Components Reusability Optimization based on Re-Structure Monolithic Code

Zeyd Saeed[1,*], Mustafa Ismael Khudair[2], Ahmed Khader Ali Ibrahim[3], Rahman Nahi Abid[4]

[1]Software Department, College of Information Technology, University of Babylon, Babylon 51001, Iraq

[2]College of Media, Al-Iraqia University, Baghdad, Iraq

[3]University Presidency, Department of Internal Affairs, University of Iraqia, Baghdad, Iraq

[4]College of Nursing, University of Babylon, Babylon 51001, Iraq

Emails: zeydsr.sw.phd@student.uobabylon.edu.iq; mustafa.i.khudhair@aliraqia.edu.iq; ahmed.kh.ali@aliraqia.edu.iq; rahmannahi@uobabylon.edu.iq

Abstract

In modern software engineering, monolithic code structures are increasingly incompatible with the flexibility demanded by today's platforms. These tightly coupled systems pose challenges for scalability, integration, and secure deployment. This paper presents a method for restructuring monolithic Java classes into optimized, reusable software components. We analyze each class using 19 object-oriented metrics from the CKJM suite, evaluating cohesion and coupling properties. Using our proposed framework—Good Global Optimization Dynamic Weighted Metrics (GGODWM)—we cluster interrelated classes and transform them into high-level components suitable for microservice environments. These components are evaluated within a Component Base Redesign Structure (CBRS) environment to measure reusability. Our experimental results show a 52% improvement in cohesion and coupling balance, outperforming traditional Turbo_MQ-based metrics. By enhancing component modularity and reducing interdependencies, the proposed approach contributes to more secure and maintainable code, thus supporting cybersecurity goals such as reduced attack surface and easier vulnerability management.

Keyword: Software engineering; Metrics; Object-oriented programming (OOP)

1. Introduction

Re-architecture code is accomplished in two levels, restructuring (OOP) classes yielded at a high level to produce clusters with tightly correlated relationships between classes constructed reusability components. A low-level re-architecture code is achieved in redesign code using design patterns and design principles [1]. Restructuring composite source code classes in OOP into an optimized component structure greatly aids in enabling component software to execute with the qualified-to-be component's reusability. The significance of using component reusability is highlighted. A realistic study was conducted indicating that the US Department of Defense submitted a report indicating that $300 million could be saved annually by relying on mechanisms and techniques for restructuring programs to be reusable programs [2]. Over the past years, code reusability approaches have witnessed many evolution phases adopted by practices and developers. McIlroy proposes concepts related to component reusability techniques that were presented at an international conference in Germany [3]. Another study by Frakes and Succih says: that a close relationship arises between software reusability and an opposing increase in software deployment productivity. The study explains the vision that components' reusability gains permanent enhancement in the entire software qualifier [4].

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

A. Concepts in Software Development

Application restructure is a software engineering requirement that seeks to develop programs compatible with modern computer systems and platforms. Redesigning the source code structure while maintaining the functional behavior of the classes, making easy changes, and achieving the best data flow direction within the improved structure. This makes the optimized application good for maintenance, debugging, and ease of delivery and integration. Many practical activities in software engineering are performed in OOP such as restructuring, re-engineering, and refactoring, listed in the following:

1. **Software Redesign**: A redesign is a modified source code and generates equivalent optimized code. The purpose is to enhance the program's performance and reduce code complexity. Modifying structure qualifies code to be good usability, easier to maintainable, and easy to understand, as well as having more scalability, merging, splitting, birth, and dead [5].
2. **Refactoring:** Refactoring is an approach adopted to modify code structure into a structure with an improved design that has new internal behavior functions without changing external behavior or altering business functions [6]. Refactoring is achieved by omitting repetition, more modular code, compatible and reusable, and reducing code complexity.
3. **Reengineering**: Reengineering is a mechanism that cares about studying and analyzing software systems and generating new code after rebuilding code with advanced design [7]. Rebuilding code seeks to improve existing code while retaining the system's current functionality. Its advantages include improving performance, reducing complexity, and adaptability to changes.
4. **Redesigning User Interfaces:** The user interface is an access point that makes code interact with the outer environment. Redesigning the interface gives a software look and is visually appropriate making the system easy to use by the clients and attractive visually [8].

B. Metrics intended for a redesign of object-oriented systems

Many researchers and practitioners used various metrics in purpose to measure software quality. Much literature is related to many metrics types such as reuse metrics, production, robustness, reliability, and many other measurements related to object-oriented programming software, Onyango et al. [9].

1. There are various metrics in many domains, but these metrics are dedicated to accessing interior instruction class code, for example, McCabe and MOOD. Furthermore, these metrics have restricted access to only interior instruction and are not designed to measure the black box component.
2. A component, in general, consists of several elements, which may be constituted from one or many object-oriented programming elements, classes/interfaces. These elements are organized simultaneously in a uniform working area system. The system has faced difficult problems against scalability features in the CBRS environment.
3. After reviewing the literature study, we saw that most metrics had been adopted to work in OOP environments. System analysis and extracting the instructions features, then calculating many external qualification degrees, such as complexity, semantic properties, reliability, and robustness, do not measure the degree of component reusability.
4. A component is a black box and isolated unit that has limited interaction with outer environments via API and, thus, is different from a regular class. Traditional metrics do not have any ability to analyze, process, and measure the degree of component qualification for component stand-alone or set of components correlated together.
5. In the software engineering field, researchers and developers suggested various metrics. The important one is concentrated on component reusability which relies on efficiency and performance. Turbo_MQ metric is a metric that provides an optimal balance between coupling and cohesion code attributes was proposed by Mitchell and Mankorides [10].

