efforts required for software during its development process. Reproducible measurement objective of software metrics are acquired for performance, quality assurance, debugging, estimating cost and management [2]. Fault in a source code, predicting project risk and success, predicting defective code are also finds through software metrics. In this paper, section II defines the software metrics used for three categories such as product, process and project. Section III represents the static and dynamic metrics and their different types. Section IV explains the bad smell and various types of bad smells in the code. Section V defines the bad smell detection process through software metric rules and also provides the analysis of various bad smells. Section VI gives the overall conclusion about software metrics and bad smells.

II. SOFTWARE METRICS

In the software engineering, Software metrics are essential research topic that is helpful for measuring software internal code quality, complexity, and efforts required to develop the project, product and process [3]. Some objective of software metrics are perception, planning, software assessment, software enhancement and quality progression. Basically, software metrics [4] are defined for following categories:

a. **Product metrics:** Product metrics are those metrics that describe the various features functioning as instance size, performance, complexity, design characteristics, portability and quality level of a product. Examples include the production system, quality of product and delivery time to the market.

b. **Process metrics:** Process metrics are those metrics that describe the effectiveness, enhancement of software development and maintenance. Examples include the efforts needed to process, defects found during testing and response time to produce the product.

c. **Project metrics:** Project metrics are those metrics that describe the characteristics of project and execution activities. Examples include the budget, software developers needed to complete project, staff pattern of whole software life cycle, scheduling and resource management, customer satisfaction.

Software quality metric is a factor of software metrics that establish the product quality areas for process, product and project [1]. Basically, product and process metrics are tightly closed to software quality metrics than project metrics because the parameters such as size, developer's skill levels, scheduling and the other corporation for project affect the caliber area of product.

III. TYPES OF SOFTWARE METRICS

Software metrics are defined under two categories such as static and dynamic metrics:

3.1. Static Metrics: At the preliminary stages of development life cycle of software, static metrics are procurable and it concerns with structural attribute of software system that are easy to collect. Actually, static metrics are measuring what may come off when a program is executed [1]. Static complexity metrics are defined to measuring the total efforts required to develop, maintain the code and non-executing code. Non executing code like blank line, comments, blank space provides the bad effect on the result of static metrics. Static metrics are not dependent on input test data and execution process [2]. Different types of static metrics [5],[6],[7],[8] are defined as following:

3.1.1. Lines of Source code (LOSC)

Lines of source code metric is used to count the only logical lines of the code. It does not count the comment lines and blank lines in a program's code. LOSC is software metric that count the lines of text in a source code for measuring the program size. In general, higher lines of source code in a program make it less understandable and maintainable. The LOSC>50 indicates the bad smell in a code.

3.1.2. Number of Attributes (NOA)

Number of attributes metric is used to count how many attributes in a code. It's normal range between 2 and 5. The number of attributes (NOA>10) indicates the bad smell design.

3.1.3. Number of Methods (NOM)

Number of methods metric is used to count how many operations in a program code. It's normal range between 3 and 7.

3.1.4. Cyclomatic complexity

Cyclomatic complexity metric used for measuring the program complexity through decision-making structure such as if-else, do-while, foreach, goto, continue, switch case etc. expressions in the source code. Cyclomatic complexity only counts the independent paths by a method or methods in a program. Complexity value of program is calculated using following formula [3]:

$$V(G) = E - N + P$$, where

V (G)	Cyclomatic complexity
E	No. of edges of decision graph
N	No. of nodes of decision graph
P	No. of connected paths

If there is no control flow statement like IF statement then complexity of program will be 1. It means there is single path for execution. If there is single IF statement then it provides two paths: one for TRUE condition and other for FLASE condition, so complexity will be 2 for it with single condition.

3.1.5. Weighted Methods per Class (WMC)

WMC metric is used to measures the single class complexity. This metric used to check the additional complexity of all methods and few instance variables in a class that are not accessed. Basically, WMC are calculated through cyclomatic complexity and also notify the maintainability of class. Normally it include

$$WMC/NOM \leq 2$$, where NOM is number of methods

Higher quantity of methods in a class has limited functionality for reusing. For this reason, smaller quantity of methods in one class signifies good usability and reusability of code.

3.1.6. Depth of Inheritance tree (DIT)

The depth of inheritance tree is a metric used to define the place of class in the hierarchy form like an inheritance tree. Depth of inheritance tree examines the largest length of class tree from node to root. This metric is measured through the number of predecessor classes. The Normal range of DIT values between 0 and 4. To find the DIT value, traverse the tree until the deepest child of class has been visited.

3.1.7. Class Coupling

Coupling metric used to measures the unique class coupling through local variables, parameters, return types, method calls, base classes etc. in the class program and also measure the program design complexity. Higher interdependency between classes makes it hard to maintain and also reduce reusing capability of code.

3.1.8. Coupling between Objects (CBO)

If one class method use the attributes or methods of the other class that means the class objects are coupled with each other. Coupling between Objects metric is used to count the number of objects of one class that are coupled with other class objects. Higher CBO decrease the software modularity.

