Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X
Volume 4, Issue 3, 2026

UNSTRUCTURED DATA ANALYSIS TECHNIQUE FOR MEASURING COHESION IN SOURCE CODE USING MACHINE LEARNING

Furqan Ashraf[1], Muhammad Nasir[*2], Zakia Jalil[3]

[1, *2,3]*Faculty of Computing & Information Technology, International Islamic University, Islamabad, Pakistan*

[1]furqan.msse483@iiu.edu.pk, [*2]m.nasir@iiu.edu.pk, [3]zakia.jalil@iiu.edu.pk

DOI: https://doi.org/10.5281/zenodo.18918301

Keywords

Article History
Received: 08 January 2026
Accepted: 21 February 2026
Published: 09 March 2026

Corresponding Author: *
Muhammad Nasir

Abstract

One of the major challenges for software community is to find source code quality matrices while using Object Oriented Paradigm. With advancements in Machine Learning and Natural Language Processing (NLP) it is now possible to evaluate code using unstructured data analysis. Many tries have been made to capture this important software quality attribute using traditional structured analysis methods. This research study aims to investigate unstructured data analysis could be performed for calculation of cohesion in source code classes. In this study, we designed an experiment to evaluate cohesion score results of two datasets of source code corpus. Furthermore, unstructured way of measuring high cohesion in source code classes is presented using semantic analysis of class names with method names and class description with methods descriptions. The results gathered through performing experiments yielded that unstructured data analysis technique can be applied for finding cohesion of classes. In this study we calculate LCOM for each class present in obtained datasets. By comparing the experiment result with LCOM we get following results. The study compares the results of cohesion score obtained from this technique with traditional LCOM score of source code classes. It is common practice in software industry to follow naming conventions of classes and writing proper description of classes and methods, unstructured data analysis can be effectively applied for calculating the cohesion score of classes.

Introduction

Basic units of object-oriented software are classes. Quality of software design is quality of classes design. Automated code analysis is widely used for object-oriented software testing. Improving code quality always remained the focus in software development lifecycle [1]. Class cohesion is described as how much a class follows a tight relationship within its elements' implementation. High cohesion is a critical quality assurance measure in object-oriented software [2]. Low cohesive class consists of unrelated methods with respect to purpose and relatedness. Finding class cohesion and improving it is the main focus of modern software design [3]. Trivial static code analysis techniques generate false alarms. With improvements in machine learning, it is now possible to generate different levels of warnings for the same software code using different trained ML models [4]. Class cohesion is one of the important attributes of class quality [5]. Despite being an important software quality measure, very few studies have been conducted focusing on high cohesion using an unstructured data analysis approach. LCOM stands for Lack of Cohesion of Methods in software engineering. LCOM is a metric that assesses how closely linked methods are to one another within a class or module. It

helps determine the degree of design complexity and quality in software systems [6].

LCOM determines lack in ability of a class in accessing instance variables within its methods implementation [7]. In other words, it evaluates the relationship between methods within a class and determines whether or not they exchange data. The coherence decreases as LCOM increases, suggesting that the class may be harder to comprehend and modify as well as less maintainable [8].

Code quality is always a challenge in software community. It consists of software quality attributes. Improving the quality attributes is focus during software development life cycle [9]. In object-oriented software design Cohesion is always ranked important. Class cohesion is relatedness in purpose and implementation of class elements [10].

High cohesion is one of the most desirable quality attributes in class design. Automatically identifying it through quality inspection tools remains a challenge in modern software development [11]. Reference [12] presented a systematic literature review, SLR, of JavaScript malware detection methodologies using the latest machine learning techniques and concluded that machine learning is actively applied in recent detection research.

There are many class and method structure-based approaches to calculate the cohesion score of a class. However, fewer unstructured data analysis approaches exist for determining the cohesion score. To date, there is no comprehensive study that uses the latest advancements in machine learning and natural language processing to calculate the cohesion level of a class.

In this study, we propose unstructured data analysis approaches to determine class cohesion and evaluate the proposed technique by comparing it with the LCOM score, a metrics based approach.

In industry level software projects, LCOM is considered to represent a lack of knowledge of abstraction in software architecture. Therefore, it generates less accurate cohesion scores in classes where abstraction is required. There is a need to conduct a study that calculates cohesion of classes without neglecting abstraction. In other words, in this study, a technique for calculating cohesion is designed that should perform well across all scenarios of class implementation.

Related Work

Quality assurance of object-oriented software source code depends on how cohesive the classes are. Cohesion is defined as the degree of relevance among all operations a class performs [13]. For best object-oriented programming practices and static analysis of Java code, cohesion stands at the top of common quality metrics [3]. Huawei Technologies Co. Ltd. developed the JPeek tool for static code analysis, with a focus on cohesion of Java source code, to achieve higher quality code.

The software community has proposed various metrics over several years to calculate how connected or related the intra-class elements are. Generally, they are not best applied in industrial programming practices. The study is focused on formulating a process for the utilization of cohesion metrics in code analysis. Results of this study will enable users to analyze their projects in an appropriate manner. High cohesion is the most commonly desired quality attribute in class design. Finding it automatically through quality inspection tools is also a challenge faced in modern software development.

Reference [12] presented a systematic literature review for JavaScript malware detection methodologies using the latest machine learning techniques and concluded that machine learning is actively used for detection techniques in recent research. There exist many class and method structure-based approaches to calculate the cohesion score of a class. There also exist fewer unstructured data analysis approaches to find the cohesion score. Still, there is no comprehensive study that uses the latest advancements of machine learning and natural language processing to calculate the cohesion level of any class.