This paper is structured as follows:

The Introduction section outlines the motivation, background, and key challenges addressed in this study. The Related Works section reviews the existing literature on component reusability, software metrics, and the principles behind component-based software design. The Methodology section presents the proposed GGODWM approach, including the definitions of key metrics, the steps involved in analyzing monolithic code, and the process of restructuring it into reusable components. The Results section reports the experimental setup and empirical findings based on real-world software systems. The Discussion section offers an in-depth analysis of the results, validates the GGODWM framework, and

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

compares it with existing approaches such as Turbo_MQ. Finally, the Conclusions section summarizes the contributions of this work and outlines directions for future research.

2. Literature Review

The recent decade witnessed great attention in software evolution by developers and researchers, and building modern applications compatible with various platforms based on splitting large applications into many small units with uniform functionality called components. CBRS is one of the important mechanisms that are interested in providing many approaches to constructing entry points after redesigning parts from source code to make it possible to interact with other system units without knowing any details about these parts. The component definition model presented by Zou et al. [11], has been adopted by many practitioners and software experts. Sharma et al. [12], study various programs to detect diverse troubles that cause the system's bad performance, and deduct the necessity of re-architecture code and regrouping into independent elements, API access point construct to elements to become ready to use different attributes, relationships, field access from rest system units. Rathee and Chhabra [13], aggregate details via a study related to systematic multiple programming and deduce that the components generated from identifying many parts with tightly correlated that achieve simultaneously business functions to be able to reusable in different systems, also to be able to substitute any failure components. Components are generated by using many techniques for assigning API interfaces for object-oriented programming [14]. The researchers' conclusions after completing their studies say: that using components with API interface gives more flexibility in organizing, setting, and getting parameters when compared with its counterparts from the rest different programming codes. Rathee and Chhabra [15] infer in their search and reveal that building reusable components capable of being used in multimedia programs and suggest a technique to identify and build reusable multimedia components from old systems and obsolete designs. In this study, a mechanism was proposed based on a fuzzy metric technique, which measures the component based on its attributes, including the size, the source quality, and the degree of its complexity.

Papamichael et al. proposed a new metric used to quantify components' reusability, obtained from automated components and investigation metrics [16]. The authors of this article generate components via a study dataset and then fit a set of class references into related programs, building components based on class and package level, to maximize subjectivity and minimize independence. Component reusability relies on optimizing the architecture features and source code attributes that had been considered based on system development and system deployment. IN [17], coupling and cohesion were estimated to identify strong interrelated attribute classes and weak outer interrelated attribute classes/packages to define complexity structure degree.

Component reusability metric is calculated automatically by monitoring the code dynamically in run time and evaluating cohesion, and coupling suite metrics that offer the behavior of the software system structure [18]. The main aim of adopting dynamic metrics is to improve the services of the CBRS technology.

2.1 Component concepts

A component is a coding unit with a tightly tied architecture that performs a specific task and provides a service that has been defined previously. The most important matter that distinguishes components is communication via the API interface access point, request service from the component, and its return response [19]. The component elements are encapsulated and isolated in a black box. Therefore, making the component able to be replaced with other faulty components. It has a grid that makes components have a plug-and-play feature and generally, component reusability distinguishes. Component reusability techniques greatly assist us in analyzing components into individual unit’s compatible with different platforms, components derived from large monolithic systems into controllable communities toward reducing the system complexity and improving the code reusable. Object-oriented program composite from multiple classes, packages, and sub-packages, which qualified him to constitute a modular program implemented by many program languages, the most important language of which is Java. Program language terms (class, package, sub-package, component, and module) have fixed meanings in run time execution. A class is a blueprint representing an entity with a specific task in correlated attributes and tied signature methods. In other terminology, components are a community designed from multiple related classes constructed in confirmed architecture. API interface represents a skin of the units of the components that make the external system units able to interact with them. In summary, it can offer the main differences between classes, packages, and components and the most significant for each one:

1. A class is a segment that represents smaller units of reusable code. Many classes may be connected with additional classes to integrate service deployment. A component does not have to connect with another component, with no significant external dependencies, hence the component is more reusable.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

2. One of the most important principles in OOP is an inheritance that mainly contributes to class design to be class reusability, while component, the aggregations is the principle that considers the main contributor in component design to be component reusability.
3. The class is invocated by calling it from other many classes in a way, while the component is communicated with other components by an API component interface which state accesses points.

The components are constructed from many independent software modules from scratch. The components program is set up hierarchically from the main components and down toward other partial components. A component structure can be represented in a set of levels scheme, the top-level form of a $C^i =< E^i, R^i >$ pair, where E^i denotes to offer a set of the elements component that belong to component i^{th}. The component includes various class types and class interfaces with signature functions. As for R^i, it symbolizes many interrelationships that link end node classes and intermediate middle nodes at a level i^{th}. The component in a level $(i-1)^{th}$ can request service from the component at level $(i+1)^{th}$, and the component in level $(i+1)^{th}$ may return a response in that delivers service into the component at level $(i-1)^{th}$. Component elements connected with relationships can be defined in a formula R^i is $R^i \subseteq E^i \times (E^{i+1} \cup E^{i-1})$. This relationship is snipped from one of the binary edges in two endpoints and represents the nature of bidirectional communication to the components of the level i^{th} with the rest of the components in $(i+1)^{th}$ or $(i-1)^{th}$.

