Examining Software Coupling and Cohesion Patterns using Social Network Analysis

Mohamed Maddeh
College of Applied Computer Science, King Saud University,
Riyadh 11451, Saudi Arabia.
Higher Institute of Finance and Taxation Sousse,
University of Sousse, Sousse 4023, Tunisia

ABSTRACT
Social network analysis (SNA) is an emerging research area that has gained significant attention in recent years. Analyzing OO program through SNA can provide insights into how a program component, classes and methods interact and collaborate. In fact, an OO program is composed of a set of classes that interact with each other. Considering a class as a node and the interaction as a relationship, we can take advantage from SNA capabilities to the benefit of OO programming. Therefore, SNA is an excellent way for detecting and quantifying coupling and cohesion in an Object Oriented Programming (OOP) based on the class interaction, by analyzing the connections between classes and methods. An accurate coupling and cohesion detection helps developers to optimize codes and improve its overall performance and maintainability. In this paper, we represent four java open source projects (JUnit 5.10.2, Spring 6.1.4, Apache Commons BCEL 6.8.2 and Guava 33.0) as a social network. We also, applied SNA techniques to identify lowly cohesive classes and highly coupled classes.

Keywords Object Oriented Programming, Coupling, Cohesion, Social Network Analysis, Refactoring, Maintainability.

1. INTRODUCTION
Social network analysis (SNA) is a technique for representing and analyzing the relationships between individuals or entities in a network. It is commonly used in fields such as sociology, anthropology, and marketing to understand social dynamics, collaboration patterns, and information diffusion. SNA [1] [2] can also be applied to object-oriented software systems to gain insights into their structure and dynamics. By viewing class interactions as a social network, we can identify key classes and components, detect communities and modules, and analyze information flow and dependencies. SNA measures such as degree centrality and betweenness centrality can be used to identify classes that have a high number of connections or play a critical role in information flow. Network visualization tools such as Gephi [3] can be used to visualize the class interaction network and identify patterns and anomalies.

This innovative approach offers a valuable alternative to conventional methods of detecting class coupling and cohesion. [4] [5] [6] [7], and offers new insights into the design and maintenance of object-oriented software systems. In fact, in the field of software engineering, various metrics are employed to evaluate the coupling and cohesion of classes in object-oriented systems. These metrics assist developers in identifying and refactoring code to enhance its modularity, reusability, and maintainability. For instance, coupling metrics include [8] [9] [10] [11]: Coupling Between Objects (CBO), it measures the number of other classes that a class is coupled to. Depth of Inheritance Tree (DIT), it measures the depth of the inheritance hierarchy in which a class resides. Number of Children (NOC) it measures the number of subclasses that a class has. Cohesion metrics include: Lack of Cohesion in Methods (LCOM), it measures the number of methods in a class that do not access the same instance variables. Coupling Between Methods (CBM), it measures the number of pairs of methods in a class that access the same instance variables.

The challenge lies in the multitude of metrics and their associated thresholds, which can complicate the measurement process [12] [13] [14] [15]. This new approach of viewing class interactions as a social network offers a promising alternative to traditional coupling and cohesion metrics. By leveraging SNA techniques, we can gain a more comprehensive understanding of the relationships between classes and identify potential areas for improvement. It can help reduce the number of classes that need to be studied, as the most problematic areas that can be targeted first. This can make the process of refactoring and improving the system more efficient and effective. Avoiding to manually inspect every class, which can be very time-consuming for large codebases. Only the top-ranked classes from the SNA prioritization need to be studied in depth, reducing the overall number of classes that require detailed analysis.

2. LITERATURE REVIEW
The concepts of coupling and cohesion have long been recognized as fundamental pillars of high-quality, maintainable code. As software systems grow in complexity, the balance between these two design principles becomes increasingly critical to ensure that code is not only efficient and effective but also easy to understand, modify, and extend. In their study [16], the authors conducted a systematic mapping to determine the commonly used coupling and cohesion metrics and their practical applicability. They found four distinct categories, evolution of coupling and cohesion metrics, research type, contribution, and context focus. This categorization allowed for a structured analysis of the existing body of research. The work presented in [17] provides a comprehensive framework to deal with all sorts of coupling. It propose a framework that takes into account the distinction between object level-and class level coupling. This distinction refers to dynamic dependencies between objects on one hand and static dependencies between implementations on the other hand. In [18] researchers focused on improving software design quality, reliability, and reducing complexity in component-based software engineering. The paper proposes a component selection framework that utilizes the Hexa-oval optimization algorithm to select the most suitable components from a repository. This framework aims to analyze the relationship between component modules by measuring their coupling and cohesion. Another work presented in [19] propose a framework for calculating hybrid system metrics in software quality metrics, specifically focusing on aspect-oriented and object-oriented programming. The paper emphasizes the importance of both static and dynamic software metrics for a comprehensive evaluation of software quality. In [5] the paper proposes an automated approach to measure and visualize class cohesion in object-oriented systems. Traditionally, measuring cohesion has been a manual and time-consuming process for software engineers and developers. The proposed

approach aims to overcome these challenges by automatically measuring cohesion, which can provide a more efficient and interactive way to assess software quality. The approach works by parsing the program source code into an XML file using an existing tool and then extracting class tokens based on the definitions of cohesion metrics. It then identifies cohesion relationships by comparing these tokens with class features and generates interactive visualizations of the cohesion using various charts. Authors in [4] proposes a new source code level class cohesion metric for Object Oriented (OO) software. The paper addresses the limitations of existing cohesion metrics, which the authors argue do not adequately capture the cohesiveness of classes. The proposed metric is based on the usage of instance variables by methods within a class. In [20] authors focus on the use of code refactoring [21] as a strategy for improving the internal structure of software systems without changing their external behavior. The paper addresses the issue of software maintenance and the degradation of software systems' internal structures over time due to maintenance operations. The paper proposes the use of object-oriented metrics, specifically cohesion metrics, to assess the quality of object-oriented classes and to guide the decision-making process for code refactoring [22] [23]. Work in [24] focuses on class cohesion as a critical factor in the quality assurance of object-oriented software. The abstract mentions that there are over thirty different metrics to measure cohesion, which are based on the analysis of class members such as attributes and methods. The study aims to utilize these metrics to promote the quality of Java code static analysis, improve object-oriented programming practices, and suggest more advanced and efficient practices.