JPeek currently does not have a multi concept approach for estimating code quality. For example, the C3 class cohesion metric bases its estimation of code quality on comment quality. It

would be possible to increase the number of correctly estimated classes by combining current metrics with class cohesion metrics. RFC implementation may also be beneficial. Regardless of coding skill level, this approach may result in over complication.

This empirical study in [5] employed the metric on classes of two open source Java applications with and without considering transitive relations and statistically analyzed the outcomes. Using both direct and transitive relations in LCOM computation improves the ability of LCOM to indicate class quality, according to the empirical study's findings.

Another study [14] used a method operations based approach for finding class cohesion. They used read and write operations of methods performed on class identifiers.

The study presented two latest metrics for finding cohesion within classes. The metrics are focused on the sizes of connected methods taken as strength of connection. The metrics were analyzed on the basis of an empirical framework.

A normalized distance approach for calculating average cohesion in procedures within classes was presented in study [15]. A direct attribute type matrix was used. Number of methods and number of isolated attribute types in a class are the cross products that make up the matrix.

The objective of this study is to present a high level design (HLD) class cohesion metric that can be used to automatically assess design quality at early development stages using UML diagrams and reasonable assumptions. In study [16], Counsell and Swift proposed a normalized Hamming distance approach using parameter occurrence vectors. The approach constructed a matrix of methods in rows and parameter types in columns. The relationship between methods and parameter types defines class cohesion.

Another metric based approach was presented by Wasiq by defining the connection between methods using certain criteria. The approach calculates cohesion by evaluating similarity in invocation of methods from different classes [17].

Study [18] conducted a literature review of ML based techniques used for software fault prediction, which helps in selecting ML techniques for finding cohesion using unstructured data analysis. The following graph shows the number of techniques used in studies.

Study [19] stated research limitations and future development directions while evaluating quality attributes focused on cohesion. Users may be able to modify parameter weights in the future through further studies. Since some classes, such as abstract classes or interfaces, are inherently less cohesive, the validity of quality assessment results may be compromised. The prototype information is currently text based, but with further research, it may be possible to display information on UML diagrams.

In study [20], Bieman and Kang presented an approach for finding class cohesion by evaluating connections of methods with other classes. They calculated cohesion using the number of direct and indirect connections.

This analysis of literature shows that most existing approaches for finding cohesion in classes are structural. However, there is very limited literature on finding class cohesion using unstructured data analysis.

According to the literature review, there is no generalized approach to find cohesion using unstructured data analysis. In our study, we will use advanced unstructured data analysis techniques such as natural language processing (NLP). Furthermore, traditional static code analysis approaches generate false positives due to their specific variable and method connection based implementation. In our study, by using semantic similarity models, we will obtain generalized code cohesion results that should also perform well on abstract classes.

System Model

The aim of this research is to develop a generalized technique using unstructured data analysis to find cohesion in object-oriented software code. According to the empirical findings presented above, the extended LCOM metric, which considers transitive relations, predicts error prone classes more accurately than the original LCOM metric, which only considers direct relations. This technique focuses on developing a cohesion detection approach to

overcome the limitations of metrics-based techniques such as LCOM.

The proposed technique is developed using semantic analysis between class names, method names, class descriptions, and method descriptions to achieve accurate cohesion results for source code classes while evaluating the quality attribute of cohesion.

- Users may be able to modify the parameter weights in the future through studies that go further.
- Because some classes, like abstract classes or interfaces, are inherently less cohesive, the validity of the result of the quality assessment may be compromised.
- The prototype's information is currently text-based, but with more research, it might eventually be possible to display the information on a UML diagram.

The research process and its diagrammatic representation are as follows:

- Literature Survey
- Research Question
- Dataset Collection
- Implementation
- Evaluation
- Conclusion

Experiment will be used for evaluation of the proposed technique. First, the dataset will be collected, and then the required data will be obtained by implementing parsing techniques. This data will be further tested using semantic analysis techniques based on natural language processing. After processing the data to obtain semantic similarity scores, the same data will be used to calculate the LCOM score. The results produced by the proposed technique will be compared with the LCOM score. Finally, future improvements to the technique will be discussed. The expected outcome is techniques to determine class cohesion using unstructured data analysis, which will help the software quality industry, obtain generalized and adaptive results.

Experimental Setup

This study employed a quasi-experimental design to evaluate the results of cohesion gathered by unstructured data analysis technique. Multiple open source repositories were selected as the experimental group. Comparison with LCOM is made by calculating the LCOM value from the same projects. Following Figure 1 shows the experiment design for our study.

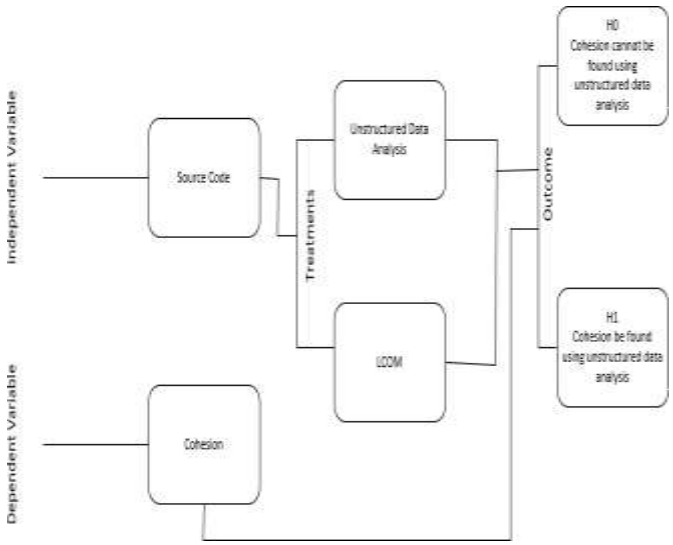

Figure 1 Experimental Design

For the first dataset, top active GitHub projects were selected. Based on the number of watchers and forks of each project from the GitHub Archive, the projects were shortlisted for this

study. We limited our study for only top 20 projects with criteria of having at least 500 commits and 50 collaborators. Furthermore, we exclude the projects where domain of project was duplicating with existing selected project. Selected projects and their descriptions are shown in following table 1:

Through this procedure, a mature, active, and diverse corpus of source code developed by large development teams is obtained as the first dataset. The second dataset is a laboratory manual code written in C#. The laboratory manual code contains 15 design pattern implementations. The following table 2 shows the list of design pattern codes used as the second dataset.

