An Empirical Study of Bad Smell in Code on Maintenance Effort

Rohit Kumar
Chandigarh Engineering College, Mohali, (Punjab), India
Email Id: er.rkkansal@gmail.com

Jaspreet Singh
Assistant Professor, Chandigarh Engineering College Mohali, (Punjab), India
Email Id:cec.jaspreet@gmail.com

Amandeep Kaur
Assistant Professor, Chandigarh Engineering College Mohali, (Punjab), India
Email Id:cecm.infotech.amandeep@gmail.com

***ABSTRACT* - In this paper, we represent an automated code smell detection and refactoring tool for calculating risk factor by detecting Code Smells and decrease risk factor by Refactoring Techniques. Refactoring is a process for restructuring or improving internal structure of software without changing its behavior. A new code smell (Lazy Catch) detection is also presented. To achieve this aim Declarative Programming approach is followed along with object-oriented software metrics. Detection of Code Smells is based on various Facts and Rules. We used this tool to detect the bad smells in oops based case studies such as (C#, CPP, Java). That means this tool is independent of any language. Risk Factor level is represented in three categories (Hi, Low, Medium).**

***KEYWORDS:* -** Code Smells, Risk Factor, Detection of Code Smell, Refactoring, Object Oriented 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 [1]; but such efficiencies come at the cost of all-encompassing observation, a characteristic that is produce a humanity that "never forgets". Software system must be flexible for extends. In the object oriented programming, it is effective way for this need to use the design patterns. The design patterns may make unnecessary give, and may give more difficulty to design software [2]. Therefore, future that use of design patterns in top process should be limited to special cases. Software system should be flexible however, unnecessary agility lower quality of software system. It is significant to constantly keep up with software maintenance process as it helps develop the system to execute to its best capability and to work suitably in line with the user's point [3]. A process can be explain as a development of the software's defects density or expansion of the software that leads to its efficient and suitable function within the system's environment.

Code smells have been defined [5] as sign of poor plan and execution choices. In some cases, such sign may invent by activities performed by developers while in a speed, e.g., implement urgent patch or simply making suboptimal choices. In other cases, smells come from some returning, poor design solution, also known as anti-patterns. *Refactoring* is a reasonable way to solve this dilemma. [4] Proposed that "Refactoring is the procedure of varying a software system in such a way that it does not modify the external activities of the code yet advance its internal structure". In order to extend a high quality software system, it is vital to change software systems into design patterns based as essential through refactoring.

Refactoring process consists of various activities:

1.	Recognize where the software should be refectories.
2.	Establish which refactoring(s) should be applied to the recognized places.
3.	Agreement that the applied refactoring preserves behavior and Apply the refactoring [5].
4.	Evaluate the consequence of the refactoring on quality distinctiveness of the software (e.g., maintainability ,difficulty, understandability) / the procedure (e.g., efficiency, rate, effort);
5.	Continue the reliability between the refectories program code and other software object (such as documents, propose documents, necessities specifications, tests and so on).

Detected code smells will differ depending on the preferred likelihood threshold. 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 as regards the occurrence of code smells. It will also be up to the developer to choose on the sufficiency of relating a given refactoring to eliminate a detected code smell [6].

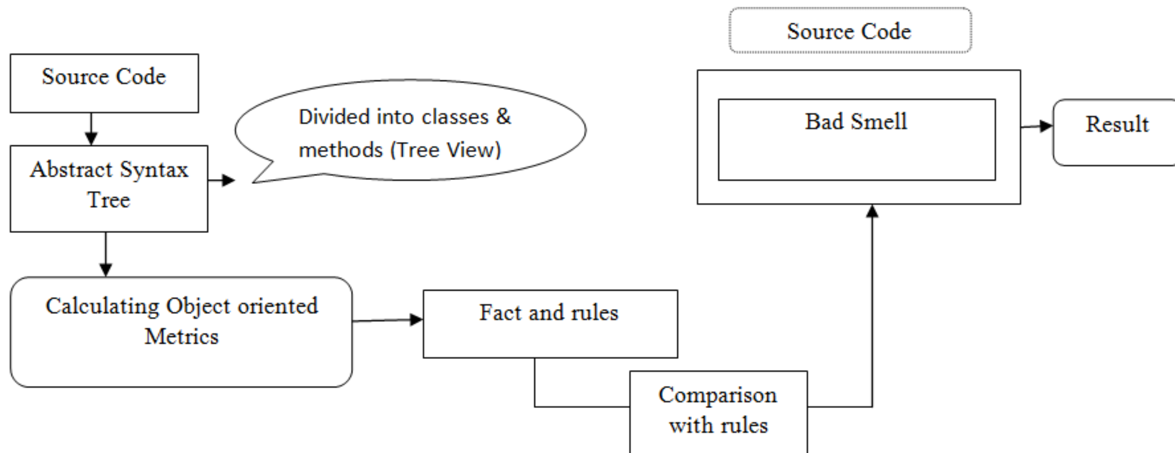

Figure no: 1 Overview of Detection Method

II. BACKGROUND

This section gives procedural background to software maintenance process; code smell and software metrics, threshold for software metrics and risk assessment.

