ISSN No. 0976-5697

Volume 4, No. 9, July-August 2013

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Design and Implementation of Testing Tool for Code Smell Rectification Using C-Mean Algorithm

Gurpreet Singh*[1,] Vinay Chopra[2]

Asst.Prof. Deptt. of CSE[1],M.Tech Student Deppt. Of CSE[2]

DAV Institute of Engineering and Technology Kabir Nagar, Jalandhar

Gps_ghotra@yahoo.com[1], Vinaychopra222@yahoo.co.in[2]

Abstract:-A code smell is a hint or the description of a symptom that something has gone wrong somewhere in your code. These are commonly occurring patterns in source code that indicate poor programming practice or code decay. The presence of code smells can have a severe impact on the quality of a program, i.e. making system more complex, less understandable and cause maintainability problem. Herein, an automated tool have been developed that can rectify code smells present in the source code written in java, C# and C++ to support quality assurance of software. Also, it computes complexity, total memory utilized/wastage, maintainability index of software. In this research paper an approach used for the design and implementation of testing tool for code smell rectification is discussed and is validated on three different projects.

Keywords: - Code Smell, Refactoring, Maintainability, Memory Utilization, Inspection, McCabe Cyclomatic Complexity, Halstead Measure.

I. INTRODUCTION

The concept of code smell was introduced Fowler and Beck as an indicator of problems within in the design or code of software by presenting an informal definition of 22 code smells. Code smells indicate that there are issues with code quality, such as understandability and changeability, which can lead to the introduction of faults [1]. A common set of design principles such as data abstraction, encapsulation, and modularity should be followed for object oriented software systems in order to assure the non-functional requirements[2][3][4]. Although developers are used to these techniques, but deadline pressure, too much focus on pure functionality or just inexperience may lead to violation of these design principle rules.

Code smells are usually not bugs—they are not technically incorrect and don't 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 [5]. Each code smell examines a specific kind of system element (e.g. classes or methods), that can be evaluated by its inner and external characteristics. The detection of code smells manually by code inspection [1], leads to different issues which are identified by Marinescu[6] as: time-expensive, non-repeatable and non-scalable. Even more issues concerning the manual detection of design flaws were identified by Mäntylä[7][8]. He showed that as the experience a developer has with a certain software system increases, his ability to perform an objective evaluation of the system as well as his ability to detect design flaws decreases. Not necessarily all the code smells have to be removed: it depends on the system. When they have to be removed, it is better to remove them as early as possible. If we want to remove smells in the code, we have to locate and detect them; tool support for their detection is particularly useful, since many code smells can go unnoticed while programmers are working[9].

In this research paper an automated tool has been designed and in rest of the paper numbers of questions wereanswered, i.e. how and which kind of code smells can it identifies? , how many languages does it support? , what refactoring has been applied on the code smells identified? How it computes Maintainability Index, Memory Utilization. This tool provides range of functionalities that helps improve quality of code by rectifying various code smells.

II. DETECTION APPROACH

In the study reported herein, we used automatic heuristics to detect the smells. These detection strategies interpret a set of code metrics that are extracted from a specific system component by using set of threshold filter rules. The main goal of this approach is to provide engineers with a mechanism that will allow them to work with metrics on a more abstract level, which is conceptually much closer to the real intentions in using metrics. Each detection strategy is structured in three consecutive elements: 1) A set of code metrics. 2) A set of filtering rules, one rule for the interpretation of each metric result. 3) The composition of filtered result.

C-Mean Algorithm is used to partition the code smells into different clusters based on the ruleset defined. The C-Mean algorithm starts with an initial partition then it tries all possible moving or swapping of data from one group to others iteratively [10].

a. Initially a set of m objects $[O_1,O_2,\ldots O_m]$ which must be grouped in c clusters. Each object is described by a set $R=\{x_1,x_2,\ldots x_n\}$of features.
b. Iteratively scan the objects and compare the features based on the rules specified.
c. Update each cluster.
d. Repeat step 2 and 3 until all classes has been scanned for code smells.