Table 1 Github Projects

Sr.#	Name	Description
1	ElasticSearch	RESTSearch Engine
2	Android-Universal-Image-Loader	Android Library
3	Spring-Framework	Application Framework
4	Libgdx	GameDev Framework
5	Storm	DistributedComputation
6	Zxing	BarcodeImageProcessing
7	Netty	NetworkApp Framework
8	Platform_frameworks_base	AndroidBase Framework
9	BigBlueButton	Web Conferencing
10	Junit	Testing Framework
11	Rxjava	ReactiveJVMextension
12	Retrofit	REST Client
13	Clojure	ProgrammingLanguage
14	Dropwizard	RESTfulwebserver
15	OKHttp	HTTP+SDPY client
16	Presto	DistributedSQLEngine
17	Metrics	Metrics Framework
18	Spring-boot	AppFramework Wrapper
19	Bukkit	MincraftModAPI
20	Nokgiri	HTML/XML/CSSparser

Table 2 Lab manual code

Sr.#	DesignPattern
1	Adapter
2	Composite
3	Decorator
4	Façade
5	FactoryMethod
6	FlyWeight
7	Iterator
8	Momento
9	Observer
10	Proxy
11	Singleton
12	State
13	Strategy
14	Template
15	Visitor

Preprocessing

First, we generate an XML file for each class code. To obtain the required data for calculating the metrics based approach and performing unstructured data analysis, the source code needs to be translated into a form from which we can extract class variables, class properties, class methods, class names, method names, class description comments, and method description comments.

The SrcML library is used in our technique for this purpose. It supports code translation for many programming languages, including C# and Java. The library supports both datasets used in this study, which are written in Java and C# syntax.

The basic flow of preprocessing is shown in the following Figure 2.

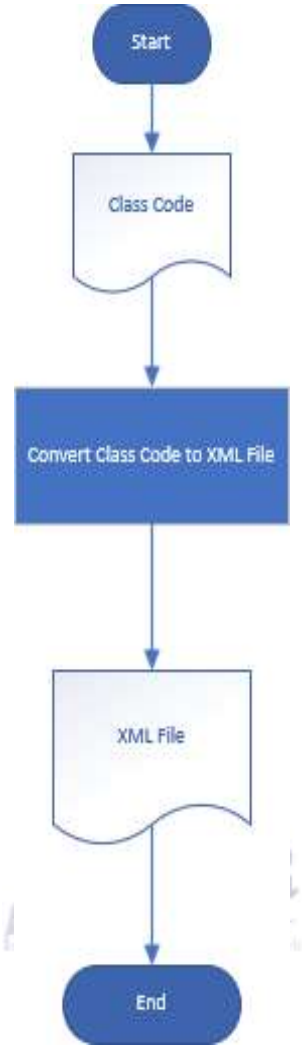

Figure 2 Preprocessing

XML for LCOM calculation

Once XML file is generated, application parse the XML file to get class name, class variables, class properties, methods' names and methods' bodies. After extracting the data, we calculate LCOM with following formula

LCOM = 1 − (a / n), where

a = sum of the count of occurrences of a variable in methods

n = l × j, where

l = number of methods

j = number of variables or properties

Finally, we write LCOM results in a file for future comparison with unstructured data analysis technique. Basic flow of LCOM calculation is shown in following Figure 3;

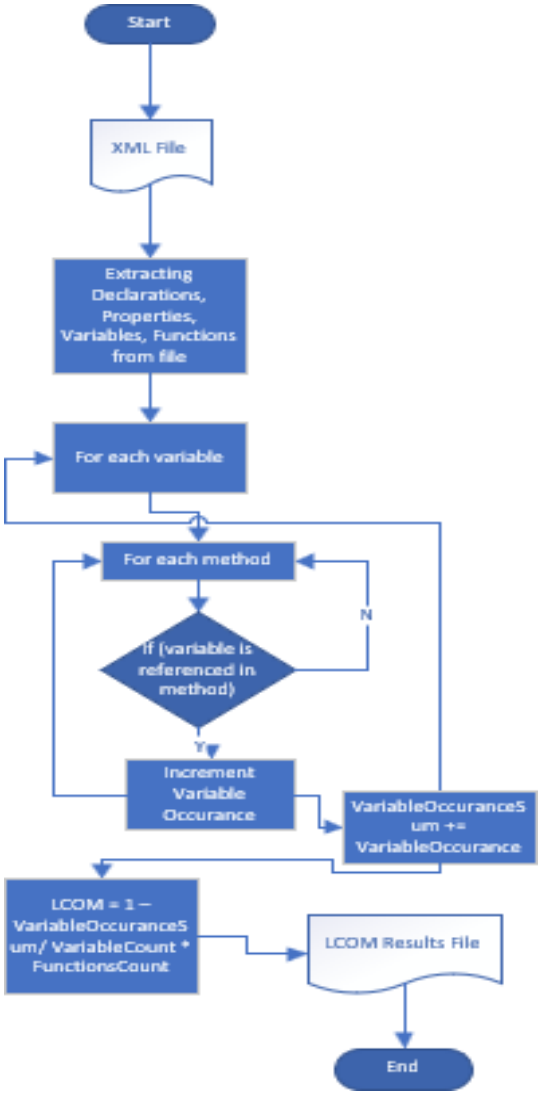

Figure 3 LCOM calculation

Data analysis for finding cohesion

From same XML file our preprocessor engine extracts class names, class descriptions, method names and method descriptions. After extraction this data is saved into a text file for future processing. Basic flow for this step is shown in following Figure 4;