3.1.9. Response for class (RFC)
Response for class metric used to measure the different methods in a class that methods are executed when object gives the response to a class message has been received.

3.1.10. Number of Public Methods (NPM)
This metric is used to count how many public methods are present in one class that methods are easily accessed by other classes. High NPM value indicates the class complexity, too much responsibility and high coupling with other classes. Large NPM values expose the internal security of a class.

3.1.11. Lack of Cohesion in Methods (LCOM)
A lack of cohesion in methods (LCOM) metric means group of methods in one class are not connected to other class methods or fields through sharing. The LCOM metric value is calculated by removing no. of method pairs that share other class field from no. of method pairs that does not share any field of other class.

3.1.12. Number of Parameters
This metric measure how many parameters in a constructor or method's signature. The normal range of number of parameters between 0 and 4 for a method or constructor. If the parameters larger than 5 in a method's signature, so it is needed to extract a new method or pass object to it.

3.1.13. Number of Accessor Methods
This metric used to count the class getter and setter methods.

3.2. Dynamic Metrics: At the late stage of development life cycle of software, dynamic metrics are attainable and it concerns with how many statements are executed, what function calls are literally taking place and what paths followed by program are being executed. Dynamic metrics means measuring what actually comes off while program is executed and these metrics are helpful for reliable design of software [1]. Dynamic metrics are dependent on input test data and execution process, so these metrics require care about machine architecture, language to be used, operating system and complier but it least dependent on the programming techniques used by system analyzer or programmer [2]. Different types of dynamic metrics [1],[9],[10] are defined as following:

3.2.1. Function Point
Function Point metric is a dynamic metric used for estimating the cost of software during designing, coding and testing. It basically measures the functionality delivered by system to a user. In function point metric, functionality is not direct measured, it is indirect measurement process. To calculate the function point value, information is collected from different parts like user inputs, outputs, inquiries, internal logical files and external interface files. Function point value is calculated using following formula [9]:

$$\text{Function Point} = \text{Count Total} * [0.65 + 0.01 * \text{Sum}(F_i)]$$

3.2.2. Halstead Complexity
Halstead metric measures the program that has series of operators and their correlated operands. This metric also provide the complexity information on a method of a class. Halstead metric is used to calculate the efforts, errors or bugs in program and time required to test the program through length and volume of a program at the run time. It has formula to calculate the value as following [9]:

$$\text{Efforts} = \text{Difficulty} * \text{Volume}$$
$$\text{Errors} = \text{Volume}/30$$
$$\text{Time} = \text{Efforts}/S \quad \text{where } S = 18 \text{ seconds}$$

3.2.3. Tight Class Coupling
Tight class coupling means one class is tightly bounded with other class. One class method or attribute cannot change without changing the other class; if it is changed then it provides the error or wrong result. Tight class coupling metric used to count the high level dependency between classes at the run time.

3.2.4. Loose Class Coupling
Loose class coupling metric used to count the low level dependency between classes at the run time.

IV. BAD SMELL

Bad smell means a piece of code in program that signifies the design problem in software code structure. Bad smell is a bug not a run time error. All bugs related to the program size and complexity makes the software architecture poor and tough to maintain the internal code of software [11]. These bad smells require refactoring to improve the code quality.

4.1. Bad smells in code: Dissimilar bad smells in code [12],[13],[14] are defined as following:

4.1.1. Long Method

Long Method bad smell signifies the great quantity of statements, variables, if-else and loop conditions in a single method that makes it tough to understand.

4.1.2. Large Class

Large Class bad smell signifies the class that is trying to do over plentiful functionality. More methods, variables or decision making conditions in a single class diminish the cohesion.

4.1.3. Feature Envy

Feature Envy bad smell indicates a method of one class that seems more intent in another class attribute from its present position.

4.1.4. Long Parameter List

Long parameter list bad smell means a large number of parameters are proceeding into one constructor or method signature that makes the code inconstant.

4.1.5. Data Class

Data class bad smell signifies a class which methods are accessed through getter/setter properties. Data class has no additional functionality and it reduces the security level of code because it is accessed by outsider classes.

4.1.6. Lazy Class

Lazy class bad smell means a class that does not have more functionality. It means there is no method in a class, only few lines of code.

V. BAD SMELL DETECTION

5.1. Software metric rules used to detect bad smells: Different types of software metric rules [13],[14],[15]are used to detect bad smells in a code that are defined as following:

Rules	Bad smell
a. **if** Number of source lines of code (NLOC) >50 and declared variables are not used. b. **if** Cyclomatic complexity >5 c. **if** Halstead effort E=D*V>15 If any above rule is/are true then bad smell is detected.	Long method
a. **if** Number of lines of code >300 and more than 5 long methods b. **if** Depth of inheritance tree (DIT)>5 c. **if** Class coupling > 10 If any above rule is/are true then bad smell is detected.	Large class
a. **if** Coupling between objects >5 b. **if** Lack of cohesion in methods>2 If any above rule is/are true then bad smell is detected.	Feature Envy
a. **if** Number of parameters of a method > 5 If above rule is true then bad smell is detected.	Long parameter list
a. **if** Lack of cohesion in methods>2 b. **if** Number of accessor methods >10 If any above rule is/are true then bad smell is detected.	Data class
a. **if** Number of methods =0 b. **if** LOC=100 and WMC/NOM<=2 If any above rule is/are true then bad smell is detected.	Lazy class