The Ultimate goal of clustering is to provide users with meaningful insight from the original data, so that they can effectively solve the problems encountered.The tool developedherein, is able to detect Long method, Large Class, Long Parameter list, Duplicated code, Switch Statements, Dead code, Temporary fields, Lazy Class and

© 2010, IJARCS All Rights Reserved

comments code smell. Herein the detection strategy for long method, large class and duplicated code is discussed.

A. Long Method:

No matter what the program paradigm is, long procedures, functions or methods are hard to understand[1]. The longer they are, the more parameters and variables they use, and long methods are more likely to do more than their name suggests.To detect Long method logical lines of code (LLOC), McCabe's Cycomatic complexity, Halstead volume and number of local variables left unused were considered.

a. **LLOC** is a variant of LOC. It shows the count of logical statements in a program, it only counts the statements which end at semi-colon. A threshold equal to 30 is taken for LLOC.
b. Thomas McCabe introduced a metric in 1976 based on the control flow structure of a program [11]. This metric is known as **McCabe cyclomatic complexity** and it has been famous code complexity metric throughout since it was first introduced. The McCabe metric is based on measuring the linearly independent path through a programand gives cyclomatic complexity of the program which is represented by a single number. McCabe noted that a program consists of code chunks that execute according to the decision and control statements, e.g. if/else and loop statements. McCabe metric ignores the size of individual code chunks when calculating the code complexity but counts the number of decision and control statements.Athreshold equal to 10 is taken.
c. A suite of metrics was introduced by Maurice Howard Halstead in 1977. Halstead volume can be calculated as:

$$V=N.\log_2 \eta$$

Where, N= Program length, η = Program vocabulary and V= program volume. Volume can be interpreted as bits, hence is the measure of storage volume required to represent the program [12]. Halstead observed that there is a relationship between code complexity and program volume. According to Halstead, *code complexity increases as volume increases*.

B. Large Class:

Large Classes are classes with too many responsibilities [1]. They have too much data and/or too many methods. The problem behind this smell is that these classes are hard to maintain and understand because of their size. Large Class code smells often coincide with Duplicated Code or Shotgun Surgery smells.

a. If LLOC is greater than 300 and has more than 5 long methods.
b. If number of instance variables and methods are greater than 15 and 10 respectively.
c. Weighted method count (WMC) is a count of sum of complexities of all methods in a class. A threshold of 20 is taken for a class to be large.
d. Depth of Inheritance tree (DIT), it access how deep, a class is in hierarchy structurei.e., maximum inheritance path from a class to the root class. DIT greater than 6 is considered for a class to be large.
e. Coupling, when one object interact with another object that is a coupling. Strong coupling is discouraged because it results in less flexible, less scalable application. A threshold of 10 is considered.

C. Duplicated Code:

The same code structure in two or more places is a good sign that the code need to be refactored: if you need to change in one place, you'll probably need to change the other one as well, but you might miss it [1][2]. Rabin karp algorithm is used to detect duplicated code. Given a text string t and a pattern string p, find all occurrences of p in t [13].The Rabin-karp string searching algorithm calculates a hash value for the pattern, and for each M-character subsequence of text to be compared. if the hash values are equal, the algorithm will do a brute force comparison between the pattern and the M-character sequence. Herein five consecutive lines were considered to find duplicated code.

D. Long Parameter List:

Long parameter list means that a method takes too many parameters. Long Parameter lists are prone to change, difficult to use, and hard to understand. With objects you don't need to pass in everything the method needs, instead you pass in enough so the method can get to everything it needs [1]. We thus need to decide how many parameters are too many. McConnell's guidebook for procedural programming [14] recommends that the number of parameters should be limited to seven. Object-oriented programming generally requires less parameter passing, since classes can encapsulate data and operations together. Therefore, we also selected two other parameter limits with values of three and five. We thus have ended up with three opinions on what a long parameter list is. The can be understood as three tolerance levels: low, medium, and high.

a. The maximum number of parameters in these categories is three for low, five for medium, and seven for high.
b. If Number of parameters of a method is greater than Average_Parameters+2 and some of which is not used, where

Average_parameters= $\sum$ n parameters of a method) / M, for all method in C

M=number of methods in a class.

