**An Empirical Study on the Impact of Refactoring Techniques on Mitigating Architectural Smells in Software Systems**

**Final Paper**

**Software Architect and Design**

**Harish Toka**

**Sita Ramanjaneyulu Yarlagadda**

**Abstract:**

By recognising code smells and reducing risk factor through refactoring

techniques, the automated code scent detection and refactoring tool we offer in this research can

be utilised to assess risk factor. Refactoring is a technique used to improve or reorganise a piece

of software's underlying structure without affecting its functioning. A new code scent detection

tool named "Lazy Catch" is also provided. Many facts and guidelines form the foundation of

code smell detection. We used this method to locate the offensive bad smells in case studies based on oops, such as those in C#, CPP, and Java. This suggests that the instrument is not language-reliant. The risk factor level is categorised into three groups (Hi, Low, Medium).Refactoring, risk assessment, bad smell detection in code, and object-oriented metrics are some of the keywords.

Research Problem:

The areas where the software engineering pays little attention from the previous researchers comes under the usefulness of refactoring all the strategies that can reduce the architectural smells that can be occurred in the software system. Even though there are some refactoring strategies that can improve the code quality and maintainability, there should be still some improvement in the process to be done. This type of approaches on the reducing the architecture smells has not been studied in depth[3].

**Introduction:**

Everyday life now involves the usage of software. The rule of software is its ability to increase

productivity, efficiency, and ease of life; nevertheless, these efficiencies come at the expense of

exhaustive observation, leading to a race of people that "never forgets." The ability to adapt is a

requirement for software systems. Using design patterns could lead to unnecessary give and

make software development more challenging.

Many tasks are part of the refactoring process, including:

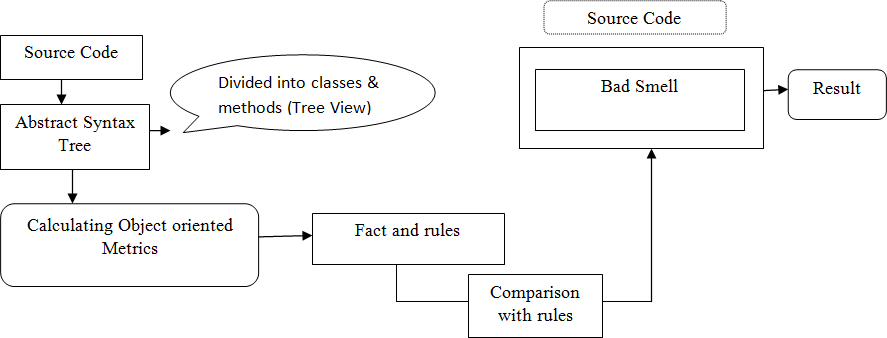
• Identify the dining areas in which the programme should be installed.

• Choose which refactoring(s) should be used in the spots that were discovered.

• Maintain the dependability of the software items from Refectories and other objects (such as

documents, propose documents, necessities specifications, tests and so on).

• Implementation of the refactoring and consent that behaviour is preserved.



**Background:**

Together with information on risk analysis, code smells, and software metrics, this part also

provides the procedural framework for the software maintenance process.

**2.1 Computer software upkeep:-**

Software engineering is the process of making changes to a software product after it has been

released to address defects, boost performance, or alter other elements. They must first

comprehend the operation of a strategy before they can make changes to it.[7].

**2.2 Scent of Code:-**

It can be difficult to evolve and maintain software due to the distinctiveness, or "code smells," of

the software. These code smells can suggest a programming or design difficulty. Hence, when

they need to be disconnected, it is advisable to take them out as quickly as possible. Tools for

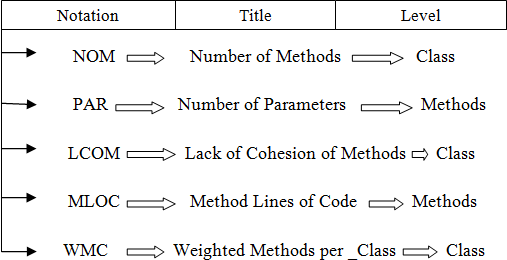
detecting code smells are particularly useful because many of them may be missed by

programmers while they are at work[1].

**2.3 Metrics for Software:-**

Metrics are a numerical component of software. The metrics connected to source code are the

only ones we concentrate on in this study, as indicated in the following Figure.

**2.4 Measurement Thresholds for Software:**

The description of code smell detection rules frequently uses metric groupings or categories.

Identifying classifications with lower dependability or classifying operations with a HIGH

degree of difficulty are two examples. We want to acquire thresholds in such a way that they can

be semantically translated to these fundamental needs, so that we can identify what LOW unity

or HIGH difficulty implies in terms of the metrics we use to quantify the unity and difficulty of

the software[6].

**2.5 Analysis of thresholds effects:**

The acceptable risk level formula, where and are the coefficient estimates and the probability po

is suggested with different five risk levels, namely (po = 0.5 to po = 0.7), gives the threshold

values of the stated metrics. The threshold values for Equation 1 depending on the risk degree of

the odour.

**2.6 Risk Evaluation:**

Before suggesting the advance for a software system, the project manager should assess the risks

presented by the development endeavour. There are many different risk assessment methods, and

depending on the system attribute as described by the system, some of them call for human

involvement. Risk assessments are essential to the process of creating an administration.

**Related Work:**

Unpublished studies on the effects of code smells are examined in this section. Articles from

Transaction, IEEE, and sci-index software engineering periodicals that were published between

2012 and 2014 were included in a structured evaluation of the literature on code smells and

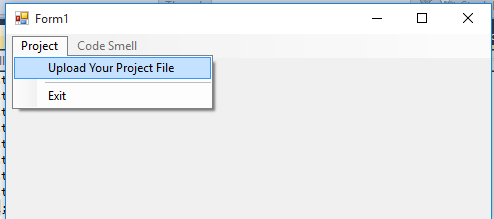
refactoring. In that study, it was stated that Danphitsanuphan et al. 2014's research constituted a

breakthrough in the detection of "poor smells" in software, also referred to as "Code Smell." [5].