3. COUPLING AND COHESION

Coupling and cohesion are two important concepts in OOP that describe the relationships among classes. In fact, the degree of connectivity between classes in an object-oriented system measures how closely one class interacts with other classes. Class coupling includes method invocations accessing data members, attributes and methods in other classes.

Cohesion measures how well the methods within a class are logically grouped and organized. Cohesion insure that we create classes with the right methods and attributes. In this paper, we focus on functional cohesion, which means that a class encapsulates single, well-defined functions that are highly interconnected. They also, depend on the data members of that class. High cohesion means that a class has a clear and specific purpose, while low cohesion suggests that a class performs multiple unrelated tasks. Low cohesion is synonym of bad design, methods and/or attributes are not defined in the appropriate class.

Coupling and cohesion are inversely proportional, a low coupling induces a high cohesion and vice versa. In an object oriented analysis and design (OOAD), it is important to decrease the coupling to enhance flexibility, maintainability, and reusability and guaranty independent development. A loose coupling reduces the scope of software modifications, making the maintainability, and reusability easier. It is also important to maintain a high cohesion in class design that enhances readability, reusability, and maintainability.

4. SOCIAL NETWORK ANALYSIS FOR OBJECT ORIENTED PROGRAM

Social Network Analysis (SNA) studies social structures and relationships within a network of individuals, organizations, or other entities based on the patterns of connections, interactions, and information flow among the members of a network. SNA focuses on understanding the structure of social networks, the relationships between network nodes, and the influence and information flow within the network. It can help uncover hidden patterns, identify key actors or influencers, measure centrality and connectivity, and explore how information, resources, and behaviors spread through the network.

In this paper, we represent SNA as graph, where nodes represent classes, and edges represent the associations. In fact, classes are active entity that communicates with each other, they access data members of other classes, and they provides data and methods for other classes. They could be used as a data type for other classes. As matter of fact, a UML class diagram can be likened to a social network, where classes represent individuals and their relationships depict connections between them. The different is that in SNA we focus on interaction rather that the structure of the classes. The objective of this work is to quantify this interaction to the benefit of OOAD, especially coupling and cohesion measurements.

In SNA we differentiate between directed and undirected relationship. The relationship can be weighted or unweighted. For an OOP we consider the association a directed relationship. A class C1 can access a class C2, in this case the flow is from C1 to C2. If the flow is from C2 to C1, it means that C1 is accessed by C2. The relationship is also weighted to measure the number of occurrence each class access other classes or is accessed by other classes.

In Table 1, we present the mapping between UML class diagram and SNA graph:

Table 1 mapping UML class diagram and SNA

UML	Properties	SNA	Properties
Class	Attributes Methods	Nodes	Number of accessed attributes Number of accessed methods Number of usage as datatype
Association	Multiplicity	Relationship	Directed weighted

If we consider the example presented in Figure 1, the class diagram contains four classes. In SNA each class is a node in the graph, the edges weight shows the number of communication between classes as presented in Table 1. This graph is based on the adjacency matrix in Table 2.

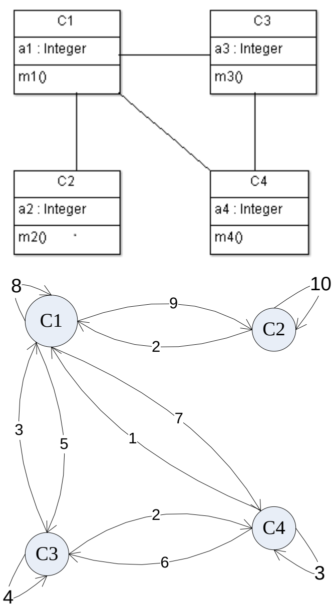


Fig 1. Mapping UML Class diagram to SNA graph

In Table 2, we have the Adjacency matrix representing all communications between classes. For example, the class C1 communicates with itself eight times and it sends and it access C2 eight times ,C3 five times and C4 seven times. C1 is accessed by C2 two times, C 3 three times and C4 one time.

Table 2. Adjacency matrix

	C1	C2	C3	C4
C1	8	9	5	7
C2	2	10	0	0
C3	3	0	4	2
C4	1	0	6	3

When considering class diagram as SNA we can identify many OOP concepts throw the measurement of nodes importance like degree centrality, it measures the number of connections a node has to other nodes in the network. Closeness centrality (CC) [1], it measures the distance between a node and all other nodes in the network. Betweenness centrality (BC) [1], it measures the number of shortest paths between all other pairs of nodes that pass through a given node. Eigenvector centrality (EC) algorithm measures the influence or importance of a node based on its connections to other influential nodes.

5. MODELLING COUPLING AND COHESION AS SOCIAL NETWORK INTERACTION

Modeling coupling and cohesion as social network interactions provides an interesting perspective on how the OOP design principles are defined. Figure 2 illustrates the methodology for applying social network analysis (SNA) to detect coupling and cohesion.

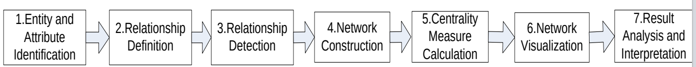


Fig 2. Methodology for applying social network analysis

Entity and Attribute Identification: Define the Java classes as the nodes in the network. For each class, consider the number of attributes and methods accessed by other classes as relevant attributes.

Relationship Definition: Establish relationships by counting the number of times one class accesses attributes and/or methods from another class. Additionally, count the instances where one class is used as a data type in another class. The relationships are directed to identify class dependencies.

Relationship Detection: The Java program we developed extracts and inspects all classes in a given Java project. This program generates an Excel file containing the list of edges (relationships) between classes.

Network Construction: In this paper, we use the Gephi tool to import the Excel file generated in step 3. This tool will map the entities (classes) as nodes and the relationships as edges, creating the network representation.

Centrality Measure Calculation [2]: Compute various centrality measures (degree, betweenness, closeness, eigenvector) to identify influential or highly connected nodes (classes).

Network Visualization: The Gephi's network visualization tools represents the network graphically, highlighting patterns of coupling and cohesion visually.

Result Analysis and Interpretation: We focus on the emergent classes (highly connected or influential nodes) and analyze the causes of coupling and cohesion defects. Based on this analysis, we could generate a list of refactoring recommendations to improve the OOP.