E. Switch Statements:

Switch Statements also known as State Checking manifests itself as conditional statements that select an execution path based on the state of an object. Switch statements tends to cause duplication [1]. You often find similar switch statements scattered through the program in several places. If a new data value is added to the range, you have to check all the various switch statements. The presence of this smell essentially signifies a violation of the Open-Closed Principle [15] since any future modification in the actions associated with a particular state or the addition of new states will require the modification of existing code increasing the required effort and the possibility of introducing errors.

a. The McCabe cyclomaticgreater than 10 is considered.

© 2010, IJARCS All Rights Reserved

b. If numbers of cases are greater than 10 and two or more cases contain duplicated code.

III. REFACTORING

Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves the internal structure[16]. It improves the design of the software by eliminating redundancy and reducing complexity. The resulting software is easier to understand and maintain[17].Refactoring opportunities are locations in the source where a) there is a need for improvement regarding a quality attribute; b) a refactoring can be applied that will reorganize the code while preserving the behaviour of the software system; and c) the application of the refactoring will indeed improve the quality attribute.
Major of the refactoring on the code is a manual process. The task of improving the code is done in three phases:

a. Identify various code smells in the code.
b. Select and apply suitable refactoring.
c. Assess the effect of refactored code i.e., whether any improvement achieved.

In order to correct Long Method code smell, **Extract Method** refactoring is applied. How do you identify the clumps of code to extract? A good technique is to look for comments. They often signal this kind of semantic distance. A block of code with a comment that tells you what it is doing can be replaced by a method whose name is based on the comment. Even a single line is worth extracting if it needs explanation. To break up the class three approaches are most common, first, **Extract class** (if you can identify a new class that has a part of this class's responsibilities); secondly, **Extract subclass** (if you can divide responsibilities between the class and new sub class); third, **Extract interface** (if you can identify subsets of features that clients use).Use **Replace Parameter with Method** when you can get the data in one parameter by making a request of an object you already know about. This object might be a field or it might be another parameter. UsePreserve Whole Object to take a bunch of data gleaned from an object and replaces it with the object itself. If you have several data items with no logical object, use Introduce Parameter Object.

The simplest duplicated code problem is when you have the same expression in two methods of the same class. Then all you have to do is **Extract Method** and invoke the code from both places.Most times you see a switch statement you should consider **polymorphism**. The issue is where the polymorphism should occur. Often the switch statement switches on a type code. You want the method or class that hosts the type code value. So use Extract Method to extract the switch statement and thenMove Method to get it onto the class where the polymorphism is needed. At that point you have to decide whether to Replace Type Code with Subclasses or Replace Type Code with State/Strategy. When you have set up the inheritance structure, you can use Replace Conditional with Polymorphism.

IV. MAINTAINABILITY INDEX

Software maintenance includes all post implementation changes made to a software entity[18]. IEEE defines software maintainability as "the ease with which a software system orcomponent can be modified to correct faults, improve performance or other attributes, oradapt to a changed environment. "Maintainability index (MI) is a single valued metric where index value between 0 - 100 iscalculated to represent the ease of maintainability of the code or software product. Higheror closer to 100 value of MI means better maintainable code while low value will representthe code that will be hard to maintain. The construction of MI is based on four matric model that includes McCabe's Cyclomatic Complexity [19], Halstead Volume [20], Source line of code (SLOC) and average number of lines of comment per module.

V. RESULTS AND DISCUSSION

In order to test the tool developed, the source code for three different projects namely, Banking System, Web Browser and Hotel management system in .Net (C#), .java and C++ respectively were downloaded from http://www.planet-source-code.com/. These source codes were tested for presence of different code smells so as to improve its quality further.

