International Journal of Latest Trends in Engineering and Technology
Vol.(7)Issue(4), pp.421-436
DOI: http://dx.doi.org/10.21172/1.74.057
e-ISSN:2278-621X

A UNIQUE CODE SMELL DETECTION AND REFACTORING SCHEME FOR EVALUATING SOFTWARE MAINTAINABILITY

Rohit Kumar[1] and Jaspreet Singh[2]

Abstract – Code smell (CS) is a sign that tells something has gone incorrect, somewhere in the code. Such problems are neither bugs nor they are technically wrong. Moreover, they do not prevent the program from its functioning. CS indicates the flaws in the design that may be a reason to slow down the development in the near future. From software engineer's perspective, detecting CS remains major concern so to enhance maintainability. However, it is a time consuming task. Refactoring method can be implied to remove CSs. Refactoring is a technique used to reconstruct the body of current code by changing its inner structure, without changing its outer behavior. Current CS detection tools are not equipped with functionality to assess the parts of code where improvements are required. Hence, they are unable to re-factor the actual code. Further, no functionality is available to permanently remove the CSs from the actual code thereby increasing the Risk factor. In this paper, a unique technique is designed to identify CSs. For this purpose, various object-oriented programming (OOPs)-based-metrics with their maintainability index are used. Further, code refactoring and optimization technique is applied to obtain low maintainability Index. Finally, the proposed scheme is evaluated to achieve satisfactory results.

Keywords – Code smell, object-oriented programming, optimization, refactoring, software maintenance, oops Metrics.

I. INTRODUCTION

Today's software has become part of everyone's life. The rule of software is its capability to make our lives easier, get better productivity and efficiency. However, such efficiencies come at the cost of all-encompassing observation [1].A characteristic of software that is such a achievement that humanity should never forgets.Smells are certain structure in the code that sign violation of major design principles and adversely impact of design quality. Code Smell (CS) are normally not errors, neither are they technically wrong nor do they check the program. Instead, CSs are weakness of design that may be slowing down along with increasing the high risk of errors or bugs in the future. CSs have been defined as sign of poor plan and execution choices. In some cases, such sign may be invented by activities performed by developers while in a speed such as, implement urgent patch or simply making suboptimal choices. While most CSs are presented, adding new characteristics better than the existing ones, refactoring events can also add bad smells. New features are not responsible for presenting bad smells, while engineers with high workloads are more responsible. Hence, by releasing pressure from engineers may be more

[1] *Chandigarh Engineering College Mohali, (Punjab), India*
[2] *Chandigarh Engineering College Mohali, (Punjab), India*

beneficial to present smell objects. Moreover, it represents need for large code inspection efforts in such busy work situations.

Refactoringmeans easy and clears the structure of previous code, without changing its behavior. Agile teams are extending and maintaining their code a lot by making repetitions [2]. But, they do not use continuous refactoring as it is no easy. This is because un-factored code tends to rot. Numerous forms are generated by un-factored rot which depicts unhealthy dependencies per method or class, duplicate code, and some other varieties of mix-up and disorder. Every time we change the code without refactoring it, rot degrades and spreads. Code rot frustrates us along with costing us time and shortens the lifespan of useful system. Refactoring process consists of various events as given below.

- Recognize whenever the software that should be refectories.
- Establish which refactoring should be useful.
- Agreement that the functional refactoring conserves behavior and apply the refactoring.
- Evaluate consequence on quality of the software (e.g., maintainability) & procedure (e.g., efficiency).
- Continue reliability between the refectories programming code and other software object (i.e. documents, propose papers, experiments and so on).

Detected code smells will differ depending on the preferred likelihood threshold [3]. Growing the probability too much will reason more false negative, while falling it in excess will grounds more false positives. It will be up to the developer to fine adjust the threshold to get the sufficient level of advice with respect to the occurrence of CSs. It will also be up to the developer to choose the sufficiency to relate a given refactoring to eliminate a detected CS. The block diagram of the detection model for CS is shown in Fig. 1.The model clearly depicts all the steps starting from dividing the source code into classes and trees moving on to calculation OOPs metrics. The model then compares facts and rules with code and finally concludes with results.