2.1 *Software Maintenance*

Software engineering is the alteration of a software produce after delivery to correct mistakes, to modify performance or other aspects. They must study how a plan functions before they can change it. They often interrelate with complex and hard to understand systems [4]. Maintenance process is affected by programmer expertise, occurrence, system documentation and the nature of the system itself.

2.2 *Code Smell*

Code smells are individuality of software that may specify a code or design difficulty and can make software hard to evolve and maintain. To delete code smells and thus better maintainability and software progress we have to apply refactoring steps to improve the inner quality of software [6]. It's not essentially all the code smells have to be removed, it depends on the system, now and then the smell cannot be removed, it is the best solution; a typical example is given, in certain cases, by the code smell Large Class.[5]Detecting smells is durable and costly. Hence when they have to be detached, it is better to remove them as early as possible. Tool maintain for their detection is particularly useful, since many code smells can go ignored while programmers are working. The dissimilar detection methods used by the detection tools are typically based on the computation of a particular set of shared metrics, or standard object oriented metrics, or metrics defined ad hoc for the smell detection reason.

2.3 *Software Metrics*

Software Metrics are a quantitative extent of software. In this article, we center of attention only on source code's metrics as referred to in the following Figure 2 [6]:

Notation	Title	Level
NOM	Number of Methods	Class
PAR	Number of Parameters	Methods
LCOM	Lack of Cohesion of Methods	Class
MLOC	Method Lines of Code	Methods
WMC	Weighted Methods per _Class	Class

Figure no: 2 Object-Oriented Software Metric [6]

2.4 Thresholds for Software Metrics

Detection rules for code smells are frequently defined in the terms of metric categories or classifications. An instance can be: “distinguish classes that have lower reliability” or “Classify methods that have a HIGH difficulty”. We want to obtain thresholds in a method that they can be semantically mapped to these easy necessities, to find out what LOW unity or HIGH difficulty means in terms of the metrics we use to measure the unity and difficulty of the software [3].

2.5 Thresholds effects analysis

The threshold values are considered with the help of Value of Acceptable Risk Level using equation (1).Only above reveal metrics are calculated with this formula. Table 3 shows the threshold values of preferred metrics at different five risk levels of two different versions of jfreechart.

$VARL = po^{1}(p_{o}) = 1 \,/\, \beta(\log{(p_{o}/1 - p_{o})} - \alpha).$ Equation (1)

The Threshold values of selected metrics are given with the value acceptable risk level formula, in which α and β are the coefficient estimates and the probability p_o is suggested with different five risk levels i.e. (po = 0.5 to po = 0.7). Threshold values with Equation 1 based on bad smell at diverse risk levels. Consequence shows some metrics have effective threshold values for the metrics. We will use the fix threshold value.

2.6 Risk Assessment

To proposal the advance for a software system, the project manager should review the risks [8] opposite the development attempt. There are several risk assessments methods so; they necessary a human involvement depends on system attribute according to the system models it. Risk assessments are an exceptionally important part in the administration development process.

III. RELATED WORK

This section evaluates before published studies on the effect of code smells. An organized literature review on code smells and refactoring covered papers available by IEEE and sci-index software engineering journals and Transaction from 2012 to 2014. That review found that Danphitsanuphan et.al, 2014 an advance for detecting the so called bad smells in software recognized as Code Smell. In allowing for software bad smells, object-oriented software metrics were used to detect the source code whereby Eclipse Plugging were developed for detecting in which location of Java source code the bad smell show so that refactored the software could then take place. The detected source code was classified into 7 types: Long Method, Parallel Inheritance Hierarchy, Large Class, Long Parameter List, Lazy Class, Switch Statement, and Data Class. Francesca Arcelli, et al., 2015 Code smells are structural uniqueness of software that may specify a code or plan difficulty that makes software hard to progress and maintain, and may activate refactoring of code. A current research is active in dining automatic detection tools to help human in ending smells when code size becomes impossible for manual review. Since the dentitions of code smells are casual and biased, assessing how ejective code smell detection tools are is both main and hard to achieve. Dag I.K et,al, 2012 This paper examine the connection between code smells and protection effort. Six developers were hired to execute three maintenance tasks each on four functionally corresponding Java system initially implement by different companies. Each developer spent three to four weeks. In total, they customized 298 Java files in the four systems. An Eclipse IDE plug-in measured the accurate amount of time a developer used up maintaining each file. Regression analysis was used to give details the effort using file property, including the number of smells.