Table 1: Description of projects under consideration

S.No	Project Name	Language	LOC
1.	Banking Management System	C#	2500
2.	Web Browser	Java	2255
3.	Hotel Management System	C++	1900

The tool takes source code as input and identifies different types of code smells presents in it, computes memory utilization and maintainability index for the same. When the above listed projects were analysed by the tool it was observed that the number of code smells were easily removed by applying the refactoring described in section III, moreover the Maintainability index was increased and memory wastage was less. Below Fig 1 to Fig 9 shows results computed by the tool for three different projects.

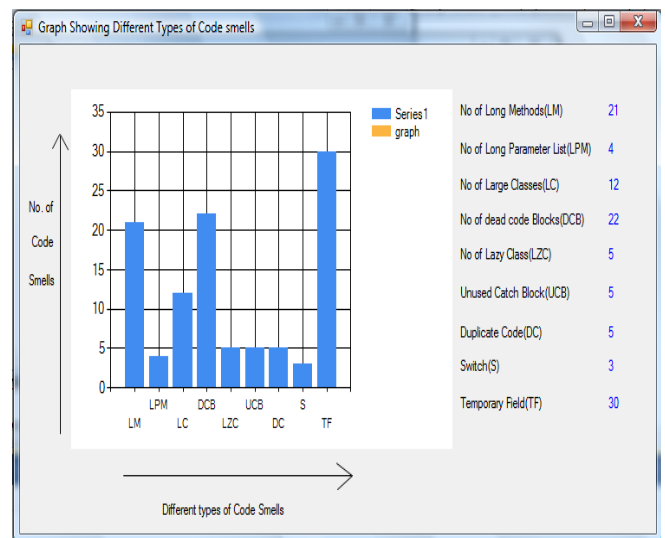

Figure 1: Different types of code smells in Banking System

© 2010, IJARCS All Rights Reserved

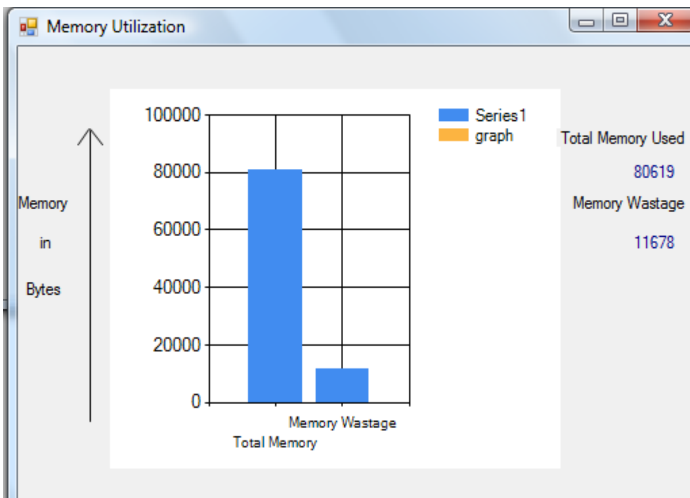

Figure 2: Memory Utilization for Banking System

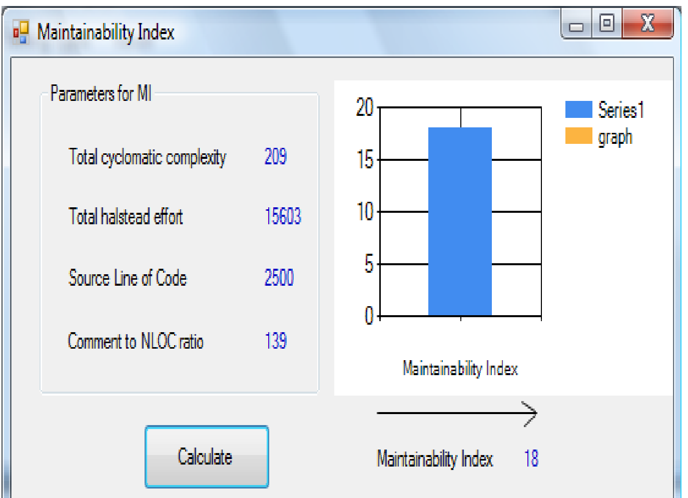

Figure 3: Maintainability Index of Banking System

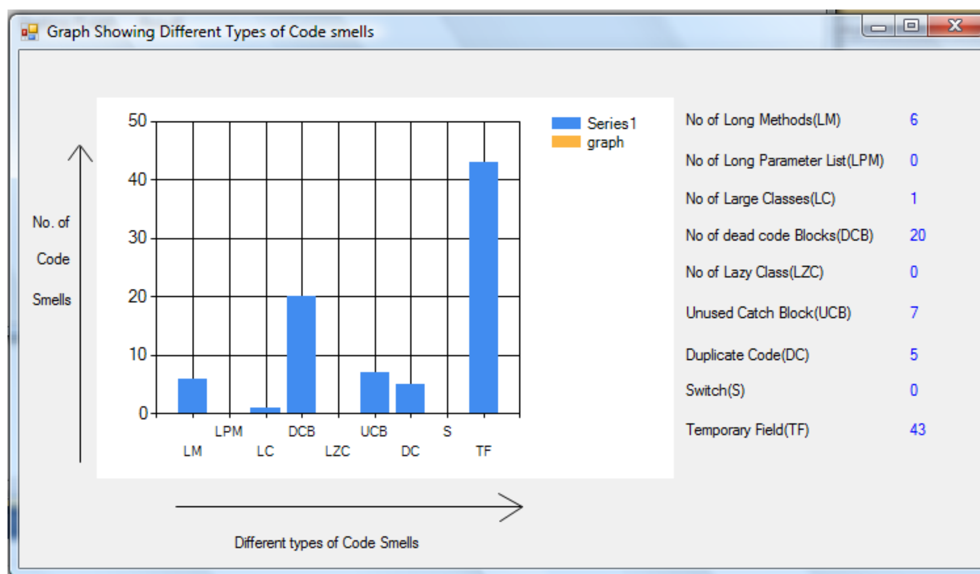

Figure 4: Different types of code smells in Web Browser

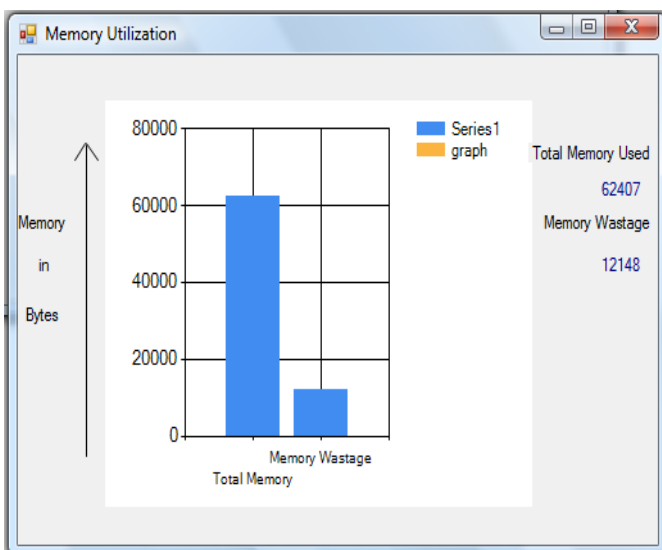

Figure 5: Memory Utilization for Web Browser