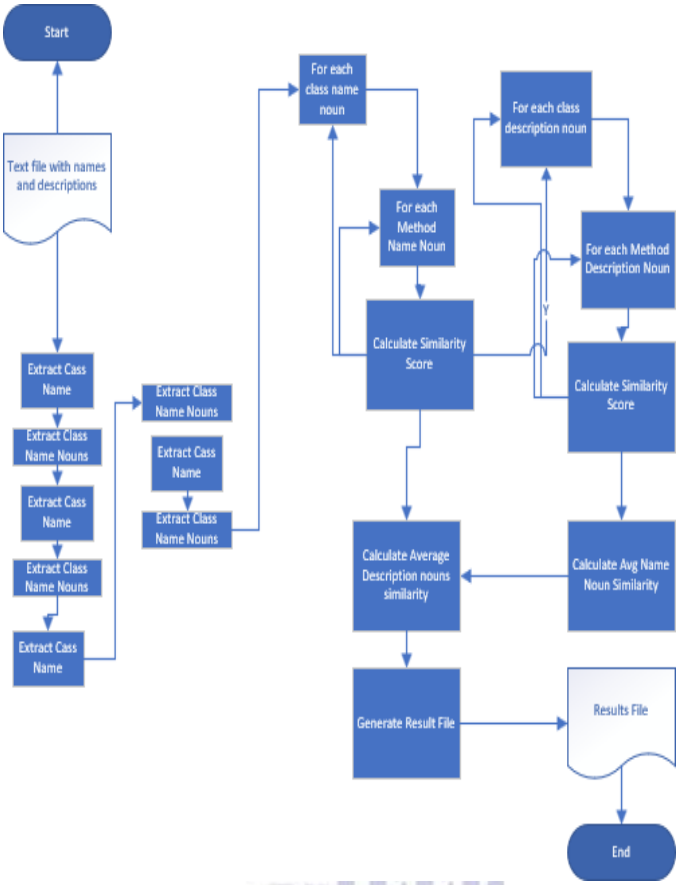

Figure 4 XML Parsing

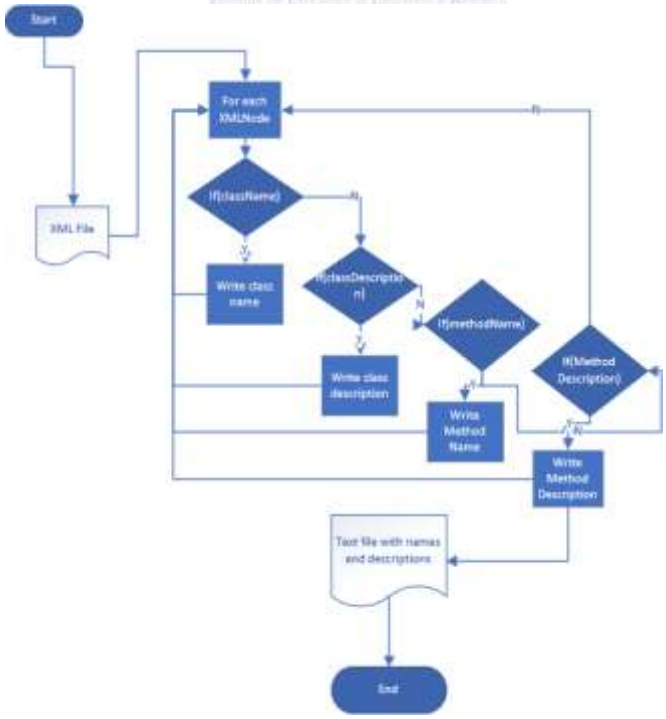

Figure 5 Similarity Score Calculation

Calculation of Similarities Score

From the text file generated, class names, class descriptions, method names, and method descriptions are gathered. It was observed that calculating similarity scores without extracting nouns from class and method names did not yield the expected results. Therefore, nouns were extracted from class names and method names. Similar results were observed for descriptions as well, so nouns were also extracted from class descriptions and method descriptions.

Nouns are collected from class names and method names. For each noun in the class name, its similarity score is calculated with nouns from method names. Similarly, for each noun in the class description, its similarity score is calculated with nouns from method descriptions.

At the end we calculated four output parameters for each class to analyze our results in terms of cohesion of class.

- Average Similarity score of class and method names
- Average Similarity score of class and method description Basic Flow of this step is shown in figure 5 below.