**Software Architecture Risk Based Detection Tool:**

The detection technique entails word-by-word analysis of the code; for instance, if the code

contains a method statement, the programme will search for lengthy methods and lengthy lists of

parameter values. In order to find any empty chains or message chains, the software then iterates

through each line of the specific code.

In the upper grey area of the image above, there are two options. Whereas the second is used to

show code smells, the first is used for projects. Click the Upload Project File option under the

Project menu. The three project types (C++, Java, and C#.NET) should be published.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Smell** | **Definition** | **Variable used** | **Results** |
| Long Process | An extended and composite method is divided into dummy and well-named methods by employing refactoring approaches like the extract method. An explanation of why specific strategies could be applied to the procedure. | • Cycle complexity  • LOC  • A Range Of Methods | Classes and methods' source code is being uploaded. says the abstract syntax tree. Calculations for object-oriented matrices included the amount of lines of code in the process, total number of variables, variables that were used but not used, cyclomatic complexity, and Halstead attempts. |
| Extensive list of parameters | Long parameter list techniques are difficult to understand and update in object-oriented programming, thus it is possible to recover a long parameter list method by passing an object in place of the parameters. | Complete Parameters  • A method's input  • Common Parameter | The fundamental criterion for this approach is that it has more constraints than 7 (NOP>7), that the average parameter in C is 3, and that there are more parameters than the average parameter. |
| Sizeable Classes | To make large classes easier to understand and maintain, they are split up into smaller ones, each for a certain dependability. | • Lines with Codes  Infrequent Variable  • Depth of Inheritance  • Connecting | The code has a class LOC > 300, a long method > 5, a utilised instance of a variable id > 15, and methods > 10. The greater distance between a node and the tree's root is referred to as inheritance depth, and this is determined by DIP > 3 and coupling > 10. |
| Code Dead | The elimination of non-organic code is referred to as dead code. This is why we have source control systems. | • Block of unused data | Unused There are 24 bytes of data in the block as a whole. |
| Block Lazy Catch | By comparing the result to the cutoff, find the empty catch block. | • The number of catch blocks left unused. | There are five unutilized catch blocks altogether. |
| Change Statement | Switch statements may have a single switch section or more. Each switch section begins with one or more case labels and ends with one or more words. | • The number of incidences  Case by case  • Complicated | There is no default case in this statement, there are more than or equal to 10 cases used, and the complexity is higher than or equal to 10. |
| Portable Field | If you're only briefly using an object as a technique parameter, make sure to use the entire object, not just the most desirable single fields. | • Temporary Field | There are 30 temporary fields overall, but no variables are used in them. |
| Comment section | Comment lines, which are meant to make resource code simpler to read, are often ignored by compilers and line-by-line code inspections. | Authentic Source Code | There are two classes and more comment lines used than there are actual source lines (33%). |

**Comparision Tool:**

Here, we take a look at a few code-smell detection programmes, each of which has a special set

of abilities[5].

**5.1 Clock Sharp:**

Clock Sharp is a command-line tool that can be used to verify code against more than 100

programming principles. It is a code organiser for the C# programming language that is

integrated with Visual Studio 2008 and 2010.

**5.2 Find Bugs:**

Using continuing research, the open source project Detect Flaws identifies the four most likely

types of frightening and unpleasant errors and searches the Java byte code for defects.

**5.3 PMD (Programming Mistake Detector):**

Source code analyzers that fall into one of five categories can find bugs like 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, sub-optimal code, vacant local

variables, dead code, avoidable statements, for, and while statements, among others.

**Simulation Model:**

The object-oriented, banking system-specific case study programme is written in C#. We will

identify faults in all classes using the Code Smell detector and code samples like Admin.cs and

Adminlog.cs. Visual Studio is the programme used to analyse the code.

There are many refactoring metrics, such as total lines of code, packages, methods, classes, and

attributes.

**Software Specification:**

The project's source code in any language (C#, C++, Java) is required to assess a project's quality

using software metrics. The tool for executing source code is required. Take Visual Studio and

its plug-in as an illustration.

**Hardware Specification:**

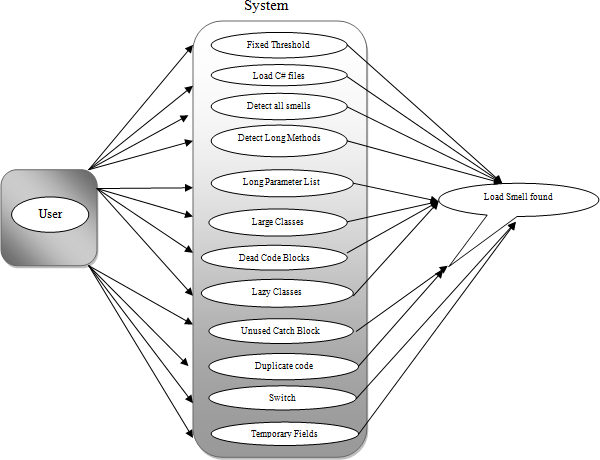
In order to establish the size of the portable coordinate measurement device needed, examine the

surface geometry of your physical model or component. Although there can be limitations on

digitizer size, they can be readily bypassed by using the leap frog feature, which is part of the

refactoring eclipse plug-in. Using a computer with a top-notch graphics card and enough

memory is always recommended[4].



**Significant Research Work:**

**• Maintainability:** It is easier to attach defects because the source code is easy to comprehend

and the author's intent is obvious. Large monolithic operations are broken up into a number of

discretely short, appropriately labelled, single-purpose methods in order to achieve this capacity.