- In a component concept-based system, class interfaces are the most major section of a component system environment. The components share various resources, and components interact with other components via an interface comprising many inner interconnection functions. Component interface deals with several mechanisms such as operations, methods, and functions which interact with the outer environment in a way that receives the request from any component, processes this request, and returns the response [17]. Source code interacts with outer requests by interfaces which is followed by sequential methods. The first method depicts the description class interfaces, one of the significant interfaces is an Interface Description Language (IDL) Guerrero-Garcia et al.[12].

- The second method is the language- Component Interface Design based methodology; it uses a predefined programming language design pattern. These patterns are used to demonstrate a set of characteristics and attributes. Much literature research that supports many components properties and makeup interfaces [14], [19], [20]. These characteristics could be adopted in different components by illustrating different mechanisms. This article provides an interface in a systematic definition, to accurately determine the appropriate metrics. The first level of component is an interface which is defined as a couple of interface $(C^i) =< provides^i, requires^j >$. This equation depicts that the $provides^i$ has several features and attributes defined as the first level of the component, and it displays many attributes that make the component send services to other external components. The second parameter of the equation is a $requires^j$ with properties requested by different other components environment.

3 Methodology

Components in software engineering are one of the most significant principles that are developed using CBRS technology. The components concept practically gives many principles in the system environment, such as speed, and saving cost, and mainly contributes to improving resources allocated to their systems in terms of adopting these components. Resource allocation in terms of reusing helps us to develop the system and build it from scratch while evolving it. Reusing resources assists developers in enhancing applications from scratch while developing them. Thus, developers have a large common collection of third-party components, and the challenge remains to select the best of these components and the in-progress demand of developers. Intuitively, a set of components often obeys developers' requirements. Thus were have a problem list is how to choose one better from multiple components.

The Good Global Optimization Dynamic Weighted Metrics (GGODWM) in this article significantly mitigate this problem by offering an automated approach of dynamic weighted metrics for calculating the component quality.

In this research, ckjm suite metric is used to evaluate the two concepts of coupling and cohesion for each class in Java programming, then restructuring the intercorrelated classes into an optimized structure design. Building components from classes after restructuring these classes into optimized structures in high-level design and evaluating the classes in high outer cohesion, and inner cohesion with low coupling components. However, the ckjm is showing its metrics in a class-level evaluation. Thus, it has used hybrid metrics from these metrics to calculate the amount of class connection with other classes in the component same, finally measuring the component reusability score. The next section illustrates the ckjm metrics suite in detail, ckjm has various metrics used to evaluate class stand-alone, explain each metric in detail in the following:

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

A. Ckjm metrics

1. WMC: Calculate the weighted methods for each class; this metric is used to measure the complexity method degree. The complexity is a function that depends on the cyclomatic complexity of the methods. This metric carries the value 1 for each method in a class [21].

2. DIT: Depth on Inheritance Tree calculates the depth of the inheritance from the top level of the specific parent class towards hierarchy into the leaves child class. The minimum value of DIT is 1[21].

3. CBO: Coupling between Objects, measuring the interrelationships quantified between a particular class on one side and the number of correlations that connected with targeted classes on the other side. This correlation could appear in many instruction codes such as method call, field access, inheritance, argument, and return type [22].

4. LCOM: Lack of cohesion in methods, this measure calculates the lack of cohesion in procedures in the class, and they are not related, not communicating with the class Characteristics. In general, it carries all pairs of class procedures. Two procedures share a pass to at least one common characteristic of the class, while in the remainder of the pairs, the two procedures do not convey access to another common characteristic. Then, the lack of cohesion is estimated by locating the number of procedures that do not share the characteristic class to the number of procedures that share the characteristic class. In other words, the measure of lack of cohesion gives the number of disjoin graph components for class methods [21].

5. LCOM3: Lack of cohesion in methods.

LCOM3 carries a value between 0 and 2.

m - methods count in a class.

a - variables count (attributes number in class).

μ(A) - number of methods that access attribute. $LCOM3 = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right)}{1-m}$ [23].

1. NPM: Number of Public Methods, total the count methods that have been publicly declared. This metric is important in calculating the API size [16].
2. DAM: Data Access Metric, this metric cared to determine the ratio among the private attribute count that has been declared in the class to the sum of total attributes in the same class. A high metric value indicates to desired class, metric range [0,1] [22].
3. MOA: Measure of Aggregation, estimates the ratio between the whole attributes that are declared in the class into partial specific attributes type in a particular class [24].
4. MOFA: Measure of Functional Abstraction, this metric depends on computing it in the method count that appears inherited from a certain class to the method count used in the same class. the metric range [0,1] [21].
5. CAM: Cohesion Among Methods of Class, this metric calculates the association between methods of a single class established on the methods' parameters. It computes the totality of the diverse parameters for each method divided by multiplying all kinds of diverse parameters for the class and the methods count for that class. (range 0 to 1) [21].
6. CBM: Coupling Between Methods, computing the whole number of procedures for a class to which the leftovers of the methods for other classes are associated and inherited from that class method [22].
7. LOC: Lines of Code, lines of Java bytecode are counted, which is the sum of the number of instructions for each method and the number of fields used for a specific class [16].