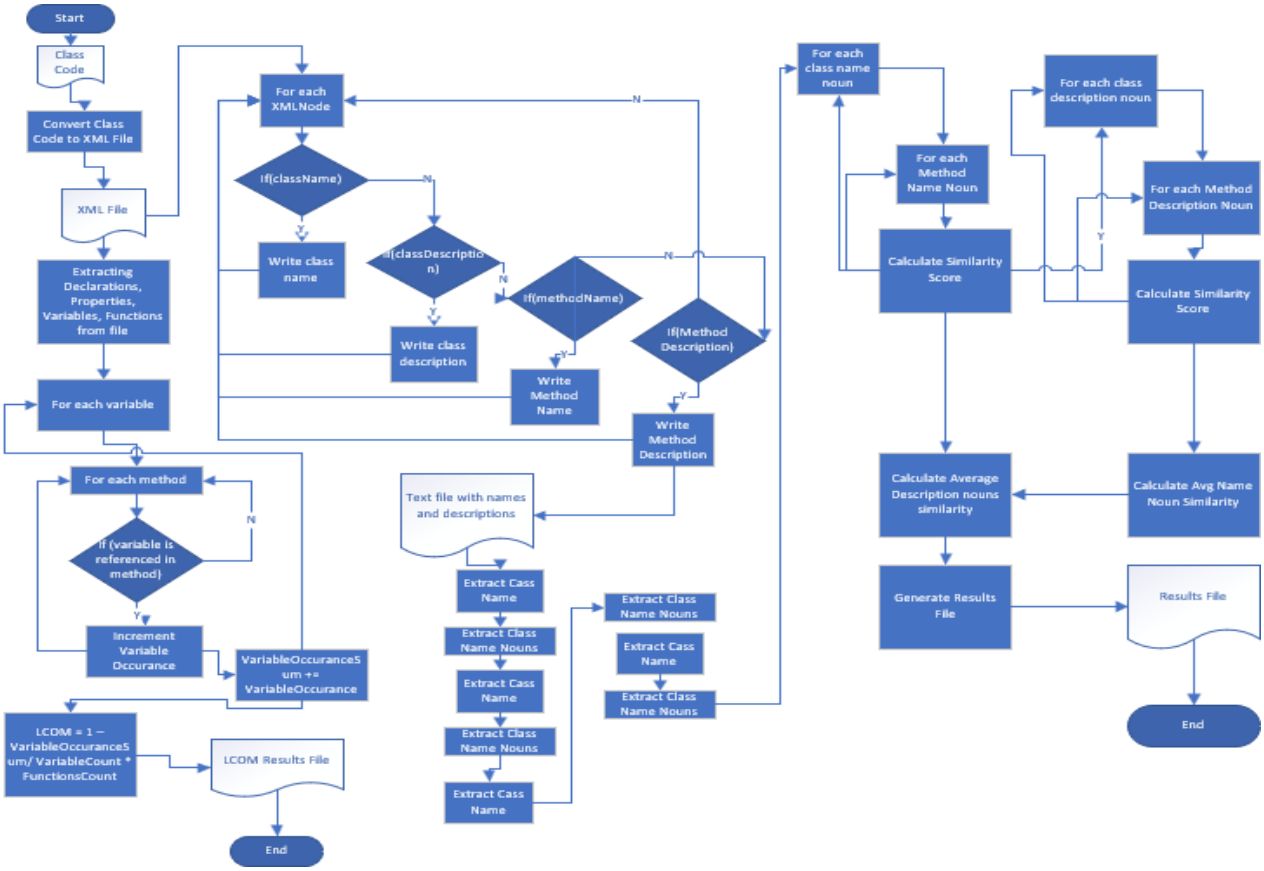

Figure 4 Technique overview

The combine flow chart of all phases of our research technique is given in Figure 6. Our technique calculates similarity scores using a method called cosine similarity. The similarity score is calculated based on the similarity between the vectors of two input texts. To do this, we first convert the input texts into their respective vectors using a pretrained word embedding model. Then, we calculate the cosine similarity between the vectors, which measures the similarity of vector directions in a high dimensional space.

The cosine similarity score ranges from −1 to 1, where −1 indicates completely dissimilar texts and 1 indicates identical texts. A score of 0 indicates that the texts are orthogonal, or completely unrelated. We also normalized the similarity score to a range of 0 to 1, where 0 indicates completely dissimilar texts and 1 indicates identical texts.

1. Results and Discussion

We first ran our technique on sample classes. The classes are divided into two groups. One is the cohesive class group and the second is the non-cohesive class group. The details of the classes are given below Table 3.

Table 3 Sample classes

Cohesive Classes	Non-cohesive classes
Animal, Cat, Dog, Number Calculator	Aero plane, Car, Elephant, Number Manipulator

We also performed LCOM calculation on these sample classes. The class details and obtained results are discussed below.

Animal Class

The Animal class is an abstract class. It has a sound attribute. The code of the class is given below.

```
publicabstractclassAnimal
{
    publicabstractstringSound{get;} public
    Animal(string sound)
    {
    }
publicvirtualvoidMove()
    {
        Console.WriteLine("Moving...");
    }
```

Our results areas follow in table 4:

Table 4 Animal Class Results

LCOM	LackofSimilarityScore
1	0.194588

It was observed that the LCOM technique is unable to produce accurate results in this case. However, our proposed technique has yielded better results than LCOM.

Cat Class

The Cat class is derived from the Animal class. It has a sound attribute. The sound attribute is overridden from the Animal class. The structure of the class is given below.

```
publicoverridestringSound=>"Meow"; public
Cat()
{
}
publicoverridevoidMove()
{
    Console.WriteLine("Walkinglikeacat...");
}
```

Our results areas follow in table 5:

Table 5 Cat Class Results

LCOM	Lack of Similarity Score
1	0.188631

It was observed that the LCOM technique is unable to produce accurate results in this case, while our proposed technique has yielded better results than LCOM.