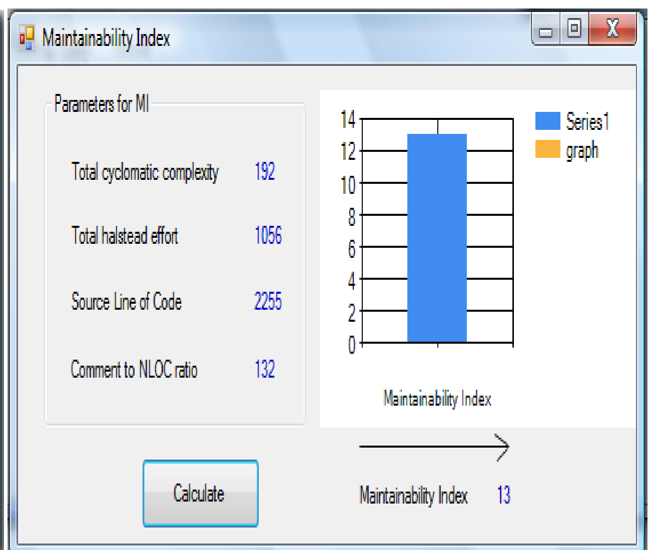

Figure 6: Maintainability Index of Web Browser

© 2010, IJARCS All Rights Reserved

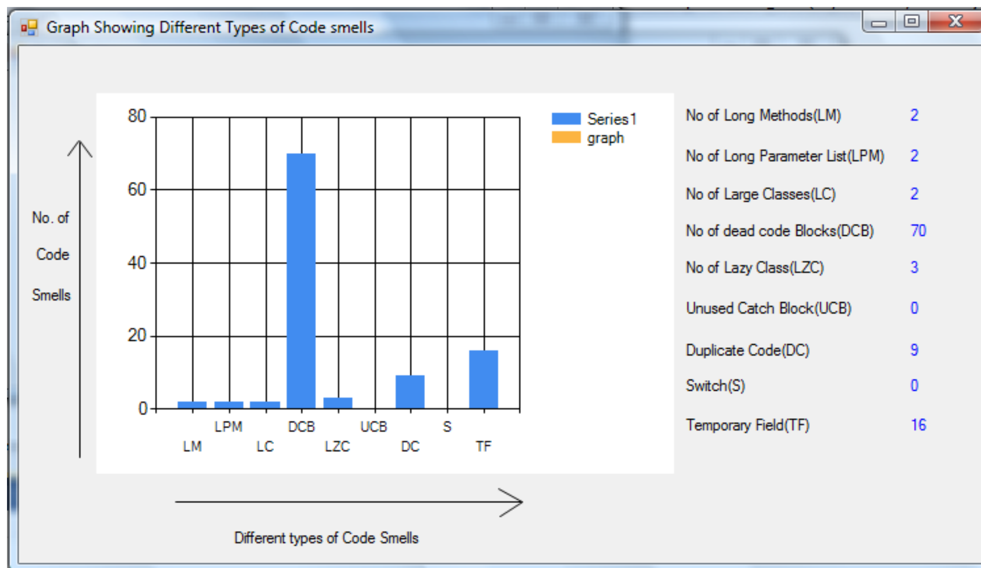

Figure 7: Different types of code smells in Hotel Management System

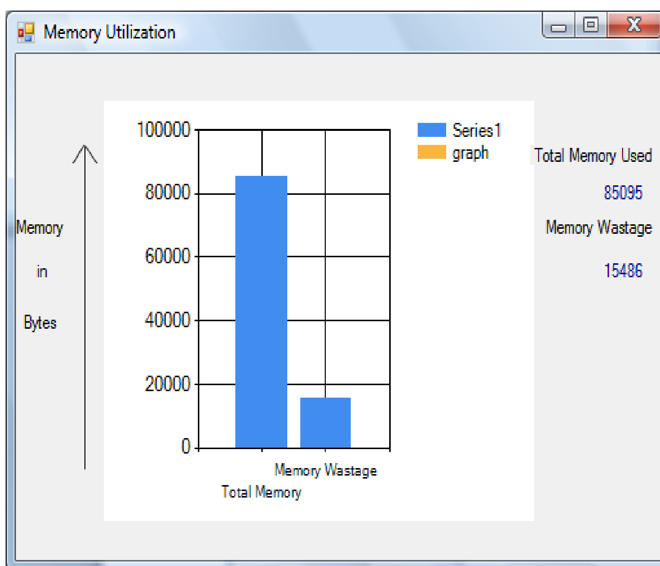

Figure 8: Memory Utilization for HMS

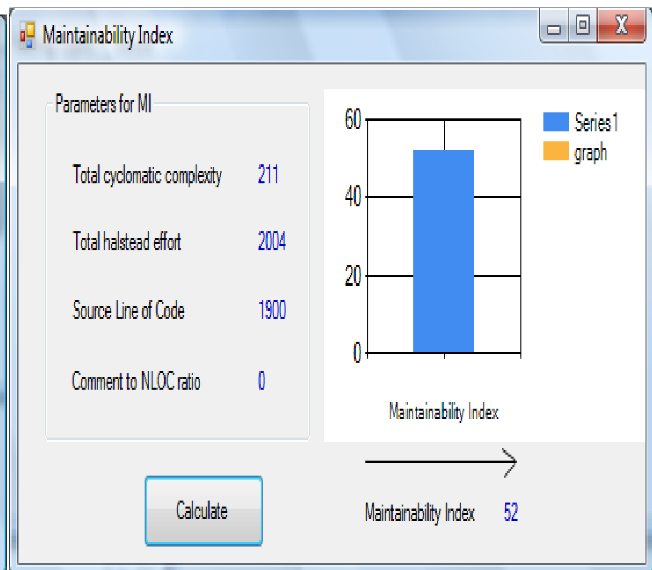

Figure 9: Maintainability Index of HMS

The figures above show the different types of code smells detected by the tool, memory utilization and maintainability index for three different projects, i.e. (Banking System, Hotel Management System, and Web Browser), . In order to detect different code smells in the source code of different projects the rule set defined in section II for various code smells were used. Each code smells contain number of rules based on the properties they possess, so that maximum number of code smells presents can be detected and no code smell is left in the source code. To calculate the maintainability index of source code of different projects four metric MI model was used, according to which MI is calculated as below in Eq. 1.

$$MI = 171 - 32.4 \times \ln(\text{aveE}) - 0.23 \times \text{aveVG} - 16.2 \times \ln(aveLOC) + 0.99 \times \text{aveCMT} \quad \text{Eq. 1}$$