IV. SOFTWARE ARCHITECTURE RISK BASED DETECTION TOOL

In this division, we converse the code smells detection tool Visual Studio which is based on the risk based concept. The detection methodology depends on evaluating the code line by kept word, in case that the code is method statement so 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.

Figure no (3) shows the tool of user interface. The subsequent gives concise description of the user interface:

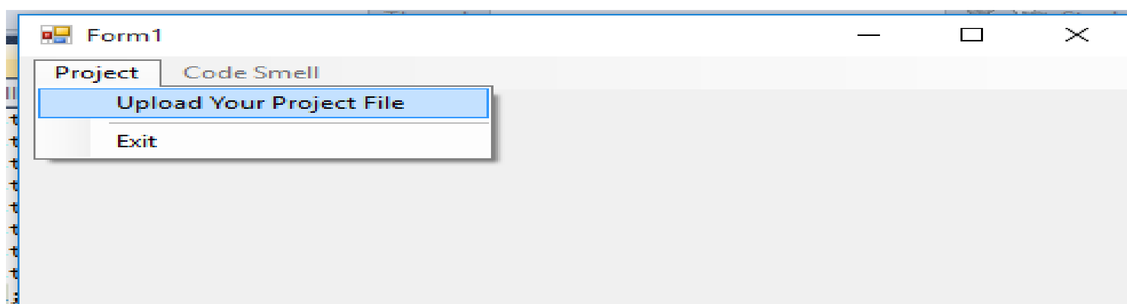

Figure no: 3 upload the project file

Above figure shows in the upper grey area, there are 2 options the first is used to Project and the other is used to show code smells. 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 Banking.sln. The loading all files in C++, java and C#.net for training section.

Table no: 1 Risk Based Detection Tools

Code Smell	Definition	Variable used	Results
Long Method	An extended and composite method is divided into dummy and well-named methods with refactoring rules like extract method. An explanation some parts of the method may be extracted as new techniques. As a rule the extracted new techniques are called within the old one in the original position; thus, the extraction does not shorten the parameter list.	• Cyclomatic complexity • LOC • Number Of Methods	The numbers of lines of code greater than 50 and variable are not used. In this method used Cyclomatic complexity (CC) > 50. Uploading source code divided into classes and methods. According to abstract syntax tree. Calculated object – oriented matrices: Number of line of code in method, total number of variable, used variable, unused variable, Cyclomatic complexity and Halstead efforts. Compare these metrics with detection rules and threshold value. Result occurred in rule wise. Number of method used = 99 and Number of Long Method = 21.
Long Parameter List	If the developer makes a method with parameters, he should know that the longer the parameters list, the more composite it becomes to maintain this method. This code smell is defined as many constraints passed into a method, this is different in object-oriented, and long parameter list method can be restore by passing an object as a substitute of the parameters because long parameter list technique is difficult to read and modify.	• Number of Parameter • ∑n parameter of a method • Average Parameter	Basically the rule of this method number of limitation is greater than 7 (NOP>7), ∑n parameter of a method = 148 , M for all method in C = 88 , average parameter = 3 and then number of parameter are greater than avg_parameter and some of parameters are not used. 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	Number of lines of code in class LOC > 300, long method > 5 , used instance of variable id greater than 15 and methods are greater than 10. Depth of inheritance means" the greater extent from the node to the root of the tree", DIP> 3 and coupling is greater than 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 particularly requesting information from exacting source their weight. Each additional class enhances the complexity of a project.	• Number of methods or weight • Lines of code	Number 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 the threshold	• Number of Unused catch block	Total number of unused catch block = 5.
Duplicate Code	Duplicate Code is duplicate 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.
Switch Statement	Switch statement contain one or more switch sections. Each switch section contains one or more case label followed by one or more statements.	• Number of cases • Default case • Complexity	In this statement used number of cases greater than 10, without default case and complexity greater than equal to 10 i.e complexity>=10.
Temporary Field	If you're temporary an object as a parameter to a technique, make sure that you're using all of it and not most desirable single fields.	• Temporary Field	Temporary field means variables are not used and total temporary field is 30.
Comment Line	Comment lines means the aim of making the resource code easy to understand, and are usually unnoticed by compilers and line by line check code.	• Actual Source Code	Comment lines more used greater than 33% of actual source code and classes = 2.