Dog Class

The Dog class is also derived from the Animal class. It has a sound attribute. The sound attribute is overridden from the Animal class. The structure of the class is given below.

```
publicclassDog:Animal
{
    publicoverridestringSound=>"Woof"; public
    Dog()
    {
    }
    publicoverridevoidMove()
    {
        Console.WriteLine("Runninglikeadog...");
    }
}
```

Our results areas follow in table 6:

Table 6 Dog class Results

LCOM	LackofSimilarityScore
1	0.198654

It was observed that the LCOM technique is unable to produce accurate results in this case, while our proposed technique has yielded better results than LCOM.

NumberCalculator Class

The NumberCalculator class performs basic increment and decrement operations. It has a _number attribute. The class has two methods, **AddOne** and **SubtractOne**, which increment and decrement the _number field respectively.

```
publicclassNumberCalculator
{
    private int _number;
    publicNumberCalculator(){

    }

    publicvoidAddOne(){_number++;}

    publicvoidSubtractOne(){_number--;}
}
```

Our results areas follow in table 7:

Table 7 Number calculator class results

LCOM	Lack of Similarity Score
0.333333	0.244906

It is observed that the LCOM technique is able to find correct results in some cases, while our proposed technique has also yielded better results than LCOM.

Aeroplane Class

The Aeroplane class performs basic operations such as **startEngine**, **setEngineNumber**, and **getSeatsCount**. Each method accesses different properties. From the LCOM perspective, this is considered a low cohesive class. However, from the method and class name semantic perspective, our technique identifies this class as highly cohesive.

```
publicclassAeroplane
{
    privateint engine_number;
    privateint _engine;
    private int _seats;
    public Aeroplane(){

    }
    publicvoidStartEngine(){_engine=1;}
    publicvoidsetEngineNumber(engine_number){_engine_number=engine_number;}
    public void getSeatsCount() { return _seats;}
}
```

Our results areas follow in table 8:

Table 8 Aeroplaneclass results

LCOM	Lack of Similarity Score
0.666667	0.315243

It is observed that the LCOM technique shows this class as less cohesive, while our proposed technique suggests this class as highly cohesive.

Car Class

The Car class has two attributes, **fly** and **run**. These attributes are accessed by two different methods. From the LCOM perspective, this is considered a low cohesive class. From a semantic naming perspective, it should also be considered a low cohesive class. However, it is observed that our technique finds a relatedness score between terms. Even acronyms are assigned a certain degree of relatedness score.

```
publicclassCar

{



    privateboolfly;

    privateboolrun; public

    Car(){



    }

    public void Fly() { fly= true;}

    publicvoidRun(){run=true;}

}
```

Our results areas follow in table 9:

Table 9 Car class results

LCOM	Lack of Similarity Score
0.666667	0.24354

It is observed that the LCOM technique shows this class as less cohesive, while our proposed technique suggests this class as cohesive. In this case, LCOM performs better for the Car class.

Elephant Class

The Elephant class has two attributes, **fly** and **startEngine**. These attributes are accessed by two different methods. From the LCOM perspective, this is considered a low cohesive class. From a semantic naming perspective, it should also be considered a low cohesive class, but it is observed that our technique finds a relatedness score between terms. Even acronyms are assigned a certain level of relatedness score.

```
publicclassElephant
{
    privateboolfly;
    privatebool_startEngine;
    public Elephant(){

    }
    publicvoidStartEgnine(){_startEngine=true;} public
    void Fly() { fly= true;}
}
```

Our results areas follow in table 10:

Table 10 Elephant class results

LCOM	Lack of Similarity Score
0.666667	0.275332

It is observed that the LCOM technique shows this class as less cohesive, while our proposed technique suggests this class as cohesive. In this case, LCOM performs better for the Elephant class.

Number Manipulator Class

The NumberManipulator class has three attributes, **_firstNumber**, **_secondNumber**, and **_thirdNumber**. These attributes are accessed by three different methods. From the LCOM perspective, this is considered a low cohesive class. From a semantic naming perspective, it should be considered a highly cohesive class. Therefore, it is observed that our technique finds a strong relatedness score between the class elements.

```
publicclassNumberManipulator
{
    private int _firstNumber;
    private int _secondNumber;
    private int _thirdNumber;
    publicNumberManipulator(){

    }
    publicvoidIncrementFirst(){_firstNumber++;}
    publicvoidIncrementSecond(){_secondNumber++;} public
    void IncrementThird() { _thirdNumber++;}
}
```

Our results areas follow in table 11:

Table 11 Number Manipulator class results

LCOM	Lack of Similarity Score
0.75	0.414084

It is observed that the LCOM technique shows this class as less cohesive, while our proposed technique suggests this class as somewhat cohesive. In this case, LCOM performs better for the Number Manipulator class.

For each class present in the dataset, LCOM is calculated. For comparison with LCOM, the average similarity between class names and method names is also calculated. Furthermore, the average similarity between class descriptions and method descriptions is also calculated. The details of the obtained results are given below.

- **LCOM vs Average Similarity between Class Name and Method Names**

A total of 6431 classes were processed using LCOM and our proposed technique. The following figure 7 shows the relationship between LCOM and the average similarity of names.

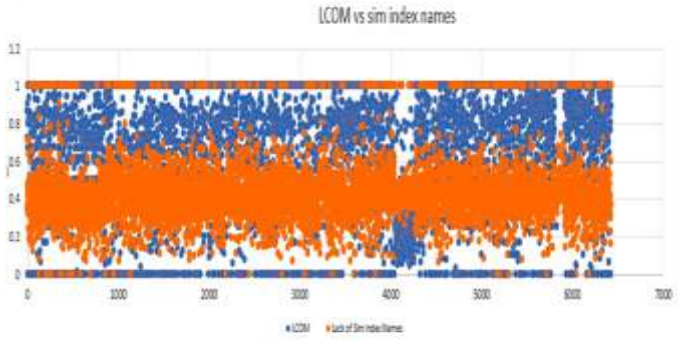

Figure 5 LCOMvs Lackof sim indexnames

LCOM vs Average Similarity between Class and Method Descriptions

Total of 6431 classes are processed through LCOM and our technique. Following figure 8 shows relation between LCOM and average similarity of descriptions.