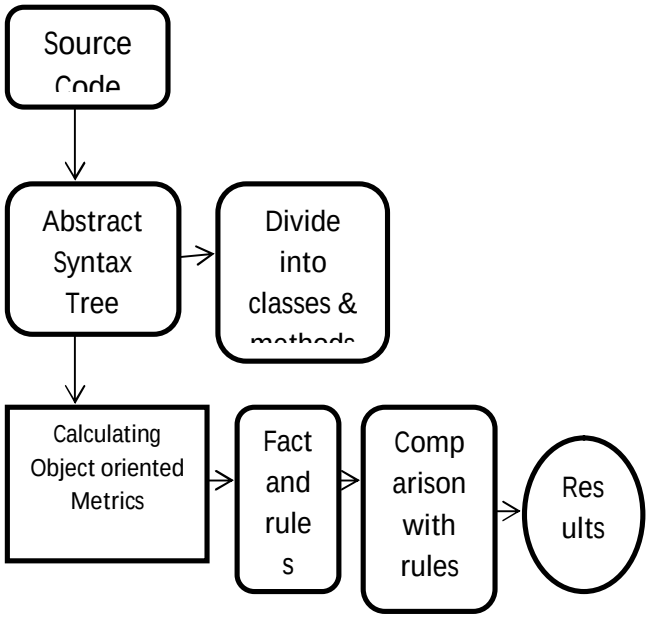

Fig. 1: Overview of detection model

II. BACKGROUND

This section gives procedural background tosoftware maintenance process; CS and software metrics, threshold for software metrics and risk assessment.

A. Maintenance of Software

The alteration in software created after delivery so as to correct mistakes, to modify presentation or other aspects is known as software maintenance.The detailed study on how [4] a plan functions before it can modification it is the preliminary task. It is frequentlyrelated with difficult and hard to understand systems. Maintenance process is affected by programmer expertise, occurrence, system documentation and the nature of the system itself. The cost of software maintenance accounts for 60% to 80% of the estimation software system charge and enhancements accounts for 70% to 85% of the maintenance effort. Various type of software maintenance are curative maintenance production with bugs corrections, adaptive maintenance concerning system varies as needs and environment change and perfective maintenance trying to recover the quality of system. Maintenance process is affected by programming skills, system documentation, experience and the behavior of the system itself.

B. Code Smell (CS)

CSs are normally not errors, they are not exactly wrong and don't presently avoid the program from functioning. This could be considered as software softness is structure that may be growing the risk of errors or faults in future. We are concerned with the succeeding code smells.

- *Long Method:* It is defined as, a function that has developed to large. The long method, not easy it becomes to read, to alternate, to maintain etc.
- *Long Parameter List:* Whenever developer produces a techniques with parameters, he should know that the larger the parameter list, the more difficult [5] it becomes to maintain this technique. The CS is well-defined as various parameters passed into a technique, this is different object oriented programming , and long parameter list method can shifted by passing an object in this place of the parameters because long parameter is not easier to read, change.
- *Empty Catch:* If programmer users the try and catch blocks sometimes they left the catch part empty with no code classified it. Either code can grip the exception, then the catch clause should not been empty, or the code can't handle the exception, then there should not be try/catch block at all.

C. Software Metrics

Software Metrics are a computable extent of software. In this paper, they are center of attention only on basis code's metrics as mentioned to in the subsequent Fig. 2.

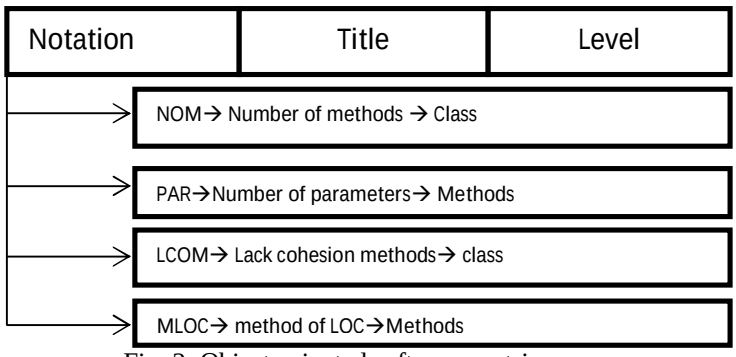

Fig. 2: Object-oriented software metric

D. Thresholds for Software Metrics

Detection rules for CSs are frequently defined in the terms of metric categories or classifications. An illustration can be: "distinguish classes that have lower dependability" or "classify methods that have a high difficulty". We want to obtain thresholds in a method that can be semantically mapped to these easy necessities, to find out what 'LOW' unity or 'HIGH' difficulty means in terms of the metrics, the unity and difficulty of the software is measured [6].In normal, thresholds may discriminate values. In case threshold gives higher bound, the values that are greater than a threshold value are measured to be difficulties. Further, the values that are lower are measured to be suitable. Thus, by incorporating threshold a simple analysis of considered values is probable. For the understanding of software metrics thresholds are essential.For example, suppose a metric *mat* that considers the size of an individual *xx*. Then a threshold *th*can be used to determine if *xx* is too huge:.