V. COMPARISON TOOLS

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

5.1 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, it can be executed as command line tool.

5.2 Find Bugs

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

5.3 PMD (PROGRAMMING MISTAKE DETECTOR)

Source code analyzer that identifies troubles in five 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.

Table no: 2 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
Memory Released	Yes	-	-	-

Table 3: Comparison of Detection Methods used

Code Smell Methods	Banking System (Yes/No)	Movie Rental Program (Yes/No)	Electricity calculating program(Yes/No)	Another Banking 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

VI. SIMULATION MODEL

In our research work, we are source code would be choose in C++, java and C# (object oriented language). Basically we are source code would be choose in C++, java and C# (object oriented language). At once detect only one language code like we can select C# code. All methods are applying and testing c# language code or you can say object oriented programming language.

The case study program is banking system c# / object-oriented Program. We will detect an error in all classes using Code Smell detector, the code samples as Admin.cs and Adminlog.cs and etc. Visual Studio is the tool used for analyzing the code. Bad smells would be detected using plug in –with-Visual 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. Apply applicable refactoring techniques to remove that detected bad smells. Refactoring using “Visual Tool”. Then again apply Metrics plug-in to re-calculate the metrics values. Repeat the simplify/test cycle until the smell is gone “without varying its peripheral performance”.

Various metrics for refactoring:

- Total lines of code
- Number of packages
- Method lines of code
- Number of classes

- Number of attributes
- Cyclomatic complexity
- Number of children
- Coupling
- Cohesion
- Depth of inheritance tree

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 Example; Visual Studio and its plug-in. DEODORANT plug-in is used to detect the bad smells in code. Metrics 1.3.6 is used to calculate the metrics values.

Hardware Specification:

Determine what size portable Coordinate Measurement Machine is needed 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 feature which can be purchased as part of the refactoring eclipse plug-in. Conclude what accuracy tolerance is required when refactoring of the physical mold, model or part.

It is always greatest to use a computer with a high end graphics card, with high end memory resources.

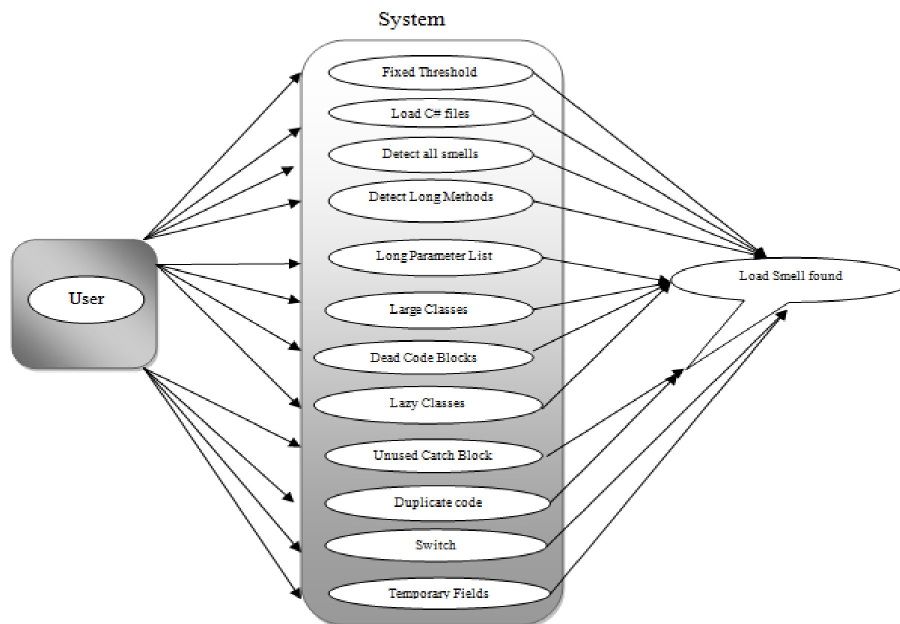

Figure no: 4 Flow chart of Research work

Significant Research work:

- **Maintainability:** It is easier to attach bugs since the source code is easy to read and the intent of its author is easy to grasp. This capacity is achieved by dropping large monolithic routines into a set of separately concise, well-named, single purpose method. It power be achieved by moving a method to a more appropriate class, or by removing misleading explanation.
- **Extensibility:** It is straightforward to extend the capacity of the application if it uses recognizable design patterns, and it provide some give 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 plays an invaluable role. It is a great procedure if documentation to an older device cannot be learned. One may need to know and understand the inner works of the device in order to develop maintenance instructions, create an improved paradigm or to replace incomplete or out-dated certification.