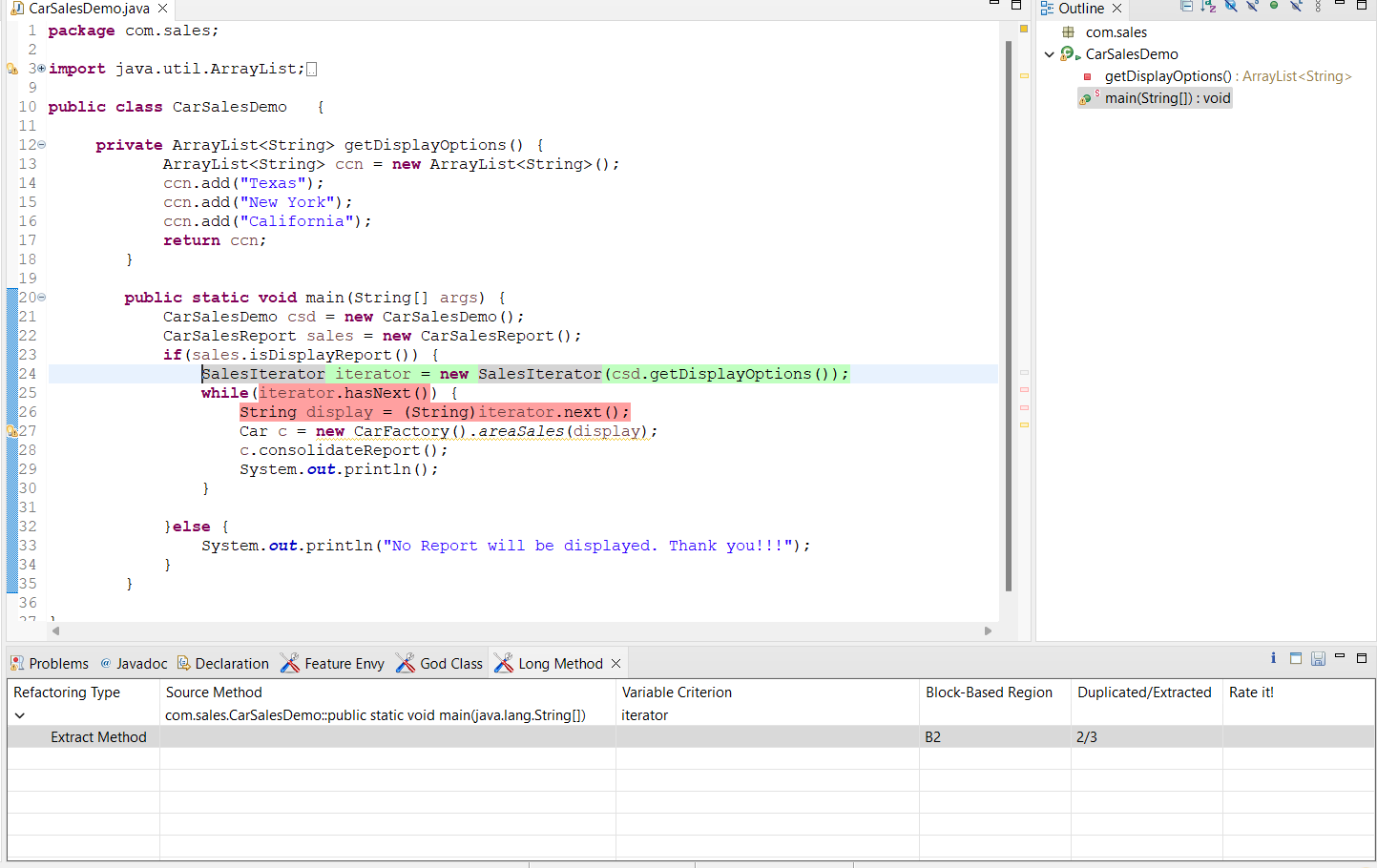
**• Extensibility:** It is straightforward to expand the capacity of an application if it uses

recognisable design patterns and offers some give where none may have previously been.