$mat(xx) > th \Rightarrow xx$ is too huge …….. (1)

Although the overhead isillustrated about the threshold used as a higher bound, it might as well know a low bound. For clarity, let thresholds are always higher bound. Though, there is no limit as low bound, it can be transformed into higher bounds. Suppose *mat* is a metric with the threshold *th*that offers a low bound, i.e., individuals *xx* are measured to be difficulties if *mat(xx)<th,* , which is equal to *1/mat(xx) >1/th*if *mat(xx)* and *th*are non-negative, metrics and thresholds normally are. By giving a novel metric *mat'(xx) = 1/mat(xx)* and a novel threshold *th'* = 1/*th*a novel metric with the reverse order is defined and with *th'* a threshold is gained that gives a higher bound. By reversing the metric, its scale is changed. To transform a low bound into higher bound while keeping its scale to minus, the metric for maximum value is used

II.RELATED WORK

In [7], Abilio, et al. studied the similar issues in numerous languages. These methods can be used to build software product lines. However, characteristic-oriented programming is a major method to offer with the modularization on characteristics in software product line. In another work, Wang, et al. presented a platform specific code smell aware system i.e. based on an abstract syntax tree and XML in [8]. Programming patters of PSCSs are defined in a formal way using abstract syntax tree sequence represented in XML. In [9], Francesa et al. proposed a data driven technique to derive threshold values for metric code, which can be used for developing detection rules for code smells. In a similar work, Arcelli et al. proposed a technique that is apparent, repeatable and allows the extraction of thresholds that respect the statistical properties of the metric.

In [10], Aiko et al. summarized the most relevant findings and discussed a series of lessons studied from calculating this study, and converses avenues for novel research in the field of CSs in [10]. Further in [11], Kim, et al. considered scheme in the reduction pattern table and modification in real, applyingan tree structure. Tree Pattern matching reducer was used to calculate patter more efficiently while its round in top to down method. However, the matching technique needs the investigating time to search pattern less than the string pattern matching techniques of acknowledgement.

Above mentioned literature strongly acknowledges the need of a unique technique for CS which could solve all the raised issues and problem. Further, such technique should provide a simplistic way for easing the software maintenance process. Hence, after analyzing the aforementioned

literature, a unique a unique technique is designed to identify CSs. For this purpose, various object-oriented programming (OOPs)-based-metrics with their maintainability index are used. Further, code refactoring and optimization technique is applied to obtain low maintainability Index.

III.SOFTWARE ARCHITECTURE RISK BASED DETECTION TOOL

In this division, we converse the CSs detection tool Visual Studio which is based on the risk based concept. The detection methodology depends on evaluating the code line by keeping words. In case the code is method statement, the program will investigate for Long Method and Long parameter List, then the program runs to check each line in the particular code to find any message chain or Empty Chain [12]. Fig. 3 shows the user interface of tool which subsequently gives concise description. In the upper grey area, there are 2 options, the first is used to project and the other is used to show CSs. Now click to project option, select upload your project file. Upload the three types of project C++, Java and C#.net. To select the file name in D-drive name is ECC.sln. The loading all files in C++, java and C#.net is done for training section. The Table I shows various types of CS detection tools and their explanation.

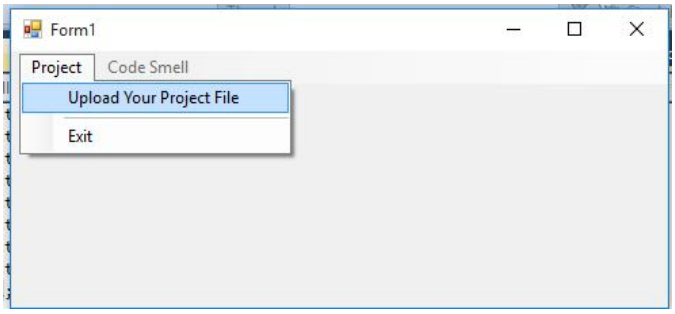

Fig. 1: Upload the Project File

IV. COMPARISON TOOLS

In this section, we evaluate some CSs tools each of them have some dissimilar features.