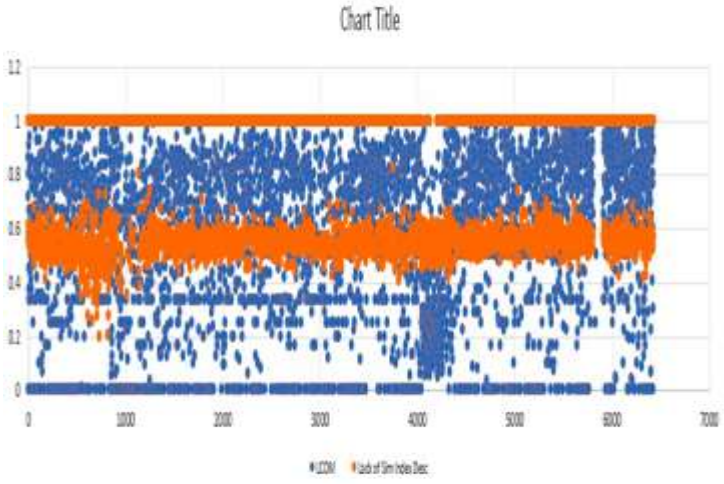

Figure 6 LCOM vs lack of sim index description

Following table 12 how results obtained for dataset

Table 12 Results of github projects

	Total Classes Processed	LCOM<Avg SimIndex	LCOM>Avg SimIndex	LCOM=Avg SimIndex
Similarity between class and method names	6431	1307	4967	157
Similarity between class and method description	6431	2856	2078	1497

Results of Third Dataset (Lab Manual Code)

For each class present in the dataset, LCOM is calculated. For comparison with LCOM, the average similarity between class names and method names is also calculated. Furthermore, the average similarity between class descriptions and method descriptions is also calculated. The details of the obtained results are given below.

- **LCOM vs Average Similarity between Class Name and Method Names**

A total of 81 classes were processed using LCOM and our proposed technique. The following figure 9 shows the relationship between LCOM and the average similarity of names.

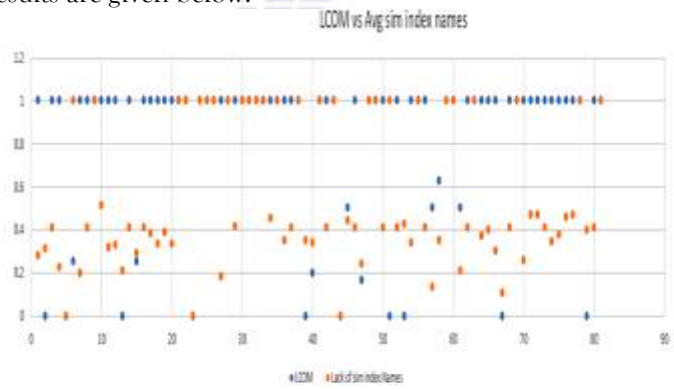

Figure 7 LCOM vs lack of sim index names

- **LCOM vs Average Similarity between Class and Method Descriptions**

A total of 81 classes were processed using LCOM and our proposed technique. The following figure 10 shows the relationship between LCOM and the average similarity of class and method descriptions.

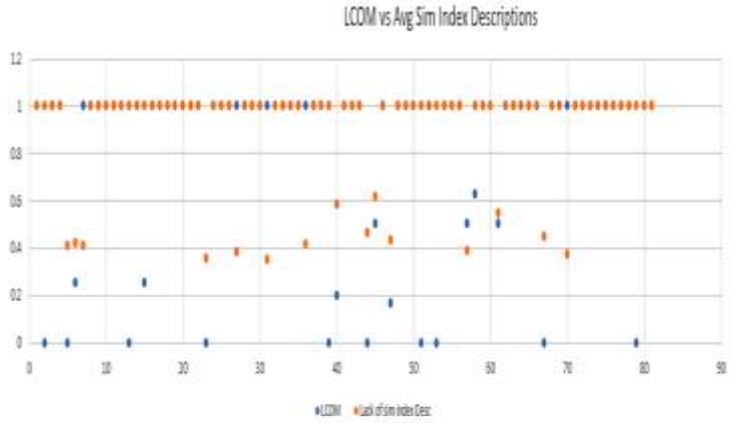

Figure 8 LCOM vs lack of sim index descriptions

Following table 13 how results obtained for dataset
Table 13 Results of Lab Manual Code

	Total Classes Processed	LCOM< Avg SimIndex	LCOM>Avg SimIndex	LCOM= Avg SimIndex
Similarity between class and method names	81	11	43	27
Similarity Between class	81	43	6	32
And method description				

Statistical Testing

The Mann-Whitney U Test was performed on GitHub projects with a significance level of 0.05 and a one tailed hypothesis.

The value of **U = 1745**. The distribution is approximately normal; therefore, the z-score was used. The z-score obtained was **8.07632**. The p-value is **< 0.00001**. The result is statistically significant at **$p < 0.05$**, indicating strong evidence to support the proposed technique over the baseline approach.

Conclusion

In this thesis, we implemented an NLP based technique for a comparative study to examine the cohesion of source code classes. Using experimental design, we collected datasets of source code from two different groups. One dataset was obtained from GitHub projects, while the second dataset was taken from lab manual sample codes used for teaching design patterns in universities. The study was conducted over a period of two semesters (one year), and results were obtained by processing source code using our proposed design technique and a traditional metrics based approach known as LCOM.