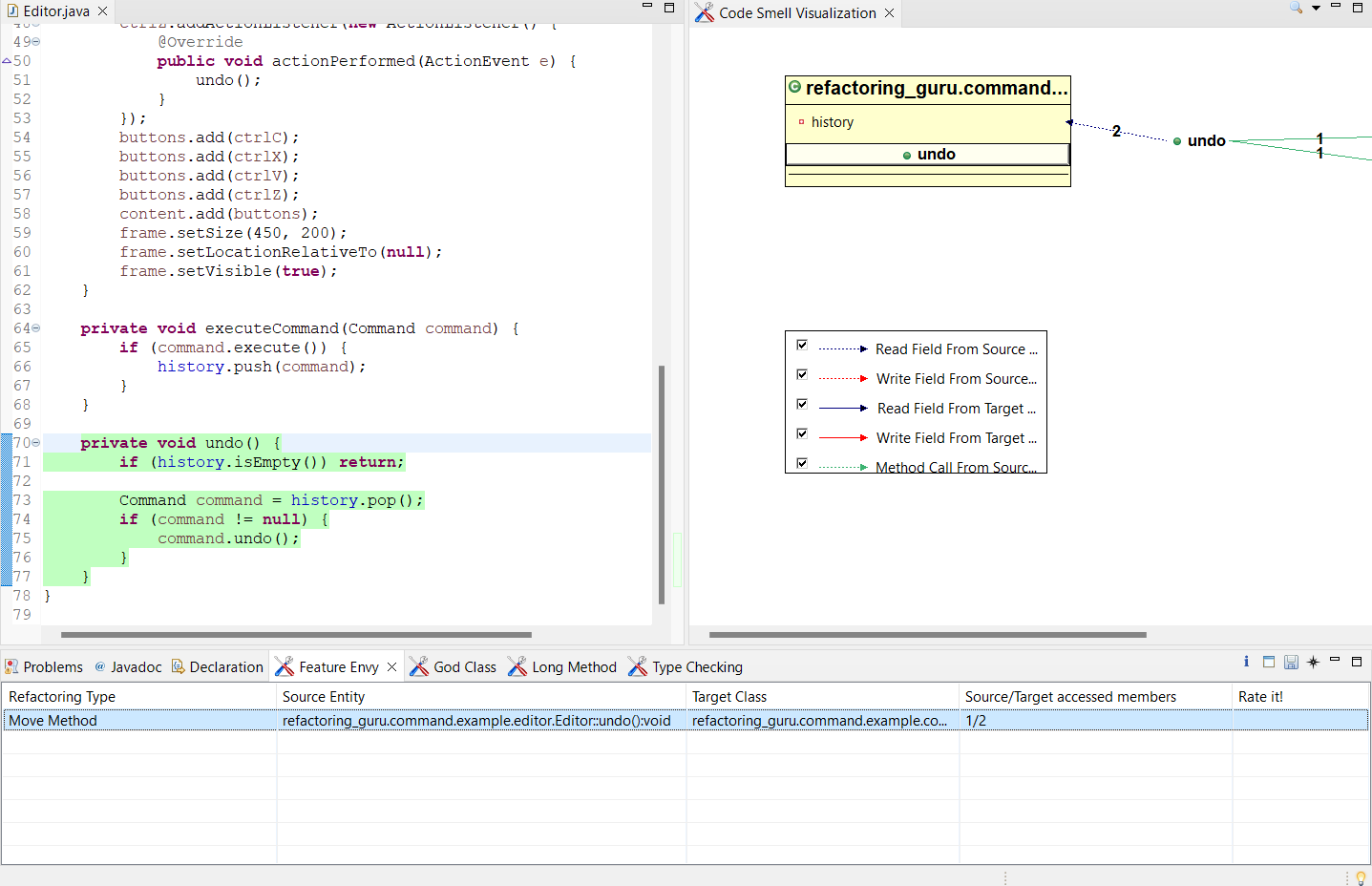
**Results from tool:**

Took the car sales as the input CSV and measured the smells.