- **Complexity:** The complexity of the project is analyzed and calculated so as to understand the scalability of the project.
- **Code smells:** Various types of code smells are generated using the refactoring.

VII. CASE STUDY

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

I. How to check long method?

There are numerous different code smells, but long a method is one of the mainly general and simply corrected code smells. A long method is some method that is so extended it is hard to understand at a fleeting look. Diverse entity programmers will have dissimilar opinion about how long is too extended, and I don't consider there 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 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 "estimate Code Metrics for [Longmethod.cs]."

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
```

II. 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:

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

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.

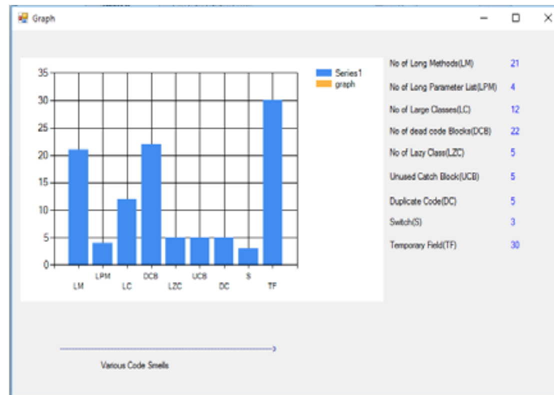

Figure no: 5 Before correction to Find Detect Methods

In this figure shows that the long method is any method that is so long it is complex to understand at a glance. But long a method is one of the most widespread and simply corrected code smells. Detect the code smell using Long Method is 21, Number of Long Parameter List (LPL) = 4, no. of large classes =12, no. of dead code blocks =22, no. of lazy classes=5, unused catch block=5, Duplicate code=5 like code clone, switch =3 and last one of the least temporary field =30.

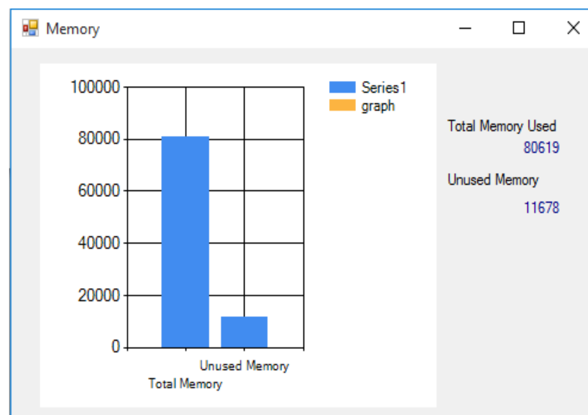

Figure no: 6 Before correction to Memory Used

Above figure 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.

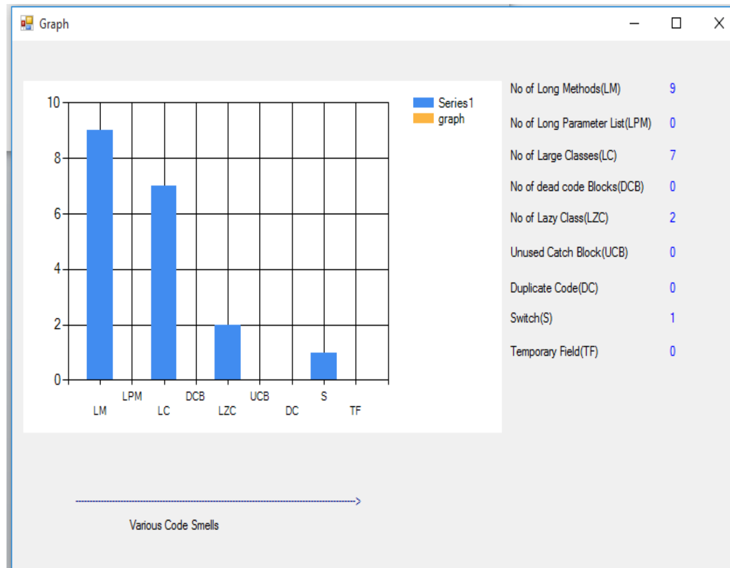

Figure no: 7 After correction to Find Detect Methods

In this figure shows that, to fresh up code smells, one must refactor. Refactoring is the process of humanizing the superiority of the code without changing its exterior behavior. In the case of the long method smell, the majority widespread way to refactor 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 = 9, no. of Long Parameter List =0, no. of Large Classes=7, no. of dead code blocks=0, no. of Lazy classes = 2, Unused Catch Blocks=0,Duplicate Code value is 0,switch value is 1 and Temporary Field value is 0.

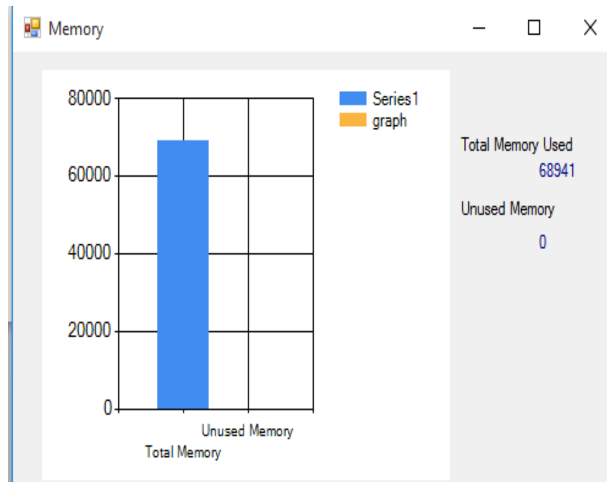

Figure no: 8 After Correction to Memory Used

In this figure 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.

VIII. CONCLUSION AND FUTURE SCOPE