6. VALIDATION

6.1 Project characteristics

To validate our approach of modeling class interactions as a social network, we conducted our research on four well-established open-source Java projects: JUnit 5.10.2, Spring 6.1.4, Apache Commons BCEL 6.8.2, and Guava 33.0. The primary objective was to identify classes that exhibit low cohesion or high coupling, thereby indicating potential candidate classes for refactoring. These classes, referred to as "suspect classes," are characterized by their lack of cohesive functionality or excessive dependencies on other classes. Table 3 presents the key characteristics of the selected projects. We intentionally chose projects with diverse characteristics, ranging from small-scale to medium-sized and large-scale projects. This diversity in project sizes and complexities allows for a comprehensive evaluation of our approach across various software system scales.

Table 3. Project characteristics

Project	Version	Classes	Relationships
JUnit	5.10.2	1014	2845
Spring	6.1.4	4070	14118
BCEL	6.8.2	585	3871
Guava	33.0	946	2002

After following the procedure outlined in Figure 2, we developed a Java program that examines the root directory of any Java project, Figure 3. The program comprises three modules.

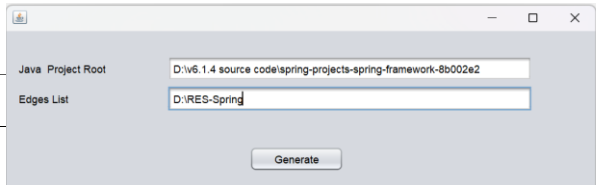
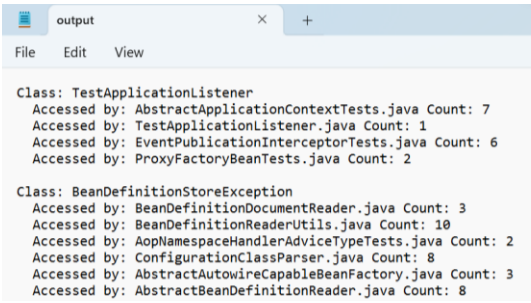


Fig 3. Main Interface

The first module includes the "ClassExtractor" class, which accepts the root directory of the Java project as input and returns the extracted class names for the project. The second module contains the "ClassUsageCounter" class, which counts the number of classes that access each extracted class, including itself. As illustrated in Figure 4, it also generates a text file that lists the accessing classes for each class, along with the corresponding access counts.



```
Class: TestApplicationListener
  Accessed by: AbstractApplicationContextTests.java Count: 7
  Accessed by: TestApplicationListener.java Count: 1
  Accessed by: EventPublicationInterceptorTests.java Count: 6
  Accessed by: ProxyFactoryBeanTests.java Count: 2

Class: BeanDefinitionStoreException
  Accessed by: BeanDefinitionDocumentReader.java Count: 3
  Accessed by: BeanDefinitionReaderUtils.java Count: 10
  Accessed by: AopNamespaceHandlerAdviceTypeTests.java Count: 2
  Accessed by: ConfigurationClassParser.java Count: 8
  Accessed by: AbstractAutowireCapableBeanFactory.java Count: 3
  Accessed by: AbstractBeanDefinitionReader.java Count: 8
```

Fig 4. Access count text file

The third module incorporates the "GenerateExcelFile" class, which takes the text file and an empty Excel file as inputs and generates an edges list for the Gephi tool in the Excel file, as presented in Figure 5.

	A	B	C
1	Source	Target	Cou
2	TestApplicationListener	AbstractApplicationContextTests	
3	TestApplicationListener	TestApplicationListener	
4	TestApplicationListener	EventPublicationInterceptorTests	
5	TestApplicationListener	ProxyFactoryBeanTests	
6	BeanDefinitionStoreException	BeanDefinitionDocumentReader	
7	BeanDefinitionStoreException	BeanDefinitionReaderUtils	
8	BeanDefinitionStoreException	AopNamespaceHandlerAdviceTypeTests	
9	BeanDefinitionStoreException	ConfigurationClassParser	
10	BeanDefinitionStoreException	AbstractAutowireCapableBeanFactory	
11	BeanDefinitionStoreException	AbstractBeanDefinitionReader	
12	BeanDefinitionStoreException	PropertyResourceConfigurerTests	
13	BeanDefinitionStoreException	ConfigurableBeanFactory	
14	BeanDefinitionStoreException	XmlReaderContext	

Fig 5.Excel edges list for the Gephi tool

6.2 Limits of identifying suspect classes based on degree centrality

In our work, coupling and cohesion in software design are evaluated using degree centrality from social network analysis. Degree centrality gives the number of connections or communication paths between classes. A class diagram can be represented as a directed network, where in-degree centrality measures how many classes access or use a given class. Out-degree centrality measures the number of communication from a class to others classes. A class with high degree centrality, meaning it has many incoming and/or outgoing connections, likely suffers from high coupling and low cohesion issues. High coupling indicates that the class is heavily dependent on many other classes, while low cohesion means that the class's responsibilities are spread out across multiple unrelated functions. In Figure 6, we present the classes with highest degree centrality, the colors range from green to yellow, purple, and blue. The nodes depicted in blue signify the highest degree centrality.