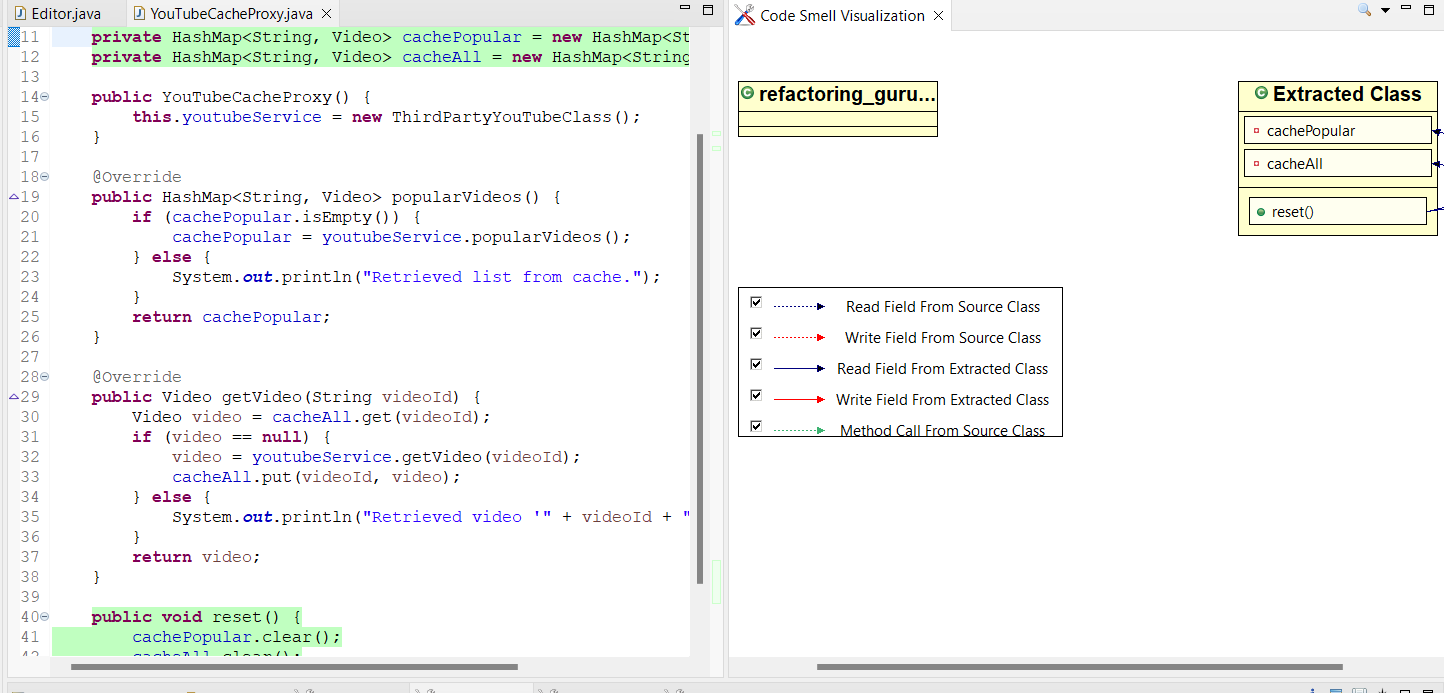
Car Sales is obtained by long method



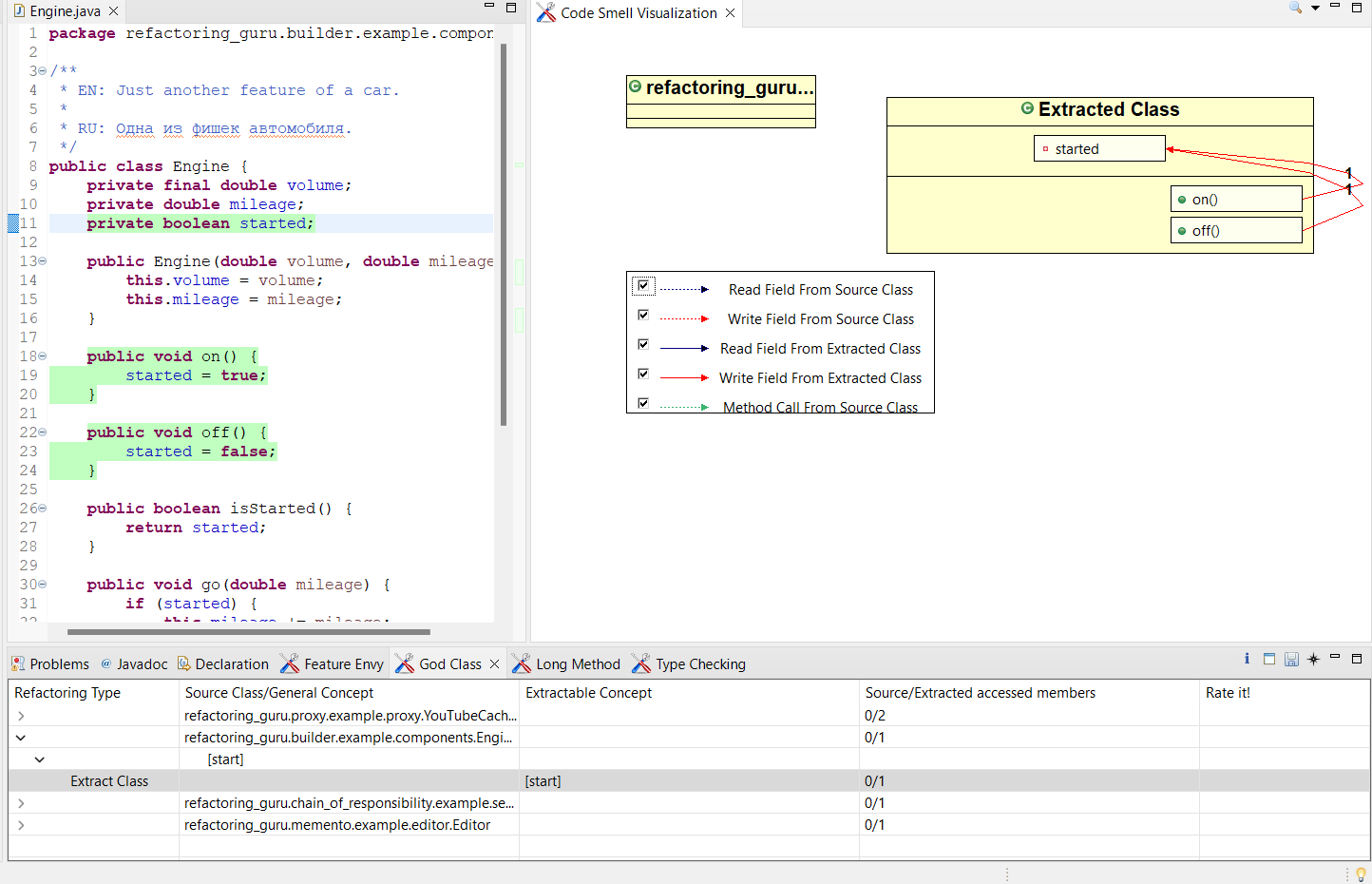
Abstract Factory (Feature Envy).