B. Component Cohesion (CCoh) Metric

A component is a design that is considered a basic program pattern and has an architecture that would be compatible with different platforms. Components operate to introduce specific functions for a complete program model. The components comprise a set of contents that are sufficiently cohesive, the class and class interface that represents component contents must be made up to share many related tightly methods and functions in the same component. The relationships that link various contents must be strong, and lead to the generation of cohesion component. However, the metrics applied to object-oriented programming are unable to apply directly to the components in evaluating the coupling and cohesion principle. These metrics are designed to measure OOP items classes inside alone. Generally, a component is constructed from a set

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

of many correlated classes. Thus, these metrics do not appropriately work directly with the components. However, many types of relationships associate different component contents. In this article, the associations among many classes are used to evaluate the cohesion and coupling degree of the component. From the good global optimization dynamic weighted metrics (GGODWM) proposed in this article, we discover that the CCoh cohesion dimensions depended on dual kinds of cohesion quantities:

1. ***Interior Cohesion**:* Interior cohesion indicates quantifying the cohesion of the content's elements of a component, for example, class and interface. If we have a component C^i, with content e_j, then measuring the cohesion quantifies that it offers tight cohesion among the many different methods and functions that belong to the contents of the same component in $inCoh(i,j)$. Inner cohesion represents the power of the core relationships that are associated with the methods and the way that the functions interconnected with other method in specific one class. In this paper, the measurement of internal (inner) cohesion is based on four selected CKJM metrics: LCOM, LCOM3, WMC, and CAM. These metrics quantify the degree of member accessibility within the same class or between methods of different classes. Each method is assumed to belong to a distinct class. The remaining CKJM metrics are not suitable for capturing internal class relationships relevant to cohesion, as they primarily assess dependencies across multiple classes rather than within a single class. Therefore, LCOM, LCOM3, WMC, and CAM were incorporated into the Good Global Optimization Dynamic Weighted Metrics (GGODWM) framework to effectively assess internal cohesion. Class cohesion is evaluated based on the number of members accessed within the class—higher cohesion indicates stronger interconnections among methods and members of the same class. The two types of internal cohesion discussed earlier are specifically calculated for the K^{th} class, which contains *m* methods.

Calculate the cohesion class, and express the degree of the bidirectional connection of the methods and function within the class. Via the metric attributes, we take an LCOM3 feature. Therefore, the study values for this parameter display values from 0 to 2, and the value of the metric describes the lack of cohesion of the methods, the less value is the more profitable, signifies high cohesion, and vice versa, because it excludes methods that are not tied to others in the class. Thus, this research focused on using cohesion. Firstly, we must calculate the inverse of the LCOM3, take the LCOM3 metric value, and divide by 2 to make the metric normalized in range 0 and 1. Secondly, subtracting the outcoming from 1 value to give the degree cohesion. The equation following expresses the mathematical formula:

$$1 - \frac{LCOM3}{2} \qquad (1)$$

2. ***Exterior cohesion:*** After merging many classes within a specific component, the interconnected relationships between the classes of the same component are the exterior cohesion, while the interconnection between the different components is the coupling. Exterior cohesion refers to the degree of cohesion of the content with other contents of the same component, for example, a class with a sub-package of the component. For example, content belongs to a component C^i, and $OutCoh(i,j)$ can be calculated for it if it has a relationship with the additional of the subjects feel right to the same component. The exterior cohesion of a module is calculated centered on the correlation ratio between the class and the sub-package within the component. Exterior cohesion can be described using the following equation:

$$OutCoh\ (i,j) = \frac{Sum\_CBO}{Class\_count - Class\_No_{Component_i} + 1} \qquad (2)$$

To calculate the Exterior cohesion, we must find the coupling quantity between component classes, which is an interior cohesion of the component and an outer coupling of the classes in the component. We compute the coupling between objects, which is the sum CBO for all classes in the component. Divided by the class count in the system, minus the number of component classes and adding 1 to it, the number 1 expresses the connection of the intended component with the sub-package and avoids division by zero.

C. Component Coupling (CCup) Metric

Exterior module dependency and its outer connection with other components outside environments are the basic reusability principles. However, a component that could be considered a high cohesion and low coupling would be possible reusability. It is necessary to correlate different components with the intended components that share the same environments. This research estimates the component qualification based on coupling and cohesion and focuses on the interconnection between the specific component with another component that is present in the same workspace due to the communication between their subjects.

The coupling for the C^i component is estimated in dual steps: For example, if we have a component C^i that is interrelated to extra components such as $C^j, C^k, C^l, \ldots, C^t$. In the first step, a component C^i is taken, and a pairwise coupling with

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

another component such as C^j is calculated by determining the relationships between one component and another. In the first step, calculate the coupling among objects (CBO), this metric expresses the tightly intra connection of the correlation between the objects when studying the features that provided by the ckjm metrics, CBO is a metric has features that achieve the purpose of the specific class with a regular interconnection link with other class, that does give formulate connection with classes in the whole component. To determine the metrics in a more, take the metric and divide it by the class count, then take the RFC (Response for a Class) metric in the ckjm tool in which that metric measures the different methods count that can be calculated when running and create an object from the distinct class. This metric is used to estimate the complexity and responsiveness of a class. To compute the influence coupling of all classes, we use the log formula to estimate the power value that gets from the divided RFC over CBO which has divided into class counts. The following equation explains this:

$$COUP_{ij} = \log\left[\frac{RFC}{\frac{CBO}{Class\_Count}}\right] \quad (5)$$

After calculating the coupling and the cohesion from the previous equations 3,5, we estimate the weight of the cohesion relative to the coupling and estimate the weight of coupling relative to cohesion, and the values we seek to be normalized between 0 and 1, and the following equations express this clearly:

$$W_{Cohesion} = \frac{Cohesion}{Cohesion+Coupling} \quad (6)$$

$$W\_Coupling = \frac{Coupling}{Coupling+Cohesion} \quad (7)$$

Finally, calculate the component, where each class is passed by a component, which are values that express the degree of interconnection of the classes with each other as a whole within the component and the strength of the interconnection of the methods as a whole with each other within the class, and the higher value of the component expresses the unity of joint work to accomplish a specific function.