A. Clock Sharp

Clock Sharp is a code organizer tool for C# Programming language integrated with visual Studio 2008 and 2010.It checks code using more than 100 programming rules and can be executed as command line tool.

B. Find Bugs

Find Bugs is an open source plan works on java byte code appear for bugs in java code using [14] still study to identify four likely types of errors scariest and disturbing, of concern.

C. PMD (Programming Mistake Detector)

Source code analyzer is tool that identifies troubles in various types class: bugs such as Copied or pasted code, Duplicate code, empty try, empty catch, empty finally, empty switch, dead code, parameters and private methods, string usage, string buffer usage,inefficient overcomplicated terminology, Sub optimal code, vacant local variables, Dead code, avoidable statements, for and while statements [13].

Table I: Various CS detection tools

Code Smell	**Definition**	**Variable used**	**Results**
Long Method	An extended and composite method is categorized into dummy and well-defined methods with refactoring methods like extract techniques. As a rule, the extracted novel techniques are called within the existing one in the original position; thus, the abstraction does not contract the parameter list.	Cyclomatic complexity, LOC, Number Of Methods	LOC >50, no variable used, CC > 50. Source code divided into classes & methods is uploaded according to syntax tree. OOPs metrics are calculated and compared with rules & threshold value. Result occurred in rule wise. No. of method used = 99 &No. of long method = 21.
Long Parameter List	CS is defined as many constraints passed into a method, which is different in object-oriented, and larger parameter list method.Can restore momentary by an object as substitute of parameters as long parameter list technique is difficult to read & modify.	Number of Parameter, ∑n parameter of a method, Average Parameter,	NOP > 7, ∑n parameter of a method = 148 , M in C = 88 , average parameter = 3 and no. of parameter > average parameter. Detection method is same applying only object oriented metrics are different.
Large Classes	Large classes to advance their intelligibility and preserve, large classes are categorized into smaller ones, each for a single dependability.	Lines of Codes, Instance Variable, Depth of Inheritance, Coupling	LOC > 300, long method > 5, used instance of variable id >15 &methods > 10. Depth of inheritance means" greater extent from the knot to root of diagram", DIP> 3 and coupling >10.
Dead Code	Dead code means, remove code that isn't organism used. That's why we have source control systems.	Unused Block of data	Unused Block of data is totally used is 24.

Lazy Class	Lazy classes should predominantly request information from exacting source. Each additional class enhances the complexity of a scheme.	Number of techniques or weight, LOC	Several of method ==0, LOC<=300 and weighted method count or no. of method <=2.
Lazy Catch Block	Discover the empty catch block, comparing number to threshold	Number of Unused catch block	Total number of unused catch block = 5.
Duplicate Code	Duplicate Code exists if more brief code exists that explains the same functionality like blocked repeated	Number of Duplicate code block	Total number of Duplicate code block is 19.

Table II shows various types of comparison tools and Table II depicts comparison of detection methods used.

Table II: Various comparison tools

Comparison Criteria	Developed Software	Clock Sharp	Find Bugs	Programming Mistake Detector
Tool Description	Standalone	Plug- in Tool	Stand alone	Plug-in Tool
Threshold	Fixed Threshold value	No threshold value	No threshold value	No threshold Value
Smell Filtration	Can view all error module wise	View all the errors at the output	View all the errors at the output	View all the errors at the output
Can work on project / language	C++,java and .net	C#	Java	Java
User Interface	User friendly	Not user friendly	User friendly	User friendly
Results	Represented in graphics	Is too long to read	Can be filter by classes, packages	Not true error
Time consumption	Less	-	-	-