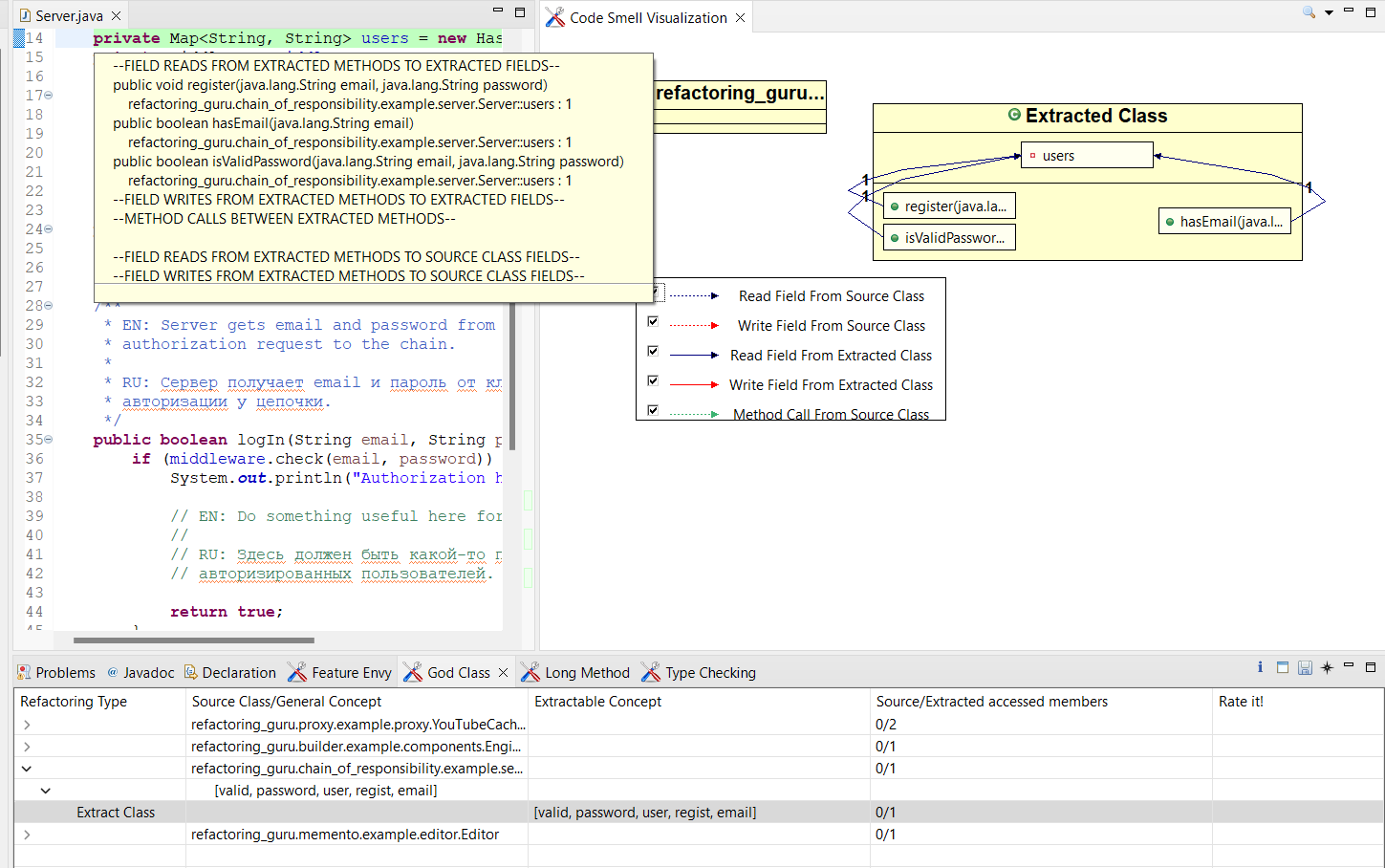
Proxy – God Class



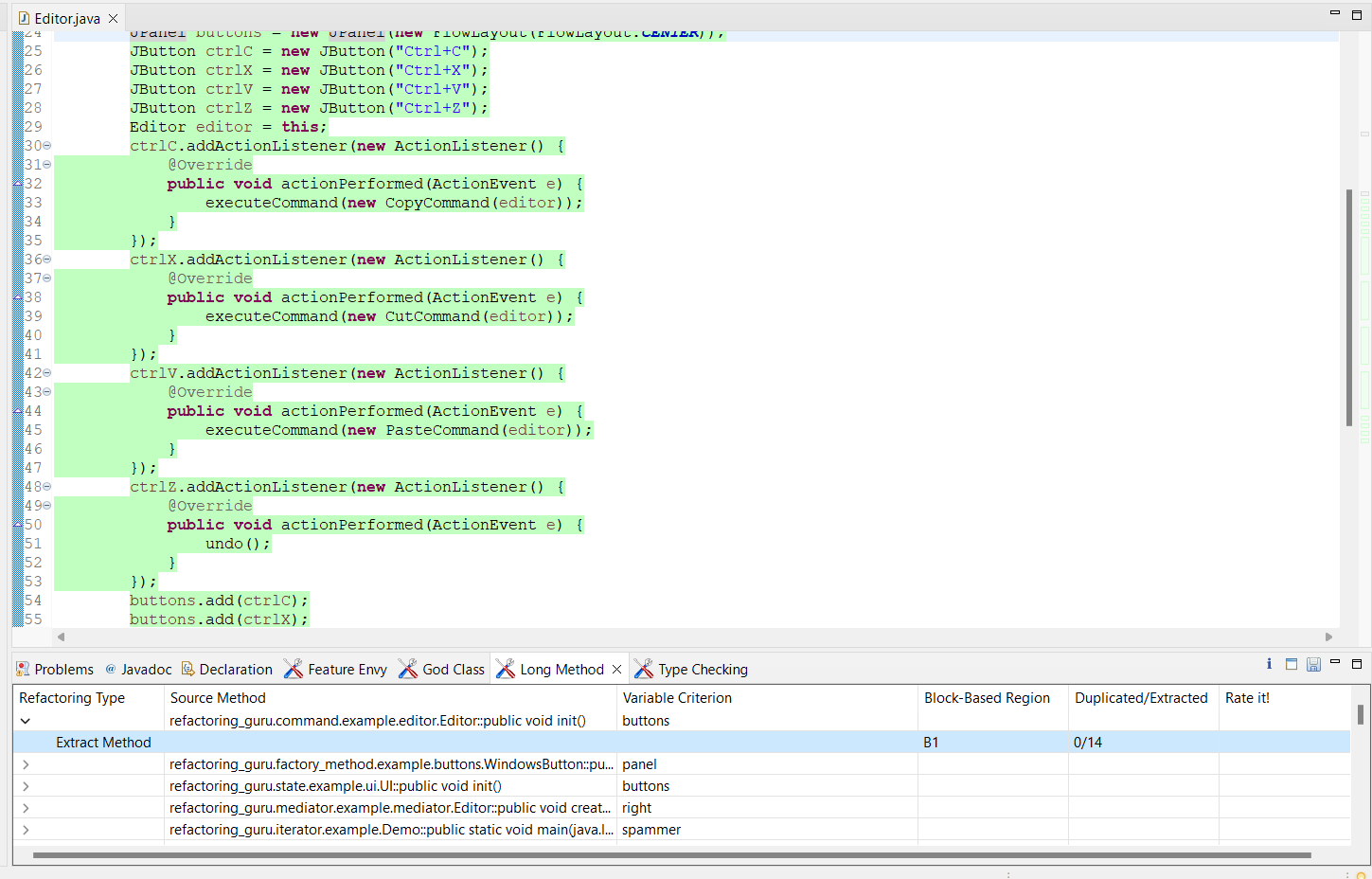
Components – God Class



Server – God Class

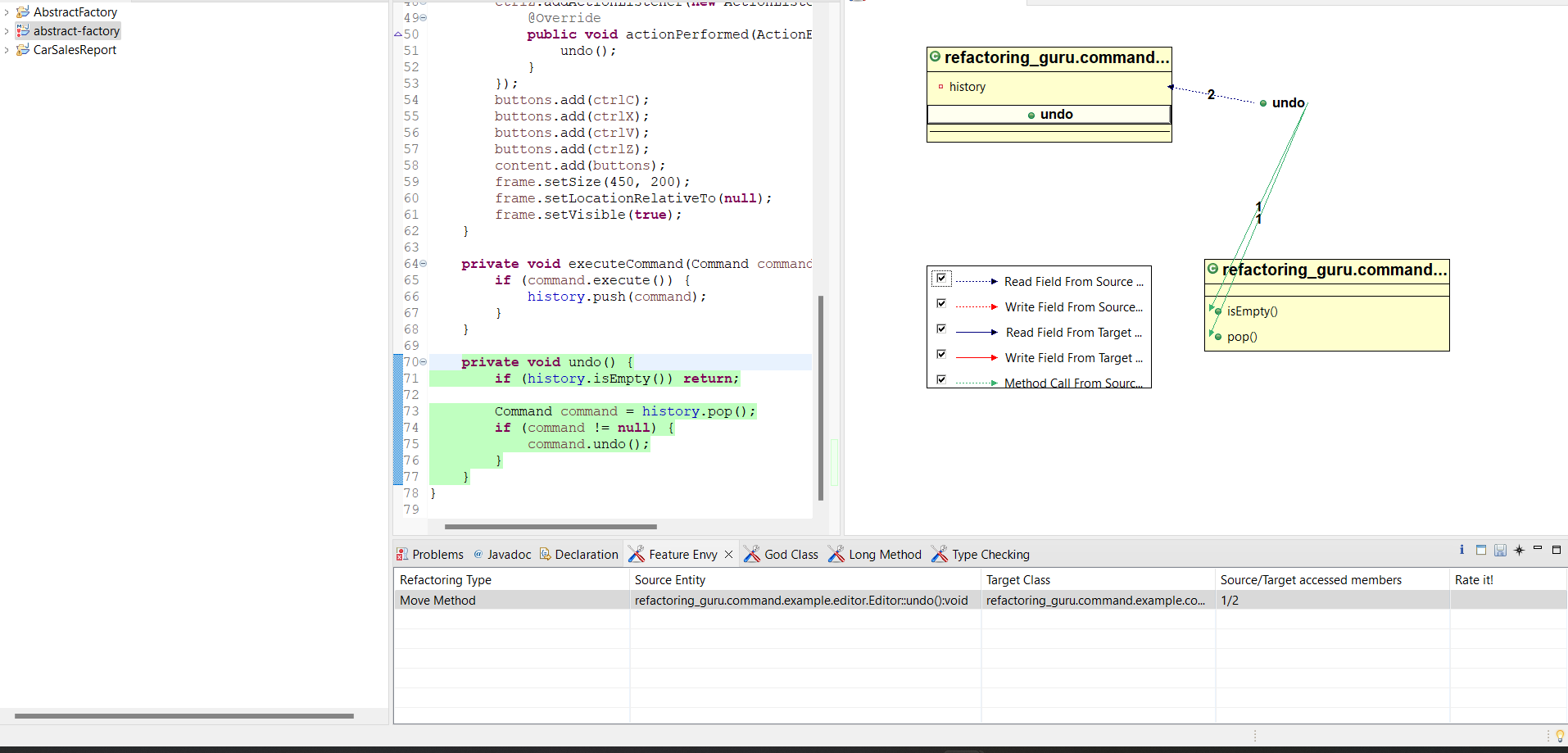


Editor – Long Method

