A Code Smell Detection & Refactoring Scheme to Enhance the Performance of Elliptic Curve Digital Signature's Authenticity System

Daljit Kaur[1] Rachna Rajput[2]
[1]M. Tech. Student [2]Assistant Professor
[1,2]Department of Computer Science & Engineering
[1,2]Guru Kashi University, Talwandi Sabo, Punjab, India

Abstract— In this paper, we explain the detection of code smells in the source code along with the refactoring that code for the sake of quality improvement of source code and it is not gain saying that quality of code is hiked by the refactoring Techniques. Actually, the refactoring is a kind of procedure in which the internal structure of source code is improved but without disturbing the behavior of software. To crack the required results the Declarative Programming approach is followed along with object-oriented software metrics. Moreover, the finding of various code smells is possible only with the help of numerous Facts and Rules. We used this tool to find out the smells in one of the ECC application along with the other oops based case studies. Code maintainability index is represented in three categories (Hi, Low, Medium) that shows source code quality (Low, Hi, Medium).

Key words: Code Smells, Code Quality, Maintainability Index, Detection of Code Smell, Refactoring, Object Oriented Metrics

I. Introduction

The term “code smell” was firstly introduced by Kent Beck to define format problems in the source code of software which are examined by the experienced programmers. As written by Kent Beck:

“A code smell is a clue that there is some fault in somewhere in your code”.

A software code source becomes very difficult to maintain because it evolves with period of time. Also there designs are getting very complex and confused to learn so it is very important to reorganize the code time to time or for once. The very crucial thing is that when rearranging of code is done it must be sure that the program behaves the same as older manner as it did previously before reorganization. Semantic preserving program transformations are known as refactoring. The process of refactoring is not considered as a main front while development process because apply refactoring by hand having lots of drawbacks like error-chance and waste of time. It is very clear that the betterment of refactoring are not available for all the developers as it neither sum ups new features to the software nor improves any external software qualities. Although it is already defined that refactoring not having quality to hike the external software aspects but it has a important feature to help in betterment of internal attributes of source code of the software named reusability, maintainability and readability. It is very challenging for the developers to find that which part of source code is profitable from refactoring even if they have very good grades in the study of refactoring.

Elliptical curve cryptography (ECC) is a public key encryption technique based on elliptic curve theory that can be used to create faster, smaller, and more efficient cryptographic keys. ECC generates keys through the properties of the elliptic curve equation instead of the traditional method of generation as the product of very large prime numbers. The technology can be used in conjunction with most public key encryption methods, such as RSA, and Diffie-Hellman. According to some researchers, ECC can yield a level of security with a 164-bit key that other systems require a 1,024-bit key to achieve. Because ECC helps to establish equivalent security with lower computing power and battery resource usage, it is becoming widely used for mobile applications.

II. Literature Survey

All the researches on bad smell detection originate from the description of bad smells. Fowler defined the different 22 bad smells and also provided refactoring operations and methods to remove them. Author named karnam sreenu (2012) defined that software refactoring is a technique that transforms the distinct software artifacts to improve the software internal structure without affecting external nature. Also it defined object oriented metrics can be calculated to detect the bad smell. In addition to metric-based approaches, there are some other peculiar approaches to detect bad smells.[2]

N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur (2010) proposed Décor, which uses a Domain-Specific Language (DSL) for specifying smells in high abstraction level, where software engineers can manually define the specification for bad smells detection using the taxonomy and vocabulary, and, if needed, the context of the analyzed systems.[18]

Marija Katic, explains the main definitions and terms concerning software redesign is closely connected with the testing. Research explains briefly presents the software redesign process and methods that are used in that process. Although one can say that for example a source code redesign belongs to the implementation phase, tests are needed to ensure that the behavior is not changed.[4]

Francesca Arcelli, Font ana Pietro Braione, Ma rco Zanonia (2011), discovered that Code smells are structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and may trigger refactoring of code. Recent research is active in defining automatic detection tools to help humans in finding smells when code size becomes unmanageable for manual review.[5]

Bart Du Bois and Serge Demeyer, Jan Verelst (2015), Research analyzes how refactoring manipulate coupling and cohesion characteristics, and how to identify refactoring opportunities that improve these characteristics. Coupling and cohesion on the other hand are quality attributes which are generally recognized as being among the most likely quantifiable indicators for software maintainability.[9]