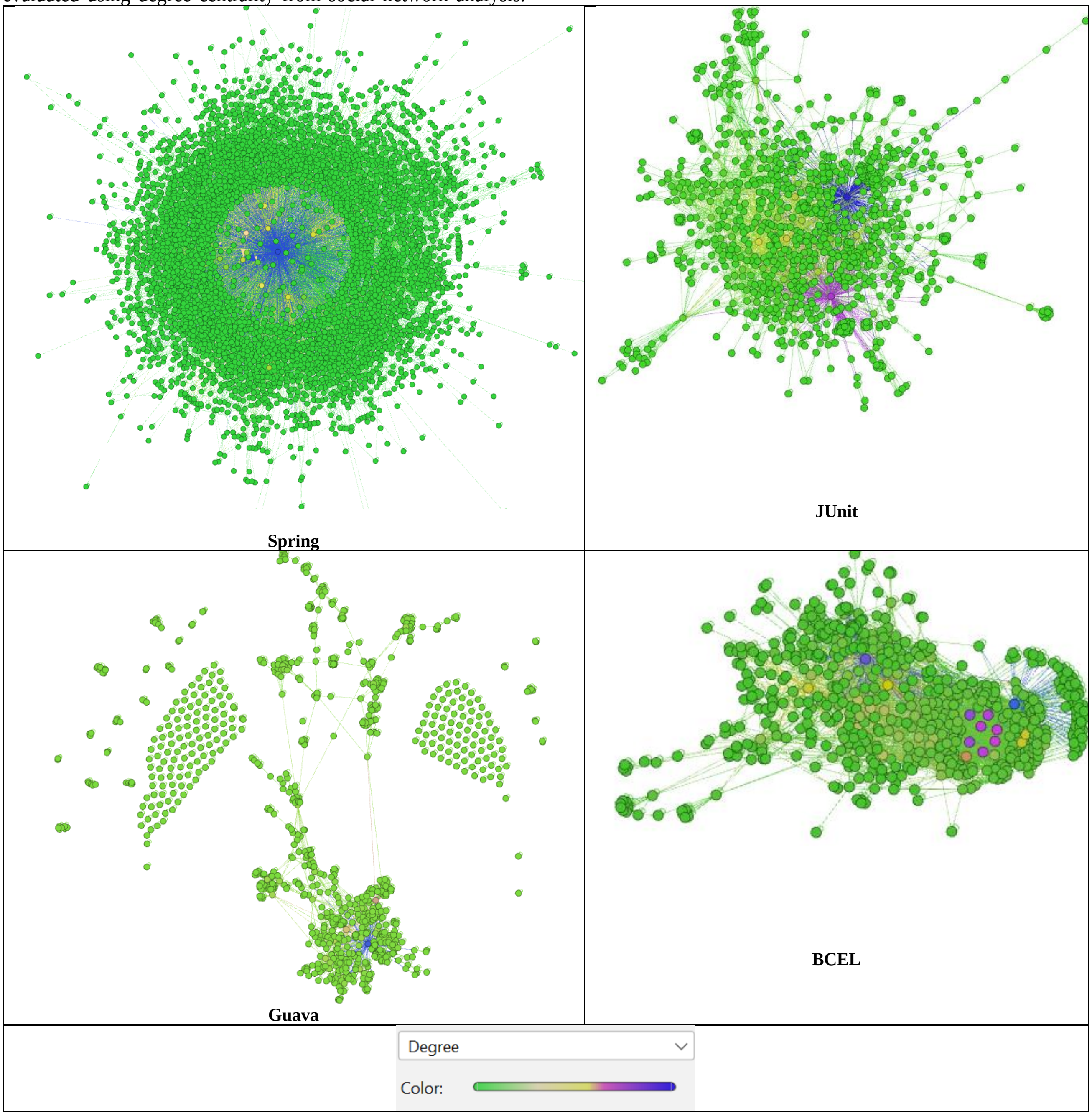


Fig 6. Visualization of the classes with highest degree centrality

Table 4 illustrates an example from the Spring project, showcasing the application of the degree centrality measure. This measure provides a ranking of the most connected classes within the project. Notably, it identifies classes like TestBean and IllegalStateException present a high degree, each having a degree of 3723 and 1166 respectively. However, these classes exhibit an unbalanced in and out degree, as for class TestBean the in-degree is 3714 compared to its out-degree of 9 and for class

IllegalStateException the in-degree is 1161 compared to its out-degree of 5. This issue of unbalanced connectivity is prevalent among the highest-ranking classes in the table.

Table 4. Spring project degree centrality simple

Id	Weighted In-Degr...	Weighted Out-De...	Weighted D...	Closeness Centrali...	Betweenness Centr...	Eigenvector Central...
TestBean	3714.0	9.0	3723.0	0.8	0.000065	0.252413
RootBeanDefinition	2530.0	60.0	2590.0	0.09032	0.001399	0.255954
ResolvableType	1738.0	262.0	2000.0	0.098271	0.027346	0.328543
DefaultListableBeanFactory	1315.0	189.0	1504.0	0.105986	0.005911	0.263958
TypeDescriptor	1232.0	118.0	1350.0	0.089645	0.014746	0.21789
IllegalStateException	1161.0	5.0	1166.0	0.0	0.0	1.0
DefaultListableBeanFactoryTests	0.0	1110.0	1110.0	0.121511	0.0	0.0
AnnotationConfigApplicationContext	979.0	37.0	1016.0	0.133046	0.026962	0.18305
Constants	856.0	31.0	887.0	0.470588	0.000132	0.135712
GenericApplicationContext	820.0	57.0	877.0	0.124388	0.011845	0.161866
MethodParameter	774.0	57.0	831.0	0.089571	0.002392	0.249013
MimeType	719.0	47.0	766.0	0.666667	0.000028	0.129021
MutablePropertyValues	669.0	68.0	737.0	0.5	0.000139	0.109594
AutowiredAnnotationBeanPostProcesso...	0.0	717.0	717.0	0.167298	0.0	0.0
XmlBeanFactoryTests	0.0	672.0	672.0	0.113683	0.0	0.0
DataBinderTests	0.0	635.0	635.0	0.107757	0.0	0.0

In certain scenarios, high coupling is not a design flaw but an intentional and unavoidable consequence of the specific purpose and requirements of particular classes or modules. This is evident in the examples of the TestBean class from the Spring Framework and the Attribute class for bytecode manipulation.

The TestBean class is not intended for real-world business purposes but serves as a comprehensive utility for testing the Spring Framework's features, such as dependency injection, type conversion, bean lifecycle management, and container callbacks. While high coupling is generally discouraged in production code due to maintainability concerns, in the case of the TestBean class within the test suite, this high coupling is deliberate and justified by the need for thorough framework testing. Importantly, this high coupling is isolated within the test suite and does not affect the loose coupling and modular design principles promoted by the Spring Framework for application code through dependency injection and inversion of control.

Similarly, the IllegalStateException class in the Spring Java Message Service (JMS) package appears to be a critical component for managing and handling JMS-related exceptions. Its high in-degree suggests that it plays a central role in the error handling strategy of the Spring framework, providing a consistent and flexible way to deal with JMS errors across different parts of the application.

Additionally, we observe that the class AutowiredAnnotationBeanPostProcessorTests has an in degree value of null. This null value occurs because the class is responsible for instantiating beans, registering them within the bean factory, and conducting assertion tests to ensure the proper functioning of the AutowiredAnnotationBeanPostProcessor class. Consequently, the class AutowiredAnnotationBeanPostProcessorTests actively interacts with and relies on other classes from both the Spring Framework and JUnit to carry out its testing responsibilities. This class uses other classes without being accessed by any other classes.