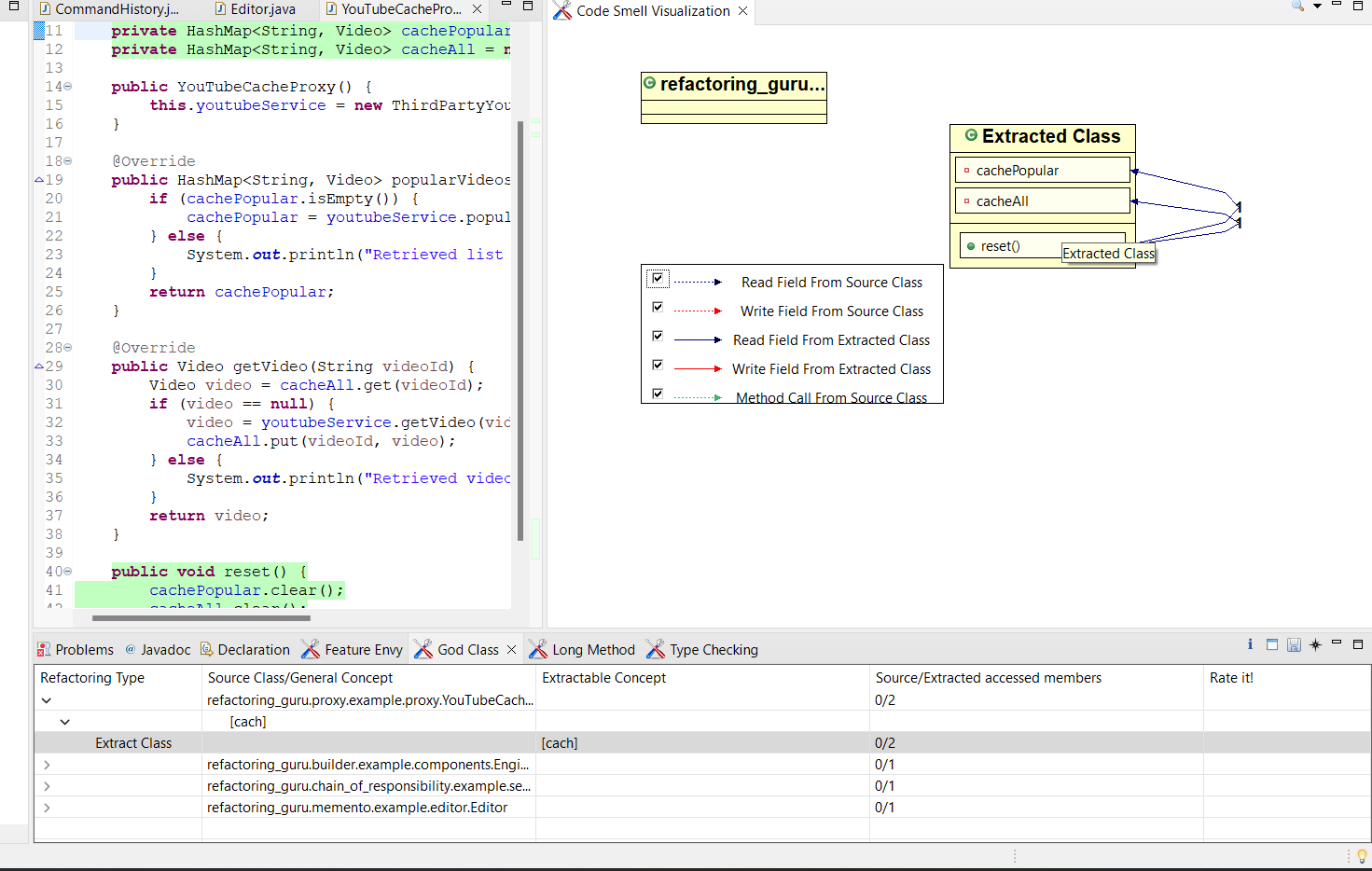




Abstract Factory – Feature Envy



God Class



**Case Study:**

An smell identification case study for the banking system has been developed in object-oriented