Table III: Comparison of detection methods used

Code Smell Methods	ECC System (Yes/No)	Movie Rental Program (Yes/No)	Electricity calculating program (Yes/No)	Another Cryptography System (Yes/No)
Long Method	Yes	Yes	Yes	Yes
Long Parameter List	Yes	Yes	No	Yes
Large Classes	Yes	No	NO	No
Dead Code	Yes	No	No	No
Lazy Class	Yes	No	Yes	No
Lazy Catch Blocks	Yes	No	No	No
Duplicate code	Yes	No	No	No
Switch Statement	Yes	Yes	Yes	No
Temporary Field	Yes	No	No	No
Comment Lines	Yes	No	No	No

V.SIMULATION MODEL

In our research work, source code is written in C++, java and C# (object oriented language). At once, only one language is detected for code like we can select C# code. All methods are applied and tested in c# language code or object oriented programming language.In Fig.4, the case study program is used for ECC system using c# / object-oriented Program. An error in all classes is detected using CS detector for code samples as Admin.cs and Adminlog.cs and etc. Visual studio is the tool used for evaluating the code. Bad smells would be detected using plug-in withvisual studio. Software metrics plug-in would be applied on source code to calculate the metrics values for analysis and measure the quality of source code. Refactoring techniques are applied to remove the detected bad smellsusing “visual tool”. Then again metrics plug-in is applied to re-calculate the metrics values. Finally, the simplify/test cycle is repeated until the smell is gone “without varying its bordering performance”.

Various metrics for refactoring:

- Total lines of code
- Several of packages
- Method lines of code
- Numerous of classes
- Several of attributes
- Cyclomatic complexity

- Number of children
- Coupling
- Cohesion
- Complexity of inheritance tree

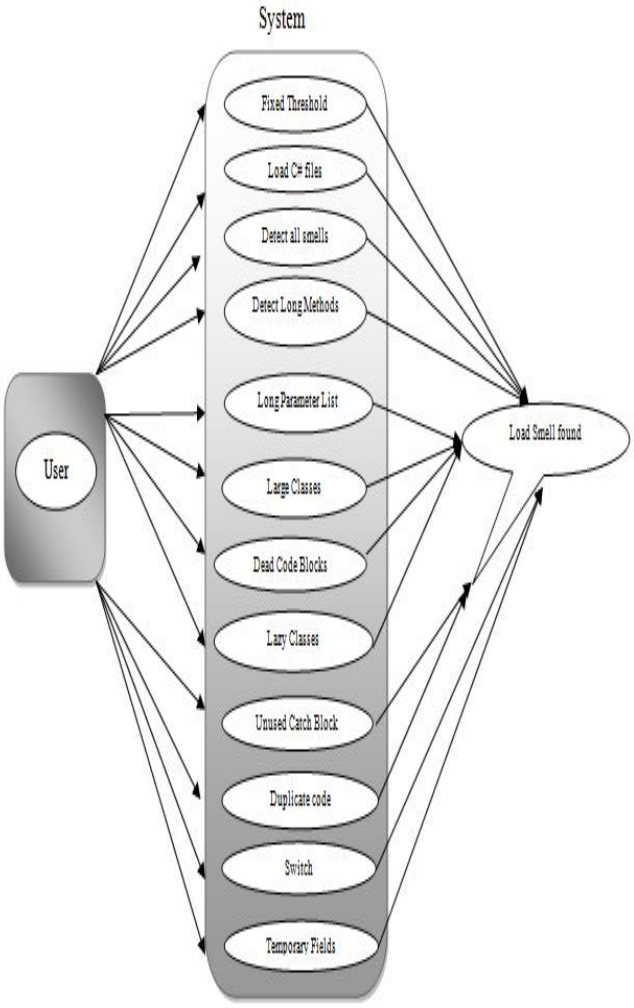

Fig. 4: Flow chart of proposed work

A. Software Specification

Source code of a project in any language (C#, C++, java) is required to calculate the quality using software metrics. The tool used to run the source code is required for e.g,visual studio and its plug-in. DEODORANT named plug-in is used to detect the bad smells in code. Metrics 1.3.6 is used to calculate the metrics values.

B. Hardware Specification

To determine the size, portable Coordinate Measurement Machine is used to reach around the surface geometry of your physical model, or part. Size of digitizers may have restrictions, although this can frequently be dealt with by using the leap frog article which can be purchased as part of the refactoring eclipse plug-in. Finally, conclude what accuracy tolerance is required when refactoring of the physical model, model or part. It is always greatest to use a computer with a high end illustrations card, with high end memory resources.

C. Significant Research work

The details are discussed as below.