$$Component_i = \frac{W_{Cohesion}*CAM}{W_{Cohesion}*CAM+W_{Coupling}*CBO} \quad (8)$$

D. Degree of Customizability (DC) Metric:

The component was invented to be able reusability adopted by software developers. Client requirements are diverse and vary from one system to another. From a software engineering perspective, a component reusability that has various features will give customization is more likely to be component reusability from than a component with fewer features that gives little or give not field customization. Calculating the degree of customization for a C^i component is based on extracting variable supporter count within the component element, the client and developers could be possible to customize these variable fields before merging many components in the embedded component. The customization of the component is a capability to alternate the field variable supporters based on the customer's request. The two-measurement metrics coupling and cohesion cannot be able to measure customization degree. These metric targets to estimate the power of the autonomy defined by the core classes. In general, extract the features method and characteristics of the class. The whole properties defined in a class could be able to directly measure the component customization. Thus, these metrics rely on the internal contents of the component to more accurately estimate that the customization is dedicated.

we have to calculate the degree of reusability of the component so that it is eligible for deployment and work on it. To calculate the component reusability, we calculate the degree of customizability (DC) for the access field. The degree of customizability (DC) is calculated from the sum of the weights of the method over its method count for the component. The weight method is estimated based on the class metrics calculated from applying the ckjm metric, we take the WMC metric, the number of methods for one class within the component divided by DAM, and DAM is the ratio of the number of private attributes declared in the class to the number of the total attributes defined in the same class, multiplied by MOA, MOA is partial or all relationships that were used to couple this class to the other classes in the component, that is, the number of field accesses that the user used for the variables that are defined in the class. The following mathematical expression explains the process the estimating the weight methods:

$$W_{Method} = \log\left(\frac{WMC}{DAM}\right) * MOA \quad (9)$$

$$Degree\ of\ Customization\ (DC) = \frac{\sum_J^N W\_Method_j}{N} \quad (10)$$

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

E. Component Complexness Interface (CCI) Metric

The component has a vital class called the interface class; they interact with this component that takes place through its interface. The component is designed to be an isolated unit and interact with this component via its interface. Therefore, the component interface mainly determines the degree of the component reusability based on the complexity of this interface. Thus, the component interface complexity has important effects on components in terms of reusability and understandability [25], [26]. This is because the component reusability is designated on the reality of just how efficient the component is to satisfy the needs requested by the designers, and in likeness the capability of the component, that supplies these requests. The component must acquire the ability to be understood. According to Washizaki et al. [27], the author says that understandability is an uncomplicated ness, this concept means that must be required to organize the interface in an optimal form. A component with highly understandable has a high reusability. Therefore, the estimation acquired from the suggested sequence GGODWM, CCI is to quantity the extent of complexity of a component according to the complexness of its interface. The complexness of the component depends mainly on the complexness that arises due to the message transient contrivance, which is handled by the interface. The message transient depends on the total number of information that needs to be passed as well as the type of data, and this data determines the complexity of the MP. Component complexness results from the use of message transients with interfaces. To determine the total complexity, we consider three parameters:

The total parameters that the interface considers to pass.

The data style of the parameters relies upon the interface.

The return used and its category in the interface approaches.

These three considerations directly affect understanding efforts, depending on the nature and type of these parameters. The developer approves these parameters to obtain the basic concepts specific to the component. The complexness of the message transient depends on the return kind of the interface method and the kind of parameter passed by interface k and the Component Complexness Interface (CCI) Complexness was measured using the following equation:

$$W_{Class} = \log\left(\frac{DIT}{MFA}\right) * MFA \qquad (11)$$

The weight of the class [28] is calculated by taking the metric DIT, the feature DIT that expresses the depth degree of inheritance of a class, a class is a parent, and the number depth level of inheritance from the parent class to the children class. Divided by MFA, it is the ratio of the number of methods that have been inherited by classes to the total number of methods that can be accessed by methods members of the same class.

$$Component\ Complixity\ Interface(CCI) = \frac{\sum_n^j W\_Class_j}{N} \qquad (12)$$

F. Component Reusability (CR) Metric

Component reuse is an important aspect of our study. It helps software developers how to choose the component that meets all their requirements and integrate it with other components or separate a component into more than one component. Here we can measure the possibility of reusing the C^i component mathematically according to the twelve equations mentioned previously as follows:

Now, we calculate the $DC\_CCI\_Magnitude$ vector as the following formula:

$$DC\_CCI\_Magnitude_i = \frac{\sqrt{DC^2+CCI^2}}{\sqrt{DC^2}+\sqrt{CCI^2}} \qquad (13)$$

To calculate each component's magnitude against other components in the system from the next equation:

$$Component\_Magnitude_{i,j} = \frac{\sqrt{Component_i{}^2+Component_j{}^2}}{\sqrt{Component_i{}^2}+\sqrt{Component_j{}^2}} \qquad (14)$$

To calculate the reusability for each final component, use the following equation:

$$Component\_Reusability_i = \frac{\sqrt{DC\_CCI\_Magnitude^2+Component\_Magnitude^2}}{\sqrt{W\_Coh^2}+\sqrt{W\_Coup^2}+\sqrt{W\_Method^2}+\sqrt{W\_Class^2}} \qquad (15)$$

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

GGODWM Description and Methodology

This section will represent many steps that illustrate the aim of this research, analyze a monolithic program and extract many tightly related classes, and then redesign these structure classes, in five steps:

Step 1: Code collection

In this step is the preparation phase to gather the data set that is depicted as a monolithic programming. A monolithic program that was instituted in an OOP, and constructed from many classes. Each program will have been compiled separately. The program was compiled to optimize the individual source code classes. A set of optimizations passes into each class to improve the version of an original class. This step's outcomes are classes. Class that represents the source code in the form of bytecode jvm, which had been generated from, compile classes. The following figure shows a monolithic program comprised of multiple classes, and Classe. Class after compiler applying.

```
class Client{
    public static void main(String[] args) {
        Service service = new Service();
        service.doSomething();
    }
}
 class Service {
    public void doSomething() {
        Dependency dependency = new Dependency();
        dependency.doWork();
    }
}
class Dependency {
    public void doWork() {
        System.out.println("Doing from dependency...");
    }
}
```

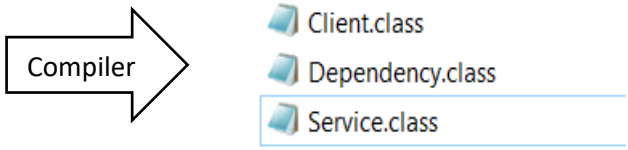

Figure 1. Java Source Code and Classes. Class after compiler.

Apply the Ckjm in the second step, which is a metric that applies to each class to extract 19 features for each class, to illustrate this step use the following instructions:

java -jar ckjm_ext.jar *.class

The outputs from applying this instruction appear on the Terminal outputs as follows:

org.apache.commons.lang3.builder.AbstractSupplier 2 1 0 1 3 1 0 1 1 2.0000 7 0.0000 0 0.0000 1.0000 0 0 2.5000

~ protected org.apache.commons.lang3.builder.AbstractSupplier asThis(): 1

~ public void <init>(): 1

org.apache.commons.lang3.builder.ToStringStyle$NoClassNameToStringStyle 2 2 0 1 5 1 1 1 0 2.0000 14 1.0000 0 0.9912 1.0000 0 0 5.5000

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

~ private Object readResolve(): 1

~ void <init>(): 1

org.apache.commons.lang3.builder.IDKey 3 1 0 2 5 0 2 0 2 0.2500 42 1.0000 0 0.0000 0.8333 1 1 12.3333

~ public int hashCode(): 1

~ void <init>(Object value): 1

~ public boolean equals(Object other): 4

The command's output will be a list of class names (prefixed by the package they are defined in), followed by the corresponding metrics for that class: WMC, DIT, NOC, CBO, RFC, LCOM, Ca, Ce, NPM, LCOM3, LOC, DAM, MOA, MFA, CAM, IC, CBM, and AMC, each metric represents a feature for that class. In lines where at the beginning is "~" is the value of CC. CC is counted for all methods in a given class. After that, we convert the Terminal outputs to a data sheet to build a data set for analyzing and processing it. The following is an example of an Excel sheet. It contains a set of sheets. Each sheet symbolizes a program. The sheet contains several classes belonging to the program, and the special data for each class, which are 19 features accompanying it as shown in figure 2.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	CC	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	Ce	NPM	LCOM3	LOC	DAM	MOA	MFA	CAM	IC	CBM	AMC
2	org.apache.commons.lang3.builder.ToStringSummary	0	1	0	0	0.1	0	0	0	0	2	0	0	0	0	0.1	0	0	0
3																			
4	org.apache.commons.lang3.builder.AbstractSupplier	2	1	0	1	3	1	0	1	1	2	7	0	0	0	1	0	0	2.5
7																			
8	org.apache.commons.lang3.builder.ToStringStyle$NoClassN	2	2	0	1	5	1	1	1	0	2	14	1	0	0.9912	1	0	0	5.5
11																			
12	org.apache.commons.lang3.builder.IDKey	3	1	0	2	5	0	2	0	2	0.25	42	1	0	0	0.8333	1	1	12.3333
16																			
17	org.apache.commons.lang3.builder.HashCodeBuilder	39	1	0	9	72	267	0	9	32	0.6789	826	1	0	0	0.0835	1	1	20.0513
57																			
58	org.apache.commons.lang3.builder.ToStringStyle$JsonToSt	20	2	0	2	76	190	1	2	10	1.0526	525	1	0	0.855	0.2294	1	5	25.15
79																			
80	org.apache.commons.lang3.builder.EqualsExclude	0	1	0	0	0.1	0	0	0	0	2	0	0	0	0	0.1	0	0	0
81																			
82	org.apache.commons.lang3.builder.MultilineRecursiveToSt	14	3	0	4	43	0	0	4	2	0.7436	400	1	0	0.8984	0.2449	3	5	27.3571
97																			
98	org.apache.commons.lang3.builder.DiffExclude	0	1	0	0	0.1	0	0	0	0	2	0	0	0	0	0.1	0	0	0
99																			
100	org.apache.commons.lang3.builder.DiffBuilder$17	5	3	0	2	5	6	1	2	4	0.8125	36	0.25	1	0.8095	0.4	2	6	5.4
106																			
107	org.apache.commons.lang3.builder.Diff	5	2	18	23	15	2	20	3	4	0.8333	60	1	0	0.7647	0.4667	0	0	10.4
113																			
114	org.apache.commons.lang3.builder.DiffBuilder$18	5	3	0	3	5	6	1	3	4	0.8125	36	0.25	1	0.8095	0.4	2	6	5.4
120																			
121	org.apache.commons.lang3.builder.DiffBuilder$13	5	3	0	2	5	6	1	2	4	0.8125	36	0.25	1	0.8095	0.4	2	6	5.4
127																			
128	org.apache.commons.lang3.builder.DiffBuilder$14	5	3	0	3	5	6	1	3	4	0.8125	36	0.25	1	0.8095	0.4	2	6	5.4
134																			
135	org.apache.commons.lang3.builder.DiffBuilder$15	3	3	0	2	4	0	1	2	2	0.875	26	0.25	1	0.8947	0.5	2	6	6.3333
139																			
140	org.apache.commons.lang3.builder.DiffBuilder$16	5	3	0	2	4	6	1	2	4	0.8125	34	0.25	1	0.8095	0.4	2	6	5
146																			
147	org.apache.commons.lang3.builder.ToStringBuilder	65	1	1	5	97	0	2	4	64	0.5547	657	1	2	0	0.1101	0	0	9.0462
213																			
214	org.apache.commons.lang3.builder.DiffBuilder$10	5	3	0	3	5	6	1	3	4	0.8125	36	0.25	1	0.8095	0.4	2	6	5.4
220																			
221	org.apache.commons.lang3.builder.DiffBuilder$11	5	3	0	2	5	6	1	2	4	0.8125	36	0.25	1	0.8095	0.4	2	6	5.4
227																			
228	org.apache.commons.lang3.builder.DiffResult	10	1	0	7	25	11	3	4	8	0.7963	135	0.8333	1	0	0.25	0	0	11.9