Where,

- MI is a Maintainability index of the module
- aveE is average Halstead effort per module
- aveVG is average cyclomatic complexity per module
- aveLOC is average lines of code per module
- aveCMT is average number of lines of comments per module.

Once the various code smells have been detected the next step is to remove them by applying an appropriate refactoring specified in section III for each code smells. Major of the refactoring on the code is a manual process.When the refactored code is given to the tool as input, the results show significant improvement. The numbers of code smells were reduced to larger extent; as a result which, source code is now well structured. Moreover the maintainability index of the code in increased that indicates relative ease of maintaining the code, easy to test the code and more understandable. This means that the overall quality of source code is improved.

The table 2 below shows the different number of code smells detected and corrected in three different projects, and table 3 shows the MI for each of three projects before and after correction.

© 2010, IJARCS All Rights Reserved

S.No	**Code Smells**	**Banking System**		**Hotel Management System**		**Web Browser**	
		No. of code smells before correction	No. of code smells after correction	No. of code smells before correction	No. of code smells after correction	No. of code smells before correction	No. of code smells after correction
1.	Long Method	21	9	2	0	6	2
2.	Long Parameter List	4	0	2	0	0	0
3.	Large Class	12	7	2	1	1	1
4.	Switch Statement	3	1	0	0	0	0
5.	Duplicated Code	5	0	9	0	5	0
6.	Lazy Class	5	2	3	1	0	0
7.	Dead Code	22	0	70	0	20	0
8.	Empty Catch Block	5	0	0	0	7	0
9.	Temporary Field	30	0	16	0	43	0
TOTAL		**107**	**18**	**104**	**2**	**82**	**3**

Table 2: Number of Code smells detected and corrected in Banking System, HMS and Web Browser

Projects		**Total Cyclomatic complexity**	**Total Halstead volume**	**Source lines of code**	**Comment to Line of code ratio**	**Maintainability index**
Banking System	Before Correction	209	15729	2500	5	18
	After Correction	**197**	**13473**	**2674**	**5**	**49**
Hotel Management System	Before Correction	192	1056	2255	132	13
	After Correction	**170**	**1254**	**2097**	**20**	**53**
Web Browser	Before Correction	211	2004	1900	0	52
	After Correction	**187**	**3178**	**2018**	**0**	**81**