- *Maintain Ability:*It is simple to attach errors since the initial code is simple to read and easy to grip. This capacity is completed by dropping large uniform routines into a set of separately concise, well-named, single resolve method. It powers by moving a method to a more suitable class, or by removing ambiguous explanation.
- *Extensibility:*It is straightforward to range the capacity of the request if it uses recognizable structure patterns, and it offer some where none before may have existed. Because of frequent changes of the source code its arrangement can be easily customized. Therefore, it becomes very hard to reorganize the code and make its design inclusive.Correction makes software easier to understand.If it is not well considered, software is very hard to appreciate, particularly in a few months' time. Applying refactoring as untimely as possible during the software life-cycle can recover the feature of intend and reduce the complexity and cost in successive development phases.
- *Documentation:* Refactoring shows an important role. It is a great technique if documentation to an older device cannot be studied. One may need to know and appreciate the inner works of the device in order to develop maintenance instructions, create an improved example or to replace incomplete or out-dated certification.
- *Complexity:*The complexity of the project is analysed and calculated so as to understand the scalability of the project.
- *Code smells (CSs):* Various types of code smells are generated using the refactoring.
- *K-mean Clustering:*This is a method of quantized vector, initially from signal processing i.e., famous for classified analysis in data mining. K-means clustering destination is used to divide m explanations into k clusters in which each explanation belongs to cluster with the neighbour mean, serving as rules of cluster. These consequences in a division of data space into small cells.
- *Optimization Techniques (GA)*: This is a technique used to resolve both reserved and unreserved reduction difficulties based on nature's initial process i.e. biological evolution. This algorithm repeatedly modifies a random population of individual solution. At each step genetic technique randomly selects individuals from recent population and uses them as parents to produce children for the next generation.

VI.CASE STUDY

The case study is full for recognition of bad smells in the Elliptic Curve Cryptography system in (.net, c++ and java) object oriented language. The many bad smells are distinguished in the ECC

system source code using graphical user interface application developed. The following metrics in .net are implemented to find out the methods of bad smells in the source code. Case Study in Various methods likes Long Methods, Long Parameter list, Large Classes, Dead Code Blocks, Lazy Classes, Unused Catch Block, Duplicate code, Switch and Temporary Fields.

A. How to check long method?

There are numerous different CSs, but long method is one of the mainly general and simply corrected method. A larger method is some technique that is so extended it is hard to appreciate at a fleeting look. Diverse entity programmers will have dissimilar opinion about how long is too extended, and here is a single rule that would relate in all cases. Though, in universal you should prefer methods that are shorter to those that are longer, technique that do only one object and methods whose lengths permit them to be view on a single screen in their total. Result obtained by long methods in your project are actually attractive easy to do using visual studio analysis tools. In visual studio 2010, while you have the project you desire to Longmethod.cs open, click "TEST_CODE" then"estimateCode Metrics for[Longmethod.cs]."

B.How to check Dead Code Blocks and Why to remove dead code?

It can be inaccessible code, unnecessary code, or unused code. Using the code analysis characteristic of visual studio we can find it. The following are possible reasons to remove dead code:

- At times we misuse a lot of time thoughts why a breakpoint does not hit a method/class.
- To add to the code coverage result.
- Code maintainability.
- Recover performance.

Pseudo Code of Long Method

```
Initialize the variables LocI=0, CCI=0, HALI=0,ci, datatype, x=0,count=0,s,semicolon and loc=0;
for (ci=0;ci <methods.Items.Count;ci++)

try
string[] data type = new string[] { " string ", " String ", " int ", " Int16 ", " Int32 ", " Int64 ", " float ", " double ", " Double ", " Single ", " char ", " Char " };
for (int i = 0; i < array. Length; i++)
if statement (array[i] == ';')                    //to check the end of the lines through semicolon (LOC)
```

```
loc++;
 end
end
if (vari. Contains(','))                    // to find the
colons
string[] variables = vari. Split(',');
for (int j = 0; j < variables. Length; j++)
  if    (s.Contains(variables[j]    +    "    =")    ||
s.Contains(variables[j]      +      "      <=")      ||
s.Contains(variables[j]      +      "      >=")      ||
s.Contains(variables[j]      +      "      ==")      ||
s.Contains(variables[j] + " +="))

     end
 else

   if  condition (loc>= 50)
   if condition (count == 0)
LOClongmethods     [locI++]     =     methods.
Items[ci].ToString();
    count++;
        end
end
else
if (s.Contains(vari + " =") || s.Contains(vari + "
<=")    ||    s.Contains(vari    +    "    >=")    ||
s.Contains(vari + " ==") || s.Contains(vari + "
+="))

end
    else
          if condition  (count == 0)

LOClongmethods[locI++]        =        methods.
Items[ci].ToString();
                                    count++;

                    end
end
```