These examples clearly demonstrate that in certain cases, high coupling and unbalanced in/out degree is not a design defect [3] [4] [5] but an intentional and unavoidable consequence employed by developers to fulfill specific requirements or purposes. As discussed in the next section, the key is to isolate and encapsulate this high coupling within the necessary components, while ensuring that the rest of the system adheres to loose coupling and modular design principles for maintainability and extensibility.

6.3 Refining suspect class detection

When examining the issue of unbalanced classes, one notable finding is that these classes exhibit a very low CC and/or BC and/or EC values. In fact, BC is a measure of a class's position as a bridge or intermediary between other classes in the program. Classes with high BC play a crucial role in connecting different program components and can represent potential bottlenecks or critical communication points between various clusters or components. Identifying such nodes can help identify areas of potential coupling or lack of cohesion within the program. CC is a measure of how close a class is to all other classes in the diagram. Classes with high CC are considered to be central and therefore more important than other classes.

Eigenvector centrality, on the other hand, is an algorithm that assesses the importance or influence of classes in the class diagram based on their communication patterns. It assigns higher scores to classes that are linked to other significant classes. EC helps identifying classes with a substantial impact within the class diagram. Classes with high EC scores indicate central or influential components, which can be indicative of strong cohesion or coupling within those components.

Figure 7 illustrates the social network visualizations of class interactions for the four chosen projects, utilizing the concept of BC.

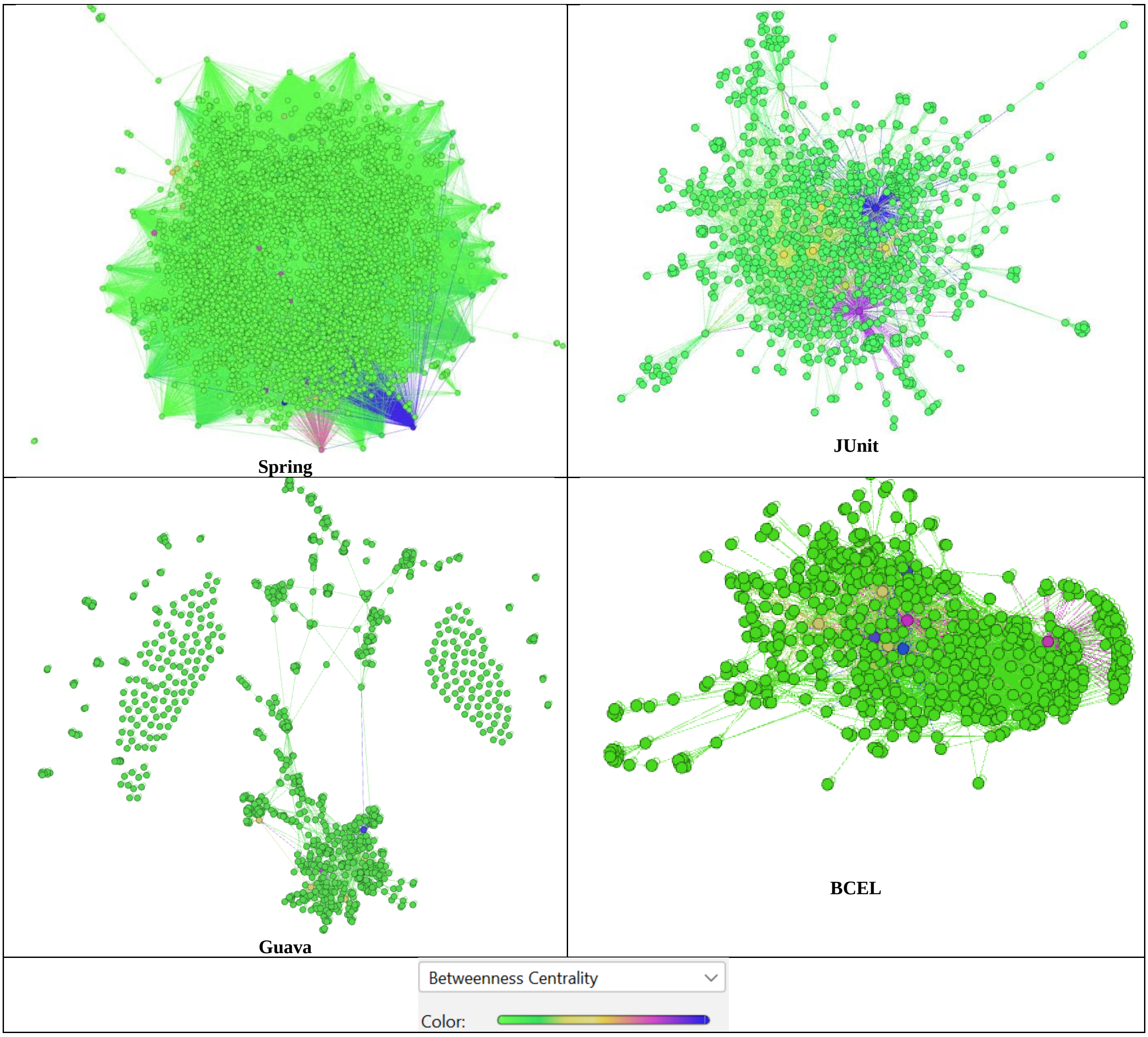
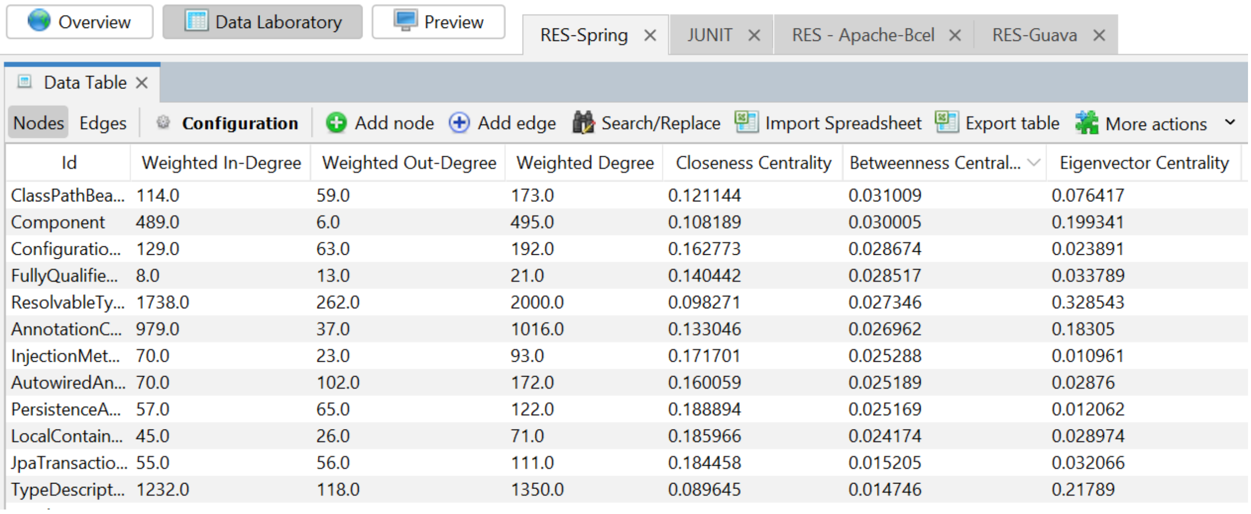