At last, we have to declare that compare the tools are very difficult, and in some cases also using them are not very easy and urgent. The various code smells are detect in the banking system source code using graphical user interface application developed. The calculated object oriented metrics shows the value of each metric in their respective code smells detected on the coding. The purpose of this paper was not to evaluate the tools, but to explain our experience in using them and draw the difficulties in the comparison task. The first experiential study on the result of code smells on software maintenance effort in a prohibited industrial setting. We used multiple linear regression analysis in which all of the smells were examine in the same mode. Code smells are the mainly ordinary bad patterns connected to bad training practices which lead to deeper troubles in maintain the software. Software products that surround code smells can be difficult to maintain. In this research effort, we propose a tool for detecting code smells which uses the threat concept. As a verification of concept, we developed an automatic risk based code smells detection tool. We used the tool to recognize problems in a C#

case study. Code smells, such as Long Method, Long Parameter list, Lazy Class, Dead code and comment Line, Temporary field and Lazy Catch Block have been detected in the case study. To find total memory used and unused memory (Before and after refactoring). Moreover, risk factor level has been qualitatively related (high, low, Medium) with each code smell based on the rate of occurrence and the rigorousness of each code smell. Main focus main ability index we plan to expand our developed software to sense other code smells and test the tool using larger case study. We mainly focus on maintainability index decrease than risk factor will also decrease.

Future Work we focus only on developer based experiment to duplicate Mantyla's developer study and on an investigation of the testing implication of smell suppression. The results accessible here are the first of many smell studies and we receive further searching in this area, to enhance the maintained of software or system and different fields.

REFERENCES:

[1] Erich Gamma, John Vissides, Ralph Johnson, Richard Helm: Design Patterns Elements of Reusable Object Oriented Software, Addison Wesley Professional, 1995.

[2] Joshua Kerievsky: Refactoring to Patterns, Addison Wesley Professional, 2004.

[3] Danphitsanuphan, Phongphan, and Thanitta Suwantada. "Code Smell Detecting Tool and Code Smell-Structure Bug Relationship." Engineering and Technology (S-CET), 2012 Spring Congress on. IEEE, 2012.

[4] 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.

[5] Mens, Tom, and Tom Tourwé. "A survey of software refactoring." Software Engineering, IEEE Transactions on 30.2 (2004): 126-139.

[6] 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.

[7] Abdelmoez, Walid, Essam Kosba, and Ali Falah Iesa. "Risk-Based Code Smells Detection Tool." The International Conference on Computing Technology and Information Management (ICCTIM2014). The Society of Digital Information and Wireless Communication, 2014.

[8] Sjoberg, Dag, et al. "Quantifying the effect of code smells on maintenance effort." Software Engineering, IEEE Transactions on 39.8 (2013): 1144-1156.

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