To start adding rules to the Deadcode.cs rule set, you can investigate for a rule using either the rule number or its name, as shown below. You can also simply increase the rule category and select the rules that you are concerned in.All the dead code exposure rules are part of a particular rule set that make it much easier to direct.

Fig. 5 shows that the larger method is any technique that is so larger which is complex to understand at a glance.But long a method is one of the most widespread and give simply corrected CSs. To detect the CS using long method is 2, number of long parameter list (LPL) = 2, no. of large classes =2, no. of dead code blocks =70, no. of lazy classes=3, unused catch block=0, duplicate code=9 like code clone, switch =0 and last one of the least temporary field =16.

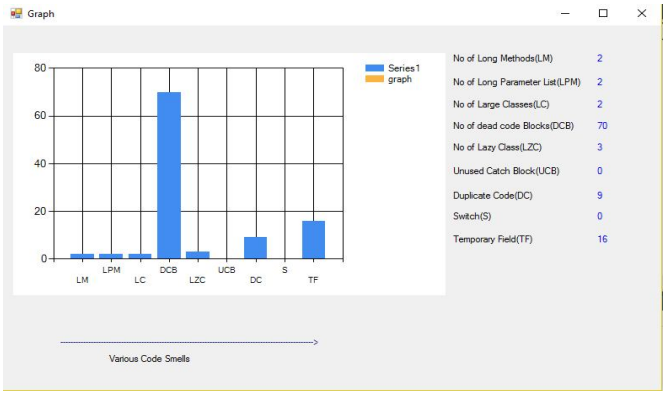

Fig 5: Correction to Find Detect Methods (Before)

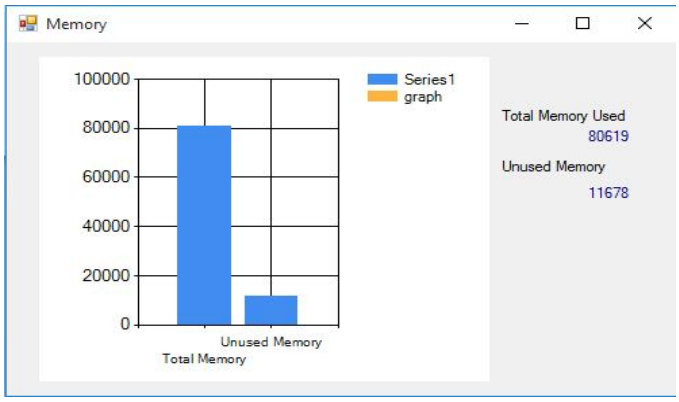

Fig. 6: Correction to Memory Used (Before)

Fig. 6 shows that, the memory used to find in two categories total memory and unused memory. Total memory value used is =80619 and unused memory value used is = 11678.

Fig. 7: Correction to Find Detect Methods (After)

Fig. 7 shows that, to fresh up code smells, one must re-factor. Refactoring is the procedure of humanizing the superiority of the program without altering its exterior behavior. In the case of the long method smell, the majority widespread way to re-factor is to remove methods from the long method. In universal, the remove method refactoring is one that can typically be done with the support of built-in tools in visual studio. To detect the code smell no. of long method = 0, no. of long parameter list =0, no. of large classes=1, no. of dead code blocks=0, no. of lazy classes = 1, unused catch blocks=0, duplicate code value is 0,switch value is 0 and temporary field value is 0.