5.2. Analysis of bad smells

Bad smells are detected in the source code of program using various types of software metric rules. Long method bad smell is detected through calculating the values of three types of metrics such as source lines of code, halstead effort and cyclomatic complexity. Long method bad smell hides the unwanted duplicate code in a method and makes it more difficult to maintain. Large class bad smell is detected through calculating the values of three types of metrics such as lines of code, depth inheritance tree and class coupling. Large class bad smell doing too much work in one class. For this reason, it has high coupling in a class and it decreases the reusability of the code.

Feature envy bad smell is detected through calculating the values of two types of metrics such as coupling between objects and lack of cohesion in methods. Feature envy bad smell makes the code more inflexible because method of one class wants to move in other class attribute. This type of bad smell takes more time to debug the code. Long parameter list bad smell is detected through calculating the value of one type of metric such as number of parameters. Long parameter list bad smell makes the method's signature hard to understand and inconsistent. If method's signature has large amount of parameters then it is unchangeable code.

Data class bad smell is detected through calculating the values of two types of metrics such as lack of cohesion in methods and number of accessor methods. Data class bad smell has public variables or fields in a class. The data classes are directly accessed by other classes and also make the security level low of class. This class provides dumb folders for data. Lazy class bad smell is detected through calculating the values of two types of metrics such as number of methods and weighted method complexity. Lazy class bad smell waste the memory and time of a system because it has null functionality. Lazy class bad smell increases the complexity in the software system.

VI. CONCLUSION

Using different rules of software metrics, six types of bad smells are detected in source code. Large complexity and size of any program makes it difficult to understand and takes more time to debugging. All bugs related to program size, complexity, cohesion and coupling indicates the bad smells in source code. Mostly source code designing requires low coupling and high cohesion. In this paper, various software metrics are used to detect bad smells. Software metrics are divided into static and dynamic parts. The dynamic metrics like Function point and Halstead complexity etc. have more efficient and accurate results than static metrics because dynamic metrics are associated with executing code.

REFERENCES

[1]. AnkushVesra and Rahul "A Study of Various Static and Dynamic Metrics for Open Source Software", *International Journal of Computer Applications Volume 122 – No.10, 2015.*

[2]. Manik Sharma and Dr. Gurdev Singh "Analysis of Static and Dynamic Metrics for Productivity and Time Complexity**",** *International Journal of Computer Applications Volume 30– No.1, 2011.*

[3]. Gauri Khurana and Sonika Jindal "A model to compare the degree of refactoring opportunities of three projects using a machine algorithm", *Advanced Computing: An International Journal, Volume 4, No. 3, 2013.*

[4]. http://www.slideshare.net/Softwarecentral/software-metrics.

[5]. B Ramalkshmi and D. Gayathri Devi, "An Efficient Sdmpc Metric Based Approach For Refactoring Software Code", *International Journal Of Engineering And Computer Science, Volume 4, Issue 5 May 2015.*

[6]. Lorenz and Kidd "Role of object-oriented metrics in software measurement", *International journal of information technology & Systems, Volume 2, Issue 1, January 2013.*

[7]. Shweta Sharma and Dr S. Srinivasan "A review of Coupling and Cohesion metrics in Object Oriented Environment", *International Journal of Computer Science & Engineering Technology, Volume 4,No. 08, August 2013.*

[8]. Dr. A.T. Chamillard and Dr. David A. cook "Using Software metrics and Program Slicing for refactoring", *The Journal of Defense Software Engineering, Volume 5, No.2, July 2004.*

[9]. Hemlata Sharma, Anuradha Chug "Dynamic Metrics are Superior than Static Metrics in Maintainability Prediction: An Empirical Case Study", *IEEE 978-1-4673-7231-2/15/2015.*

[10]. http://www.tutorialspoint.com/software_engineering/software_design_complexity.

[11]. J. Fields, S. Harvie, M.Fowler, K. Beck; "Refactoring in Ruby", *Addison Wesley, 2009.*

[12]. Karnam Sreenu 1and D. B. Jagannadha Rao, "Performance - Detection of Bad Smells In Code for Refactoring Methods", *International Journal of Modern Engineering Research (IJMER), Volume 2, Issue 5, Sep-Oct 2012.*

[13]. Anshu Rani and Harpreet Kaur "Detection of bad smells in source code according to their object oriented metrics", *International Journal for Technological Research in Engineering Volume 1, Issue 10, 2014.*

[14]. Sandeep Kaur and Harpreet Kaur "Identification and Refactoring of Bad Smells to Improve Code Quality", *International Journal of Scientific Engineering and Research, Volume 3, Issue 8, August 2015.*

[15]. Ph.D thesis by Kwankamol Nongpong, University of Wisconsin-Milwaukee "Integrating Code Smells Detection with Refactoring Tool Support", *August 2012.*