Fig 7. Visualizations of class interactions based on BC

Classes with highest BC are shown in blue, purple, and yellow. Class in green has the lowest BC.

Table 5 provides an illustration of the spring project, where the classes are listed in descending order of their BC. Before detecting the suspect classes using weighted degree centrality, and assess the class coupling and cohesion through the analysis of in and out weighted degrees.

Table 5. Spring project BC simple



Overview | Data Laboratory | Preview | RES-Spring | JUNIT | RES - Apache-Bcel | RES-Guava

Data Table

Nodes Edges Configuration Add node Add edge Search/Replace Import Spreadsheet Export table More actions

Id	Weighted In-Degree	Weighted Out-Degree	Weighted Degree	Closeness Centrality	Betweenness Central...	Eigenvector Centrality
ClassPathBea...	114.0	59.0	173.0	0.121144	0.031009	0.076417
Component	489.0	6.0	495.0	0.108189	0.030005	0.199341
Configuratio...	129.0	63.0	192.0	0.162773	0.028674	0.023891
FullyQualifie...	8.0	13.0	21.0	0.140442	0.028517	0.033789
ResolvableTy...	1738.0	262.0	2000.0	0.098271	0.027346	0.328543
AnnotationC...	979.0	37.0	1016.0	0.133046	0.026962	0.18305
InjectionMet...	70.0	23.0	93.0	0.171701	0.025288	0.010961
AutowiredAn...	70.0	102.0	172.0	0.160059	0.025189	0.02876
PersistenceA...	57.0	65.0	122.0	0.188894	0.025169	0.012062
LocalContain...	45.0	26.0	71.0	0.185966	0.024174	0.028974
JpaTransactio...	55.0	56.0	111.0	0.184458	0.015205	0.032066
TypeDescript...	1232.0	118.0	1350.0	0.089645	0.014746	0.21789

To refine our selection, we apply filters that are based on BC, CC , and EC. This strategy allows us to omit classes that were designed purely for programming needs, directing our focus to classes that are most pertinent to the business logic. Our analysis is centered on the classes that are deemed most significant, as they embody a balanced degree they also represent the business logic. The importance of these classes is determined by their ranking in one of the three metrics: BC, CC , and EC.

7. RESULTS AND DISCUSSION

The outcomes of the network analysis are examined to comprehend the program's architecture and to pinpoint classes that may be problematic. As previously mentioned, we aim to extract classes with a high degree centrality. We then evaluate these classes based on their BC, CC, and EC.

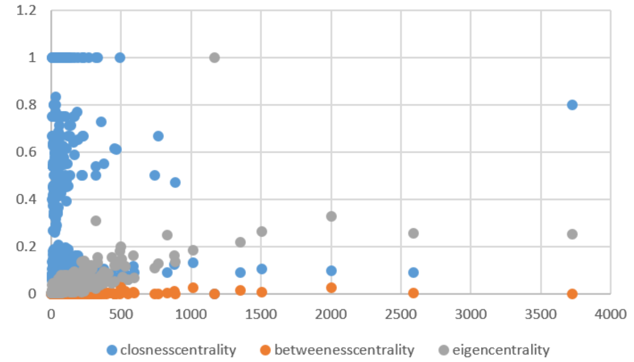


Fig 8. Metrics distribution for the Spring Project

For each of these metrics, we seek the optimal threshold to fine-tune the detection process. Figure 8 illustrates the distribution of each metric. After examining various threshold values, we select the third quartile as an acceptable threshold for filtering out classes with the highest degree of centrality. Consequently, for the Spring project the thresholds for the BC, CC , and EC metrics are set at 0.174521, 0.0000695, and 0.0188085, respectively. In table 6 we present the thresholds for each project.

Table 6. Project thresholds

Project	CC threshold	BC threshold	EC threshold
JUnit	0.714286	0.00001925	0.03126
Spring	0.174521	0.0000695	0.0188085
BCEL	0.4772205	0.000625	0.11379
Guava	1	0.000004	0.024199

Once the thresholds for each metric have been established, and given the average weighted degree, we can concentrate on classes with a high degree of connectivity to identify both coupling and cohesion. Classes with a high weighted degree might exhibit a high degree of coupling and low cohesion. Conversely, classes with a low weighted in degree might show signs of low cohesion. Additionally, classes with a high out-weighted degree should be examined for potential high coupling. In table 7 we present the detected classes.

The detection process starts by identifying all classes with a weighted degree centrality exceeding the Average Weighted Degree (AWD). Subsequently, three filters are applied to this subset of classes based on the thresholds identified previously. These filters serve as criteria to further refine the selection of the suspect classes. For instance, in the JUNIT project, which encompasses 1014 classes, we observe that 220 classes are greater than the average. When applying the CC filter, this number is reduced to 57 classes. Similarly, the BC filter decreases it to 55 classes, while the EC filter selects 55 classes. Following the same approach, we identified the relevant classes for the remaining projects, and the results are presented in Table 7.

Table 7. Detected classes

Project	Number of classes	AWD	Classes greater than the AWD	CC classes	BC classes	EC classes
JUnit	1014	19.69	220	57	55	55
Spring	4070	33.66	907	227	227	227
BCEL	585	53.68	122	31	31	32
Guava	946	12.40	189	65	49	48

By examining the filtered classes, we are able to analyze and explore each suspected. By reducing the number of classes to a manageable size, we have identified and isolated the potential candidates for investigation. At this stage, the designer's expertise becomes crucial. The designer can examine the final list and pinpoint the classes that require refactoring. Table 8 displays a sample of the classes selected for review in the Spring project, based on the BC. For each project, we select the most connected classes based on the CC, BC and EC.