Almas Hamid, Muhammad Ilyas, Muhammad Hummayun and Asad Nawaz (2013) discuss that Refactoring is a technique to make a program's source code more readable, efficient and maintainable. A bad smell is an indication of some problem in the code, which requires refactoring to deal with. Many tools are available here for detection and refactoring of these code smells. These tools vary greatly in detection methodologies and acquire different competencies. In this work, we studied different code smell detection tools minutely and try to comprehend our analysis by forming a comparative of their features and working scenario.[17]

III. Research Methodology

We will firstly examine all the smells of code from previous work as well as papers. It is seen that there are at least six smells which are defined and observed by coders. All these smells are further classified into metrics which define these briefly so that study if these will become simple and easy for us.

Firstly, we begin with very first smell that is long method. In it we consider number of method along with line of codes as metrics to detect this smell. Here, we set a certain limit for both line of codes as well as value of methods and if that limit is crossed then that particular code is declared as smelled code. Moreover, in upcoming paragraphs these all are defined with full explanations with long definitions.

Here present the main simple point to tell the methodology used in this thesis is as following:

- Source code would be chosen in .net.
- Visual Studio is the tool used for analyzing the code.
- Bad smells would be detected from our created tool.
- Software metrics plug-in would be applied on source code to calculate the metrics values for analysis and measure the quality of source code.
- Select a project and upload it into our tool. Here we choose ECC application that is ECDS ,in which two different algorithms are scanned to detect the smells.
- Click on Testing of smells button.
- Select a parameter to scan.
- Load all classes of selected project.
- Perform scanning and get results.

IV. Experimentation

The experimentation done is as following:

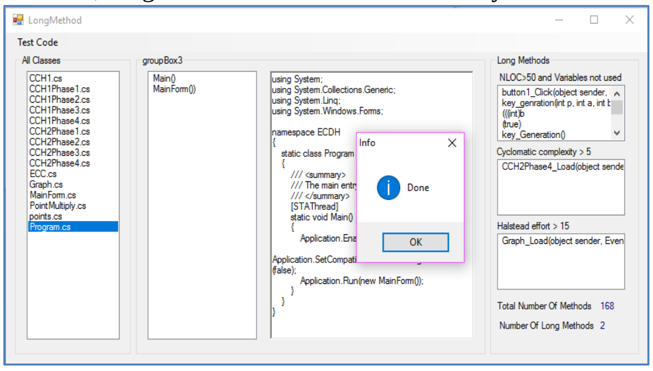
Tool used to create window base GUI application: visual studio ultimate. The application for this system is formed with the tool called visual studio. It is developed with the help of the c# .net. This application is used to detect the different type of code smells along with the various distant metrics rules. The source codes for the database in this work are in .net as well as java language. Mainly, the whole results are performed on the algorithms of ECC application which is in .net language. In this, different types of metrics are measured according to code smell. Each bad smells has different metrics rules.

A. *Long Method*

The following metrics for the detection of long method smell

- Rule 1: If Number of line of code (NLOC) exceeds the number 50 moreover, there are declared variables present but not used in the running code.
- Rule 2: If Cyclomatic complexity is more than the number of 5(if else).
- Rule 3: Halstead effort E=D*V should be smaller than the 15. It consists of the operators and operands used in the source code.

Out of these three rules if one of the true or all are true then, long method bad smell is detected by our work.



CCH1.cs
CCH1Phase1.cs
CCH1Phase2.cs
CCH1Phase3.cs
CCH1Phase4.cs
CCH2Phase1.cs
CCH2Phase2.cs
CCH2Phase3.cs
CCH2Phase4.cs
ECC.cs
Graph.cs
MainForm.cs
PointMultiply.cs
points.cs
Program.cs

Main()
MainForm())

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace ECDH
{
    static class Program
    {
        /// <summary>
        /// The main entr
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Ena