Table 3: Values of Various parameters to calculate MI for different project

© 2010, IJARCS All Rights Reserved

VI. CONCLUSION AND FUTURE SCOPE

The tool developed is capable of performing code analysis automatically on regular basis. It can analyse source code written in three different languages i.e., Java, C++ and .Net (C#). With the help of this developers can view quality of their code. As a result of this tool automatic measurement of source code complexity is possible to implement. Potentially fault-prone code can easily be identified which can suggest developers about the code that require refactoring. It is also possible to identify what parts of code have changed and how much they are changed. The tool built can effectively compute the memory utilization and measures maintainability index value between 0-100 that represent relative ease of maintaining the code. The results observed in section V shows significant improvement in the quality of software. In the future the research work must focus on the identification and removal of the left code smells, so as to make source code free from all types of code smells given by Fowler et.al, 1999.

VII. REFERENCES

[1] Fowler, M. and K. Beck, Refactoring: improving the design of existing code. 1999: Addison-Wesley Professional.

[2] Johnson, R.E.; Foote, B. Designing reusable classes. Journal of Object-Oriented Programming, Journal of Object Oriented Programming, pp. 22-35, 1988

[3] Rising, L.S.; Calliss, F.W. An experiment investigating the effect of information hiding on maintainability. 12th Ann. Int. Phoenix Conference on Computers and Communication, March, pp. 510-516, 1993

[4] Wilde, N.; Mathews, P.; Ross, H. Maintaining Object-Oriented Software. Addison-Wesley, 1993

[5] Code Smells: http://en.wikipedia.org/wiki/code_smel l.

[6] Marinescu, R. Detecting Design Flaws via Metrics in Object-Oriented Systems. TOOLS pp. 173-182, 2001

[7] Mäntylä, Mika. Vanhanen, Jari and Lassenius, Casper ,A Taxonomy and an Initial Empirical Study of Bad Smells in Code. ICSM, pp.381-384, 2003

[8] Mäntylä, M.; Vanhanen, J.; Lassenius, C. Bad Smells - Humans as Code Critics. ICSM pp.399-408, 2004

[9] Fontana, F.A. and Braione, P. and Zanoni, M. “Automatic detection of bad smells in code: An experimental assess”, Journal of Object Technology, Vol.11 No.2, 2012.

[10] Xu, Rui and Wunsch, Donald and others, Survey of clustering algorithms, IEEE Transactions on Neural Networks, vol.16 no.3 pp: 645-678,2005.

[11] T. J. McCabe, “A Complexity Measure,” ICSE '76: Proceedings of the 2nd international conference on Software engineering, 1976.

[12] Everald E. Mills, "Software Metrics", Software Engineering Institute, 1988.

[13] Crespo, Yania, Carlos Lopez, Raul Marticorena, and Esperanza Manso, "Language independent metrics support towards refactoring inference,“ in 9th ECOOP Workshop on QAOOSE 05 (Quantitative Approaches in Object-Oriented Software Engineering, Glasgow: UK. ISBN: 2-89522-065-4, July 2005.

[14] S. McConnell, Code Complete, Redmond, Washington, USA: Microsoft Press, 1993.

[15] R. C. Martin, Agile Software Development: Principles, Patterns and Practices, Prentice Hall, 2003.

[16] Martin Fowler “Improving the Design of Existing Code”Addison Wesley, Massachusetts, April 2006.

[17] Van Emden, E. and Moonen, L. “Java quality assurance by detecting code smells” Proceedings Ninth Working Conference on Reverse Engineering, IEEE, pp.97-106, 2002.

[18] Young Lee, "Automated Source Code Measurement Environment for Software Quality",Auburn University Alabama, December 2007.

[19] T.J. McCabe, “A Complexity Measure, “ICSE ’76: Proceeding of the 2nd International Conference on Software Engineering, 19776.

[20] Hamer, P. G. and Frewin, G. D., "M. H. Halstead's Software Science - A CriticalExamination", in the Proceedings of the 6th International Conference on SoftwareEngineering, Tokyo, Japan, 1982.

© 2010, IJARCS All Rights Reserved