languages (.net, c++, and java). The many obnoxious odours in the source code of the banking

system are located with the use of a built-in graphical user interface programme.

1. How do I verify a long method?

A lengthy method is one of the more common and readily repaired forms of code smells among

the many others. The implementation of lengthy methods in your project is incredibly

straightforward when using Visual Studio's analysis tools[3].

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] == ';') loc++;

end end

if (vari. Contains(','))

string[] variables = vari. Split(',');

//to check the end of the lines through semicolon (LOC)

// to find the colons

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)

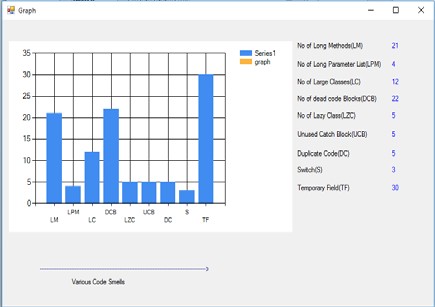
2. Dead code Block checking and why should dead code be removed?

It might contain unused, redundant, or inaccessible code. Using the Code Analysis function in

Microsoft Studio, we could find it.

1. To enhance code coverage results.

2. The code's ability to be updated.

3. Increase your output.

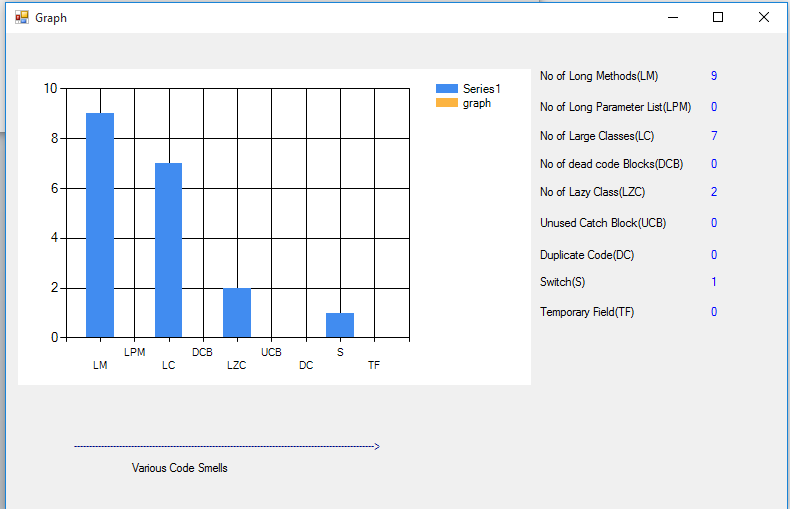
This picture illustrates that a long procedure is any method that is convoluted enough to be

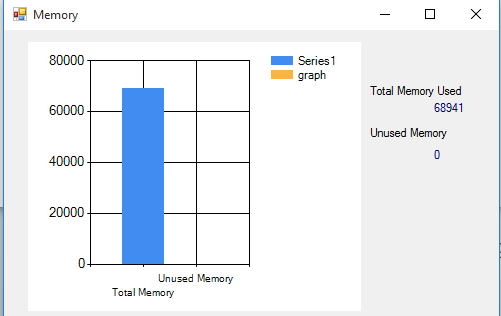
challenging to understand in a single glance.There are four long parameter lists (LPL), and there

are twelve enormous classes.

The graph up top shows how total memory and unused memory were separated into two groups.

The entire amount of RAM consumed is 80619, and 11678 of it is being left unused.



This diagram illustrates the necessity of restructuring in order to eliminate odours in the code.

Refactoring is the process of improving the code's superiority without changing the way it

appears to operate[2].

**Conclusion and Future Scope:**

Last but not least, we have to recognise that comparing the tools is quite difficult, and in some

circumstances, using them is as difficult and urgent. In the source code of the banking system,

which makes use of a graphical user interface, numerous code smells have been discovered. The

created object-oriented metrics show the significance of each statistic in connection to certain

code faults. Future research will only focus on an experiment using developers to duplicate

Mantyla's developer research and on an examination of the testing ramifications of smell

suppression.

**References:**

[1] D. Sas, P. Avgeriou, and U. Uyumaz, “On the evolution and impact of architectural smells—an industrial case study,” *Empir. Softw. Eng.*, vol. 27, no. 4, p. 86, 2022.

[2] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” *J. Syst. Softw.*, vol. 167, p. 110610, 2020.

[3].Garousi, Vahid; Küçük, Barış (2018). "Smells in software test code: A survey of knowledge in industry and academia". Journal of Systems and Software. 138: 52–81. doi:10.1016/j.jss.2017.12.013

[4].Miller, Jeremy. "Downcasting is a code smell". Archived from the original on 16 February 2019. Retrieved 4 December 2014.

[5].Sharma, Tushar; Spinellis, Diomidis (2018). "A survey on software smells". Journal of Systems and Software. 138: 158–173. doi:10.1016/j.jss.2017.12.034

[6].Suryanarayana, Girish (November 2014). Refactoring for Software Design Smells. Morgan Kaufmann. p. 258. ISBN 978-0128013977.

[7] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smell," *Empir. Softw. Eng.*, vol. 25, pp. 4020–4068, 2020.