Application.SetCompati
(false);
            Application.Run(new MainForm());
        }
    }
}
```

Done

NLOC>50 and Variables not used
button1_Click(object sender,
key_genration(int p, int a, int b
(((int)b
(true)
key_Generation()

Cyclomatic complexity > 5
CCH2Phase4_Load(object sende

Halstead effort > 15
Graph_Load(object sender, Even

Total Number Of Methods 168
Number Of Long Methods 2

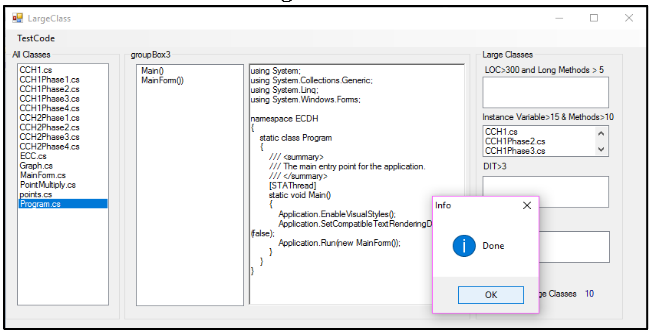
Fig. 1: Long Method

B. *Large Class*

The following metrics for the detection of large class smells

- Rule1: the number of lines of source code must be less than 300 and furthermore, the long methods not greater than the number 5.
- Rule 2: If Number of instance variables and methods are more than 15 and 10 respectively.
- Rule 3: the depth of inheritance tree (DIT) should be lower than the 3
- Rule 4: coupling, if number of operation calls and number of called classed are high.

So, at the end if from the above rules is or are true then, the occurrence of large class code smell is sure.



CCH1.cs
CCH1Phase1.cs
CCH1Phase2.cs
CCH1Phase3.cs
CCH1Phase4.cs
CCH2Phase1.cs
CCH2Phase2.cs
CCH2Phase3.cs
CCH2Phase4.cs
ECC.cs
Graph.cs
MainForm.cs
PointMultiply.cs
points.cs
Program.cs

Main()
MainForm())

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace ECDH
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingD
(false);
            Application.Run(new MainForm());
        }
    }
}
```

Done

LOC>300 and Long Methods > 5

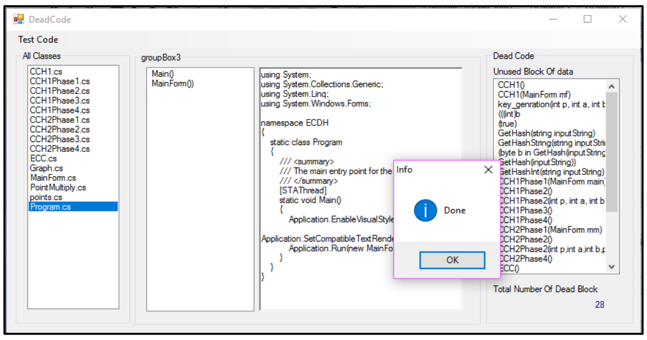
Instance Variable>15 & Methods>10
CCH1.cs
CCH1Phase2.cs
CCH1Phase3.cs

DIT>3

ge Classes 10

Fig. 2: Large Class

C. Dead Code



CCH1.cs
CCH1Phase1.cs
CCH1Phase2.cs
CCH1Phase3.cs
CCH1Phase4.cs
CCH2Phase1.cs
CCH2Phase2.cs
CCH2Phase3.cs
CCH2Phase4.cs
ECC.cs
Graph.cs
MainForm.cs
PointMultiply.cs
points.cs
Program.cs

Main()
MainForm())

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace ECDH
{
  static class Program
  {
    /// <summary>
    /// The main entry point for the
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.EnableVisualStyle

Application.SetCompatibleTextRende
      Application.Run(new MainFo
    }
  }
}
```

Done

Unused Block Of data