The results of this study indicate a significant improvement when cohesion is calculated using unstructured data analysis techniques. However, it was also observed that class description analysis alone is not always optimal for determining class cohesion, as descriptions may contain technical

notes or documentation style text that is more suitable for general NLP frameworks rather than precise software structure analysis.

Overall, our findings suggest that with advancements in machine learning and NLP techniques, unstructured data analysis has strong potential for static code analysis in software engineering. These findings have positive implications for the software engineering community, particularly in the development of intelligent static code analysis tools.

References

[1] "Code Smells Detection and Visualization: A Systematic Literature Review | Archives of Computational Methods in Engineering | Springer Nature Link." Accessed: Feb. 26, 2026. [Online]. Available: https://link.springer.com/article/10.1007/s11831-021-09566-x

[2] A. Gosain and G. Sharma, "A New Metric for Class Cohesion for Object Oriented Software," *Int. Arab J. Inf. Technol.*, vol. 17, no. 3, pp. 411–421, May 2020, doi: 10.34028/iajit/17/3/15.

[3] D. Alexandrov, M. Ismoilov, A. Kozlov, A. Savachenko, and S. Zykov, "Validating New Method for Measuring Cohesion in Object-Oriented Projects," *Procedia Comput. Sci.*, vol. 192, pp. 4865–4876, Jan. 2021, doi: 10.1016/j.procs.2021.09.265.

[4] J. Chen *et al.*, "Utilizing Precise and Complete Code Context to Guide LLM in Automatic False Positive Mitigation," May 31, 2025, *arXiv*: arXiv:2411.03079. doi: 10.48550/arXiv.2411.03079.

[5] J. Al Dallal, "Transitive-based object-oriented lack-of-cohesion metric," *Procedia Comput. Sci.*, vol. 3, pp. 1581–1587, Jan. 2011, doi: 10.1016/j.procs.2011.01.053.

[6] S. Lee, "Understanding Lack of Cohesion in Methods." Accessed: Feb. 26, 2026. [Online]. Available: https://www.numberanalytics.com/blog/ultimate-guide-lack-cohesion-methods-software-metrics

[7] "Understanding LCOM Metric: Ensuring Single Responsibility in OOP | Suraif Muhammad | Medium." Accessed: Feb. 26, 2026. [Online]. Available: https://medium.com/@suraif16/lack-of-cohesion-in-methods-265a9a26fd66

[8] Shipra, "Cohesion measurements between variables and methods using component-based software systems," *Int. J. Syst. Assur. Eng. Manag.*, vol. 15, no. 7, pp. 3242–3250, Jul. 2024, doi: 10.1007/s13198-024-02331-w.

[9] "What is Code Quality? Definition Guide & How-to Improve and Measure." Accessed: Feb. 26, 2026. [Online]. Available: https://www.sonarsource.com/resources/library/code-quality-2026/

[10] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," 1995. Accessed: Feb. 26, 2026. [Online]. Available: https://www.semanticscholar.org/paper/Measuring-coupling-and-cohesion-in-object-oriented-Hitz-Montazeri/a85a0b5b7a33c5c7ce305dfcea775026fc52e6f9

[11] T. CodeReliant, "The Principle of High Cohesion: A Pillar of Reliable Software Design." Accessed: Feb. 26, 2026. [Online]. Available: https://www.codereliant.io/p/the-principle-of-high-cohesion-a-pillar-of-reliable-software-design

[12] "A Systematic Literature Review and Quality Analysis of Javascript Malware Detection | IEEE Journals & Magazine | IEEE Xplore." Accessed: Feb. 26, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9226527

[13] "A Large-Scale Empirical Comparison of Object-Oriented Cohesion Metrics | IEEE Conference Publication | IEEE Xplore." Accessed: Feb. 26, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/4425882

[14] T. Welzer, S. Yamamoto, and I. Rozman, *Knowledge-based Software Engineering: Proceedings of the Fifth Joint Conference on Knowledge-Based Software Engineering*. IOS Press, 2002.

[15] "A design-based cohesion metric for object-oriented classes | Request PDF," *ResearchGate*, Accessed: Feb. 26, 2026. [Online]. Available: https://www.researchgate.net/publication/284026416_A_design-based_cohesion_metric_for_object-oriented_classes

[16] S. Counsell, S. Swift, and J. Crampton, "The interpretation and utility of three cohesion metrics for object-oriented design," *ACM Trans Softw Eng Methodol*, vol. 15, no. 2, pp. 123–149, Apr. 2006, doi: 10.1145/1131421.1131422.

[17] "Measuring class cohesion in object-oriented systems - ProQuest." Accessed: Feb. 26, 2026. [Online]. Available: https://www.proquest.com/openview/1a7e82c1c7c606b750c9028a90622a62/1?pq-origsite=gscholar&cbl=18750&diss=y

[18] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, "Machine learning based methods for software fault prediction: A survey," *Expert Syst. Appl.*, vol. 172, p. 114595, Jun. 2021, doi: 10.1016/j.eswa.2021.114595.

[19] C.-Y. Chen, K.-Y. Tai, and S.-S. Chong, "Quality Evaluation of Structural Design in Software Reverse Engineering: A Focus On Cohesion," *IEEE Access*, vol. 9, pp. 109569–109583, 2021, doi: 10.1109/ACCESS.2021.3102295.

[20] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," *SIGSOFT Softw Eng Notes*, vol. 20, no. SI, pp. 259–262, Aug. 1995, doi: 10.1145/223427.211856.