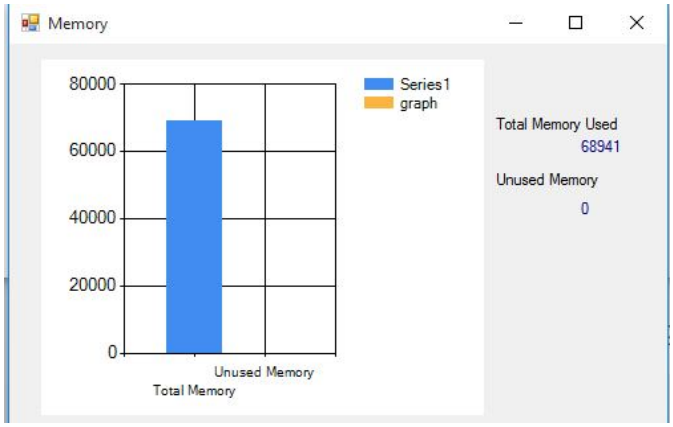

Fig. 8: Correction to Memory Used (After)

Fig. 8 shows that in this way, the technique can be broken up in to a compilation of smaller, more unified methods. Total Memory value used is 68941 and Unused Memory value used is 0.

VII. CONCLUSION AND FUTURE SCOPE

In this paper, we have proposed a unique CS detection scheme. The scheme is evaluated using various parameters for a case study of ECC system. Various code smells were detected in the ECC system source code using graphical user interface application developed. The calculated

object oriented metrics shows the value of each metric in their respective CSs detected on the coding. The objective of this paper was not to evaluate the implements, but to explain our knowledge in using them and difficulties related to its evaluation task. Linear regression analysis was used in which all of the smells were examined in the similar mode.In this paper, a tool for detecting CSsis proposed to deal with the threat concept. As a verification of concept, an automatic risk based code smells detection tool was developed. The tool was used to recognize problems in a C# case study. Various CSs have been detected in the case study. Total memory used and unused memory (before and after refactoring) was also calculated. Moreover, risk factor level has been qualitatively related (high, low, medium) with each CS based on the rate of occurrence and rigorousness.

In future, we plan to expand our developed software to sense other CSs and test the tool using larger case study. Further, developer based experiment to duplicate Mantyla's developer study and an investigation of the testing implication of smell suppression is also in scope of this problem.

REFERENCES

[1] Hazelwood, Kim, and Michael D. Smith. "Generational cache management of code traces in dynamic optimization systems." *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*.IEEE Computer Society, 2003.

[2] van Emden, Eva, and Leon Moonen."Assuring software quality by code smell detection." *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012.

[3] Van Emden, Eva, and Leon Moonen."Java quality assurance by detecting code smells." *Reverse Engineering, 2002.Proceedings.Ninth Working Conference on*.IEEE, 2002.

[4] Palomba, Fabio. "Textual analysis for code smell detection." *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015.

[5] Nguyen, Hung Viet, et al. "Detection of embedded code smells in dynamic web applications." *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*.IEEE, 2012.

[6] Herbold, Steffen, Jens Grabowski, and Stephan Waack. "Calculation and optimization of thresholds for sets of software metrics." *Empirical Software Engineering* 16.6 (2011): 812-841.

[7] Abilio, Ramon, et al. "Detecting Code Smells in Software Product Lines--An Exploratory Study." *Information Technology-New Generations (ITNG), 2015 12th International Conference on*.IEEE, 2015.

[8] Wang, Chunyan, et al. "A platform-specific code smell alert system for high performance computing applications." *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014.

[9] Fontana, Francesca Arcelli, et al. "Automatic metric thresholds derivation for code smell detection." *Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics*. IEEE Press, 2015.

[10] Yamashita, Aiko. "How good are code smells for evaluating software maintainability? results from a comparative case study." *2013 IEEE International Conference on Software Maintenance*.IEEE, 2013.

[11] Kim, Jungsook, and Seman Oh. "EM-code optimization algorithm using tree pattern matching." *Information, Communications and Signal Processing, 1997.ICICS., Proceedings of 1997 International Conference on*. IEEE, 1997.

[12] Ramos Conceicao, Carlos Fabio, Glauco de FigueiredoCarneiro, and Brito E. Abreu."Streamlining Code Smells: Using Collective Intelligence and Visualization." *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*.IEEE, 2014.

[13] Kessentini, Wael, et al. "A cooperative parallel search-based software engineering approach for code-smells detection." *Software Engineering, IEEE Transactions on* 40.9 (2014): 841-861.

[14] Ito, Yu, et al. "A Method for Detecting Bad Smells and ITS Application to Software Engineering Education." *Advanced Applied Informatics (IIAIAAI), 2014 IIAI 3rd International Conference on*. IEEE, 2014