```
CCH1()
CCH1(MainForm mf)
key_genration(int p, int a, int b
(((int)b
(true)
GetHash(string inputString)
GetHashString(string inputStri
(byte b in GetHash(inputString
GetHash(inputString))
GetHashInt(string inputString)
CCH1Phase1(MainForm main)
CCH1Phase2()
CCH1Phase2(int p, int a, int b
CCH1Phase3()
CCH1Phase4()
CCH2Phase1(MainForm mm)
CCH2Phase2()
CCH2Phase2(int p,int a,int b,p
CCH2Phase4()
ECC()
```

Total Number Of Dead Block 28

Fig. 3: Dead Code

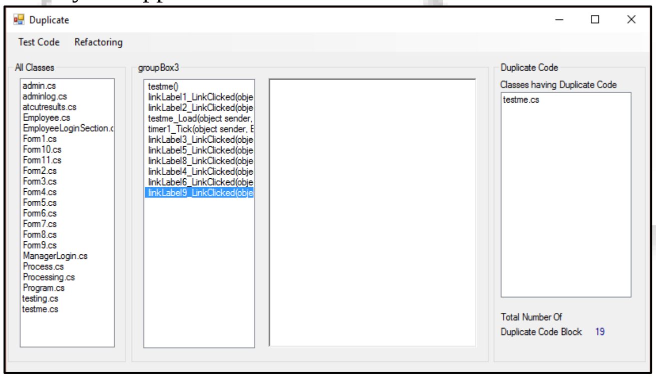
Here, we see all the metric used in case of dead code smell detection in our system.

– Rule1: In it all the data is analysis by our application and if there is present any unused block or say class of data then, as result of it the dead code smell is detected in our source code by the application.

D. Duplicate Code

The following is the metric used for the duplicate code detection.

– Rule1: Here, if there is any variable or method which is write again and again without using calling function option then present of duplicate code smell is detected by the application.



admin.cs
adminlog.cs
atcutresults.cs
Employee.cs
EmployeeLoginSection.c
Form1.cs
Form10.cs
Form11.cs
Form2.cs
Form3.cs
Form4.cs
Form5.cs
Form6.cs
Form7.cs
Form8.cs
Form9.cs
ManagerLogin.cs
Process.cs
Processing.cs
Program.cs
testing.cs
testme.cs