Table 8. Spring project Sample of ranked classes.

Class Name	In-degree	Out-degree	Degree	BC
ClassPathBeanDefinitionScanner	114	59	173	0.07617
Component	489	6	495	0.199
ConfigurationClassPostProcessor	129	63	192	0.023
ResolvableType	1738	262	2000	0.328
AnnotationConfigApplicationContext	979	37	1016	0.183
InjectionMetadata	70	23	93	0.010
AutowiredAnnotationBeanPostProcessor	70	102	172	0.028
PersistenceAnnotationBeanPostProcessor	57	65	122	0.012
LocalContainerEntityManagerFactoryBean	45	26	71	0.028
JpaTransactionManager	55	56	111	0.032
TypeDescriptor	1232	118	1350	0.217

Given the Spring project and the extracted classes mentioned in table 8 it is evident that classes such as Component, ResolvableType, and AnnotationConfigApplicationContext exhibit a significant level of coupling. Therefore, these classes should be further investigated. For instance, if we analyze the class Component, it provides service or data for 489 classes and it access data or services only from 6 classes. As shown in Table 9, the Component class provides a maximum of 46 services or data to the class MergedAnnotationsTests. Similarly, other classes in the ranked list exhibit comparable behavior. However, the Component class accesses itself only twice. This indicates that the class is highly coupled but lacks cohesion. In fact in Spring project, the Component class is used to mark a class as a managed component. Spring will create an instance of this class and manage its lifecycle. The Component annotation is a stereotype, which means it is used to identify a class as a particular type of concern. Several others classes in Spring provides other stereotype annotations such as Service, Repository, and Controller, each of which is used to identify different types of beans within a Spring application. This class is intentionally highly coupled and it don't need a refactoring.

Table 9. Component class interaction simple

Source	**Target**	**Weight**
MergedAnnotationsTests	Component	46
AnnotationUtilsTests	Component	35
TestContextAnnotationUtilsTests	Component	35
AnnotationDrivenEventListenerTests	Component	20
AnnotatedElementUtilsTests	Component	15
AnnotationMetadataTests	Component	14
…		
Component	Component	2
…		

Table 10 presents a simplified view of the ResolvableType class connections, showing that it provides services to the ResolvableTypeTests class 443 times. Interestingly, it also serves itself 234 times. Examining all the tables, we can deduce that this class, with a high degree of self-service, appears to be well-balanced and exhibits an acceptable level of coupling and cohesion.

Class AnnotationConfigApplicationContext, presents a degree of 1016 but it accesses its own data and services in only 9 times. This kind of classes present a high coupling and a low cohesion and needs to be refactored, since it encapsulates too many responsibilities.

Table 10. ResolvableType class connections simple

Source	**Target**	**Weight**
ResolvableTypeTests	ResolvableType	443
ResolvableType	ResolvableType	234
ConstructorResolver	ResolvableType	52
…		

Table 11 presents the optimized number of classes that a designer should focus on. Through this analysis, the number of classes requiring scrutiny has been considerably reduced. This reduction enhances the investigation process, saving valuable time and effort while improving the efficiency of assessing coupling and cohesion within the system. We identified the most central and influential classes in the four projects. By focusing on these key classes, developers can quickly identify the areas that are most critical for understanding the system's architecture and behavior.

Table 11 Suspect classes.

	JUnit			**Spring**			**BCEL**			**Guava**		
Metrics	**CC**	**BC**	**EC**	**CC**	**BC**	**EC**	**CC**	**BC**	**EC**	**CC**	**BC**	**EC**
Refined classes	**17**	**19**	**22**	**63**	**103**	**133**	**19**	**20**	**12**	**5**	**10**	**15**
Initial classes	1014			4070			585			946		
Max-Union	58			299			51			30		
Max-Intersection	22			133			20			15		
Percentage of detected class Max-Union	0.057			0.073			0.087			0.031		
Percentage of detected class Max-Intersection	0.021			0.032			0.034			0.015		
Average detection	0.039			0.0525			0.0605			0.023		

We apply three filters to extract the highly coupled and lowly cohesive classes. As shown in Table 11, for each metric, we compile a set of classes. The "Max-Union" approach signifies the maximum number of classes detected, which happens when the classes in each set are entirely distinct from one another. Conversely, "Max-Intersection" provides the minimum number of detected classes; this occurs when every set is contained within the others.

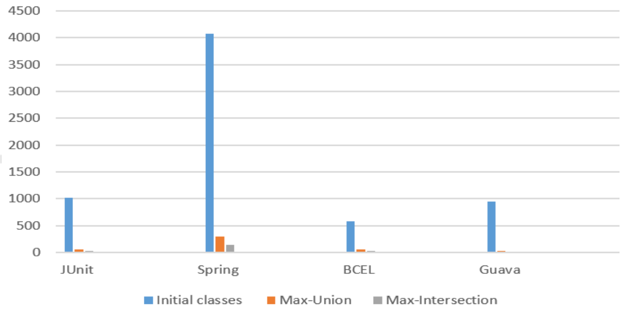


Fig 9. Comparative number of classes

As illustrated in Figure 9, we present a comparative number of classes. The application of SNA has consistently reduced the number of classes across all projects, regardless of their size. On average, the detection rate is below 0.07, significantly refining the number of classes that require investigation. As example, for the Guava project, we identified 16 classes across the three metrics, resulting in a detection rate of 0.016.

8. CONCLUSION

Coupling and cohesion are two fundamental concepts that contribute significantly to the quality and maintainability of software systems. Achieving high cohesion and low coupling is a key goal in OOP design. It leads to more modular, maintainable, and understandable code, which in turn reduces the cost of maintenance and increases the overall quality of the software system. Detecting coupling and cohesion is not just a technical task; it also requires a good understanding of the domain and the problem. Indeed, many studies, often rely on metrics and predefined thresholds to identify classes with high coupling and low cohesion. This approach involves calculating various metrics that measure the interdependencies between classes. Metrics alone may not always accurately capture the complexity of software systems, and thresholds can be subjective. Additionally, the choice of metrics and thresholds can significantly influence the results of a study. This paper introduced a new approach for the detection of classes that are highly coupled presenting a low cohesion. Our approach is based on SNA. Throw the analysis of CC, BC, EC and degree centrality we detect the list of classes that should be investigated. This reduction in the number of classes to be examined results in time and effort savings, ultimately lowering the cost of maintenance for large projects. As future work we intend to apply the same technique to detect design defects.