```
testme()
linkLabel1_LinkClicked(obje
linkLabel2_LinkClicked(obje
testme_Load(object sender,
timer1_Tick(object sender, E
linkLabel3_LinkClicked(obje
linkLabel5_LinkClicked(obje
linkLabel8_LinkClicked(obje
linkLabel4_LinkClicked(obje
linkLabel6_LinkClicked(obje
linkLabel9_LinkClicked(obje
```

Classes having Duplicate Code
testme.cs

Total Number Of
Duplicate Code Block 19

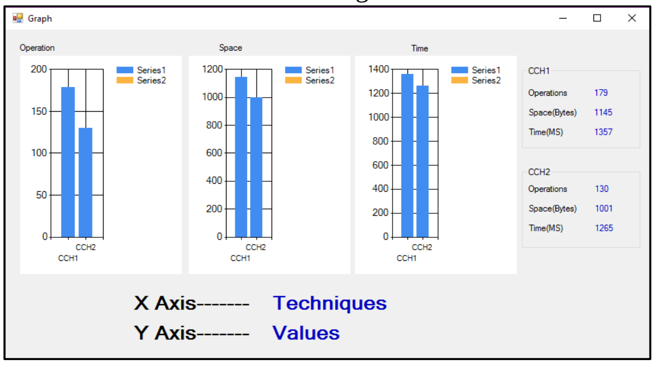
Fig. 4: Duplicate Code

V. RESULTS

A. Result Before Refactoring

This graph shows the value of all the parameter which are considered to measure the performance of both the CCH1 and CCH$_2$ algorithm of ECC application. The parameters are time, space and operations. Time is defines as a parameter in which we are count the total time used by the application to run all the source code of both the algorithms differently.

After that the space is parameter in which we consider the total memory utilized by the both algorithms while running. Last but not least, operation considers the total numbers of operations are running within the source code of both CCH1 and CCH2 algorithms.



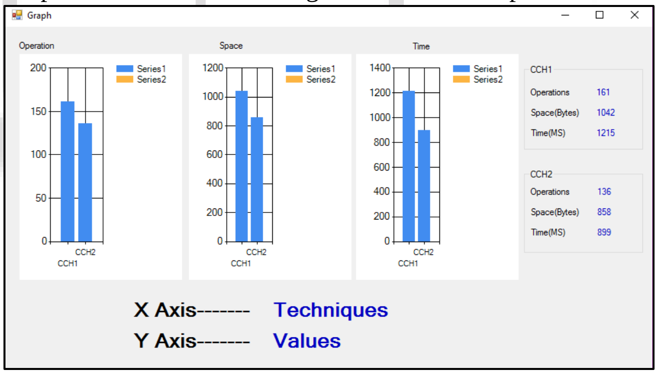
CCH1
Operations 179
Space(Bytes) 1145
Time(MS) 1357

CCH2
Operations 130
Space(Bytes) 1001
Time(MS) 1265

Fig. 5: Show the results of CCH1 and CCH2 algorithms before refactoring

B. Result After Refactoring

This graph depicts the number of code smell again but after the fixing all the detections. Therefore, the number of all code smells now decreased if can easily solved with the help of metrics which are derived in all the code smells.As the code smells are fixed as consequences of it the graph of all the parameters of both the algorithms is also improves.



CCH1
Operations 161
Space(Bytes) 1042
Time(MS) 1215

CCH2
Operations 136
Space(Bytes) 858
Time(MS) 899

Fig. 6: Show the results of CCH1 and CCH2 algorithms after refactoring

Comparison Criteria	Developed Software	Clock Sharp	Find Bugs	Programming Mistake Detector
Tool Description	Stand alone	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 research work / language	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 1: Comparison of present tool with old developed tools

VI. Conclusion

The five bad smells are detected in the application of ECC, under with two different algorithms called CCH1 and CCH2's source code using GUI application developed. The measured object oriented metrics shows the value of each metric in their respective code smells detected on the coding. Calculated metric values will help in applying the refactoring methods directly on the source code to eliminate the bad smells and to improve the structure of existing code. Like Coupling factor will be helpful to decide whether we can apply "MOVE" method of refactoring or not .The value of NLOC, Cyclomatic complexity and Halstead effort will be helpful in applying Extract Method and Replace Temp with Query methods of refactoring.

VII. Future work

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 and further we can implement it using GA (Genetic Algorithm). The results accessible here are the first of many smell studies and we receive further searching in this area.

References

[1] Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading, MA, 2000.

[2] Karnam Sreenu 1, D. B. Jagannadha Rao2 "Performance - Detection of Bad Smells In Code for Refactoring Methods" International Journal of Modern Engineering Research (IJMER). Vol.2, Issue.5, Sep-Oct. 2012

[3] Safwat M. Ibrahim1, Sameh A. Salem1, Manal A. Ismail1, and Mohamed Eladawy1 " Identification of Nominated Classes for Software Refactoring Using Object-Oriented Cohesion Metrics" IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 2, No 2, March 2012 ISSN (Online):

[4] Marjia Katic "Software Redesign Methods" Departments of Applied Computing Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia.

[5] Francesca Arcelli Fontanaa Pietro Braionea Marco Zanonia, "Automatic detection of bad smells in code: An experimental assessment ," Journal of Object Technology Published by AITO Association Internationale pour les Technologies Objets, c JOT 2011

[6] Stefan Slinger , "Code Smell Detection in Eclipse," Delft University of Technology

[7] Kwankamol Nongpong, "Integrating \Code Smells" Detection with Refactoring Tool Support " The University of Wisconsin-Milwaukee August 2012

[8] Panita Meananeatra and Songsakdi Rongviriyapanish," Using Software Metrics to Select Refactoring for Long Method Bad Smell" Computer and Information Technology Software Engineering Paper ID 118

[9] Bart Du Bois "Refactoring - Improving Coupling and Cohesion of Existing Code", august 2015.

[10] M. Fowler, Refactoring: Improving the Design of Existing code. Addison-Wesley, 1999.

[11] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in 20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA. IEEE Computer Society, 2004, pp. 350-359.

[12] Almas Hamid, Muhammad Ilyas, Muhammad Hummayun and Asad Nawaz " A Comparative Study on Code Smell Detection Tools" International Journal of Advanced Science and Technology Vol.60, (2013)

[13] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to ObjectOriented Programming, Systems, Languages, and Applications. ACM Press

[14] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in Proceedings of 5th European Conference on Software Maintenance and Reengineering. Lisbon, Portugal: IEEE CS Press, 2001, pp. 30

[15] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02). IEEE CS Press, Oct. 2002.

[16] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in Proceedings of the 11th International Software Metrics Symposium. IEEE Computer Society

[17] Almas Hamid, Muhammad Ilyas, Muhammad Hummayun and Asad Nawaz " A Comparative Study on Code Smell Detection Tools" International Journal of Advanced Science and Technology Vol.60, (2013)

[18] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20–36, 2010