9. REFERENCES

[1] Z. Junlong and L. Yu, "Degree Centrality, Betweenness Centrality, and Closeness Centrality in Social Network," in 2nd International Conference on Modelling, Simulation and Applied Mathematics (MSAM2017), Bangkok, Thailand, March 2017.

[2] U. Brandes, " A Faster Algorithm for Betweenness Centrality," Journal of Mathematical Sociology , pp. 25(2):163-177, 2001.

[3] T. LewowskiLech and M. Madeyski, "How far are we from reproducible research on code smell detection? A systematic literature review.," Information and Software Technology, vol. 144, no. 3, p. 106783, 2022.

[4] E. Fernandes, J. Oliveira, G. Vale, T. Paiva and E. Figueiredo, "A Review-based Comparative Study of Bad Smell Detection Tools," in 20th International Conference on Evaluation and Assessment in Software Engineering (EASE), Ireland, June 2006.

[5] D. Di Nucci, F. Palomba, D. Tamburri, A. Serebrenik and A. De Lucia, "Detecting Code Smells using Machine Learning Techniques: Are We There Yet?," in 25th IEEE International Conference on Software Analysis, Evolution, and Reengineering, Italy, March 2018.

[6] Z. Wei, Z. Mingyang, Y. Ling and F. Fengchun, "Social network analysis and public policy: what's new?," Journal of Asian Public Policy , vol. 16, no. 2, pp. 115-145 , 2021.

[7] P. B. Stephen, E. Martin G, J. Jeffrey C and A. Filip, Analyzing Social Networks Third Edition, SAGE Publications Ltd;, February 26, 2024.

[8] B. Anuja and M. P. S, "Visualization and Interpretation of Gephi and Tableau: A Comparative Study," in International Conference on Advances in Electrical and Computer Technologies, Coimbatore, India, February 2021.

[9] "A New Metric for Class Cohesion for Object," The International Arab Journal of Information Technology, vol. 3, no. 17, May 2020.

[10] Y. Afrah and H. Mustafa, "An Approach to Automatically Measure and Visualize Class Cohesion in Object-Oriented Systems," in International Conference on Decision Aid Sciences and Application (DASA), Sakheer, Bahrain, 2020.

[11] M. K. Bhatia, "A Survey of Static and Dynamic Metrics Tools for Object Oriented Environment," Emerging Research in Computing, Information, Communication and Applications, vol. 790, pp. 521-530, 2021.

[12] M. Lanza and R. Marinescu, "Object-oriented metrics in practice," Springer, Heidelberg, 2006.

[13] A. Amjad and M. Alshayeb, "A metrics suite for UML model stability," Softw Syst Model, December 2016.

[14] M. Zhang, T. Hall and N. Baddoo, "Code Bad Smells: a review of current knowledge," Journal of Software Maintenance and Evolution: Research and Practice, vol. 23, no. 3, p. 179–202, October 2010.

[15] B. Boczar, M. Pytka and L. Madeyski, "Which Static Code Metrics Can Help to Predict Test Case Effectiveness? New Metrics and Their Empirical Evaluation on Projects Assessed for Industrial Relevance," Developments in Information & Knowledge Management for Business Applications, vol. 3, p. 201–215, 2022.

[16] M. Maddeh, Ayouni, Sarra, S. Alyahya and F. Hajjej, "Decision tree-based Design Defects Detection," IEEE Access, vol. 9, pp. 71606-71614, 2021.

[17] S. Badri and M. Moudache, "Using Metrics for Risk Prediction in Object-Oriented," Journal of Software, vol. 17, no. 1, pp. 1-20, 2022.

[18] K. Erni and C. Lewerentz, "Applying design metrics to object-oriented frameworks," IEEE METRICS, p. 64–74, 1996.

[19] B. Kitchenham, Software metrics: Measurement for software process improvement, NCC Blackwell Publishers, 1996.

[20] M. Mohamed, A.-O. Shaha, A. Sultan and H. S. A. Fahima, "A comprehensive MCDM-based approach for object-oriented metrics selection problems," Applied Sciences, vol. 13, no. 6, p. 3411, 2023.

[21] T. Saurabh and R. Santosh, "Coupling and Cohesion Metrics for Object-Oriented Software: A Systematic Mapping Study," in 11th Innovations in Software Engineering Conference, India, 09 February 2018.

[22] H. Martin and M. Behzad, "Measuring coupling and cohesion in object-oriented systems," in Int. Symposium on Applied Corporate Computing, Monterrey, Mexico, Oct. 25-27, 1995.

[23] I. M, K. S. A. Arvind, S. J. A. Bader and A. Alharbi, "A Component Selection Framework of Cohesion and Coupling Metrics," Computer Systems Science & Engineering, vol. 44, no. 1, p. 351–365, January 2022.

[24] I. G. Mazen and A. Gary, "Quality Metrics measurement for Hybrid Systems (Aspect Oriented Programming – Object Oriented Programming)," Sustainable Future and Technology Development , vol. 3, 2021.

[25] B. Sarika and P. Rashmi, "Cohesion Measure for Restructuring," in Information and Communication Technology for Intelligent Systems, Ahmedabad, India, october 2020.

[26] M. Misbhauddin and M. Alshayeb, "UML model refactoring: a systematic literature review," Empirical Software Engineering, vol. 20, no. 1, pp. 206-251, 2013.

[27] S. Freire, A. Passos, M. Mendonça, C. Sant'Anna and R. O. Spínola, "On the Influence of UML Class Diagrams Refactoring on Code Debt: A Family of Replicated Empirical Studies," in Euromicro Conference on Software Engineering and Advanced Applications, 2020.

[28] R. Malveau, W. J. Brown, H. McCormick and T. Mowbray, AntiPatterns : Refactoring Software, Architecture and Projects in Crisis, John Wiley & Sons, 1998.

[29] A. Dmitry, I. Maqsudjon, K. Artem, S. Anton and Z. Sergey, "Validating New Method for Measuring Cohesion in Object-Oriented Projects," Procedia Computer Science, vol. 192, pp. 4865-4876, 2021.