Figure 2. Data Sheet Programs with Features.

The following flowchart explains the five stages see figure 3, which are compiling the program, applying the ckjm metrics to the classes, and building the data set features classes.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

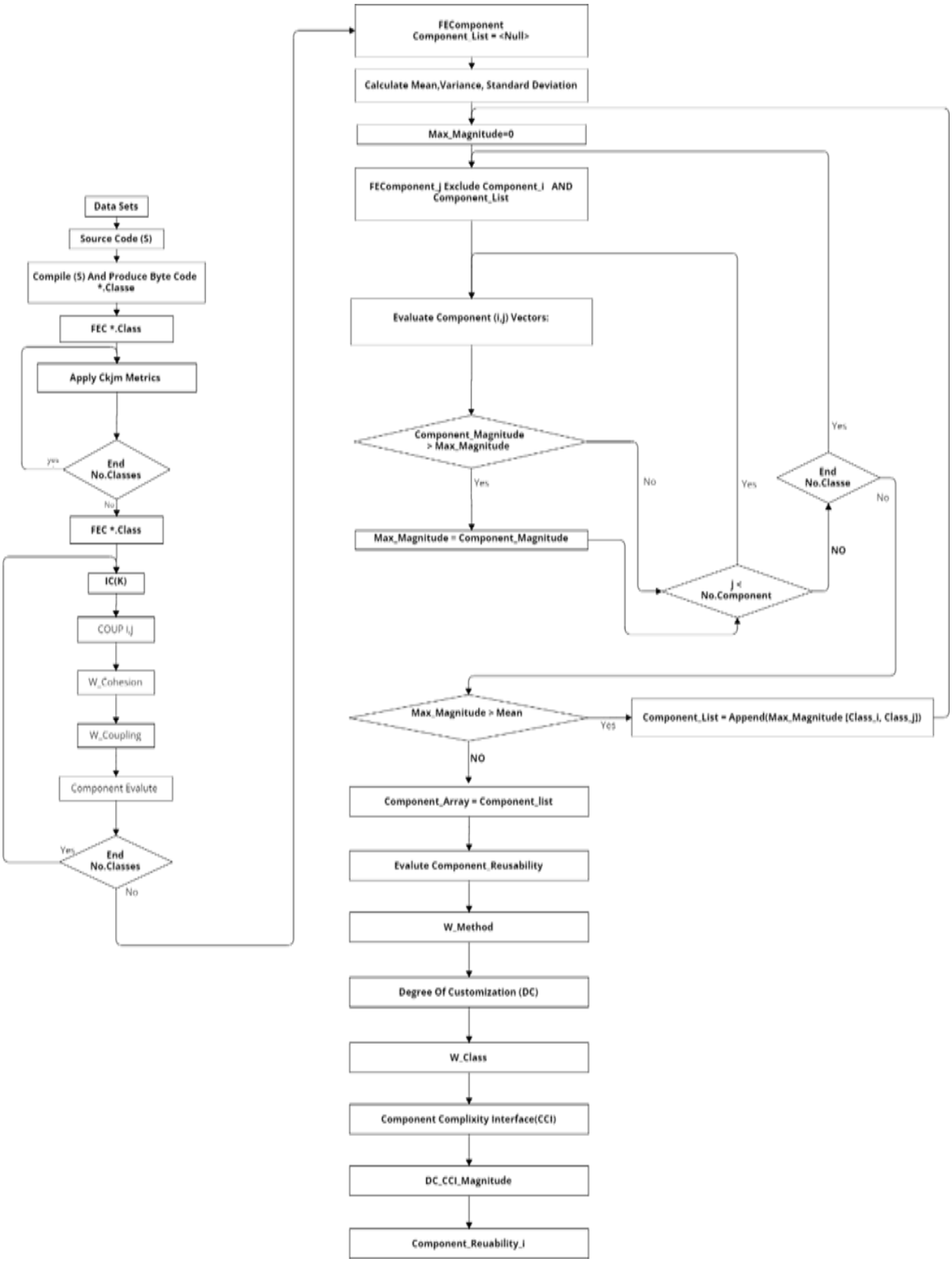

Figure 3. Good Global Optimization Dynamic Weight Metrics Proposed.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

Second step:

In this step we calculating the second version of cohesion is an LCOM metric, this metric illustrates the second feature offers value defining the lack of cohesion for the class, but the value has an unlimited number, thus the inability to normalize the value in range [o,1]. Studying this metric and the equations that derive from them and the approaches adopted to evaluate the lack of cohesion of methods for the class. We state that the computations rely on the number of couples of methods that do not communicate attributes, and the likelihood of the number of these couples is calculated based on a mathematical model expressing the permutations, by computing the factorial of the methods number divided by factorial 2, which sound the pair of linkages between two methods, multiplied by the factorial methods number subtracting 2 of them [21]. To normalize the metric value in the range [0,1], use the following equation to produce cohesion degree.

$$1 - \frac{LCOM}{WMC!/2!(WMC-2)!} \qquad (2)$$

Then take two values from the prior two equations: equation 1,2 and estimate the average cohesion for the class. The conclusive form of the equation becomes:

$$IC(k) = \log\left[\frac{CAM}{\frac{1-\frac{LCOM3}{2} + 1-\frac{LCOM}{WMC!/2!(WMC-2)!}}{2}}\right] \qquad (3)$$

From the above equation 3, we use the CAM metric (cohesion among methods). The CAM is a metric that expresses the strength of the internal correlation between all the methods of the class. An expression of the power of cohesion is calculated by dividing the CAM metric by the average cohesion from the first feature calculated previously, and the result is a value that reveals the class cohesion.

Third step:

In this step we estimating the dynamic cohesion and coupling for each monolithic class individually, and generate distinct components for each class, this step will generate a number of components equally according to the number of monolithic program classes. The important thing that distinguishes this article is computing dynamic cohesion and dynamic coupling, which means that it allocates dynamic value for each class according to some of the features that are associated with the ckjm metrics. In the literature study, cohesion, and coupling metric weight fixed dedicated depend on software experts and software developers, the cohesion weight value is often 40 percent and 60 percent for coupling weight value. The proposed approach Good Global Optimization Dynamic Weighted Metrics (GGODWM) technique is adopted to estimate the cohesion and coupling dynamic weight of the component that is generated from the second step. Many principles that are adopted to calculate the dynamic coupling and cohesion, and to achieve this, perform the following operations:

- We calculate inner class cohesion, which expresses the power of the connection of methods and functions within a class by using equation 1.
- We calculate the second version of the cohesion LCOM metric, equation 2.
- Then we take two values from the previous two equations and estimate the average cohesion for the class. Apply equation 3.
- We will calculate the coupling between the classes, using equation 5.
- we estimate the weight of the cohesion relative to the coupling and estimate the weight of coupling relative to cohesion, and the values we seek to be normalized between 0 and 1, equations 6 and 7 are used for this.

The last operation in this step, component evaluation is an important step, where each component considers construct from one class, expresses the interconnect elements class with other class elements, and interconnect class method inside the class, estimates this interconnection, extracts the higher value which reflects the unity of joint function accomplish in the component. Apply equation 8.

Fourth step:

The many tightly related classes are combined into one component that has high cohesion and low coupling as a cooperative function model. Collecting a subset of classes from a set of classes is a sub of the sum problem. The aim is to generate an optimized design that appears from the aggregation of classes under the best counsel, and for each class, figure out the shortest path to the neighbor class and the strongest connection with it, and so on with the rest to build a robust and improved class network. In our research, we calculated the vector from the component values, and as pictured in the coordinate axes, it falls in one of the quadrants depending on whether the cohesion and coupling values are negative or positive, this appears in figure 4.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

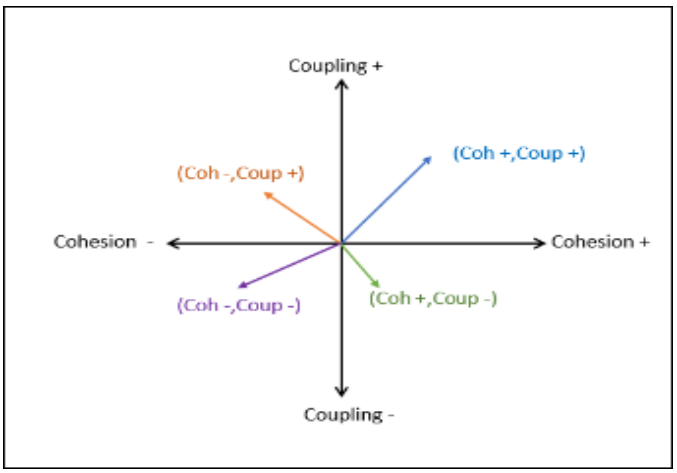

Figure 4. Cohesion and Coupling Vectors based on it Values.

After that, we compare each component with the rest of the components, except for the component itself and the component list that comprises the previously sketched classes. The comparison is done by calculating the magnitude vector for each two components and recording the value of the largest component that expresses the highest correlation between the two classes compared within the variance and standard division. Iterate for all components until all components about the variance are verified. Component magnitude done by using equation 14.

After completing this step, the output is an array containing the classes for the best component.

The sequence of these operations can be described through the following results fig 5,6,7,8,9, which show the action mechanism of our method by finding the best tie by the most powerful coupling and cohesion between two classes in iteration 1, In this iteration, two classes are merged in the component block, and the other iteration, we calculate the cohesion and the coupling for the resulting component by finding the coupling between component classes, which is considered to be the exterior cohesion, and the coupling of the compound with the rest of the classes and the best magnitude component is selected, merge that class with the component, in the end, the algorithm stops when the standard division condition is met, all the classes of the compound under the Variance range.

0	0.733614	0.797012	0.950178	0.734211	0.725044	0.759658	0.734211
0.733614	0	0.861653	0.970767	0.707111	0.7081	0.713095	0.707111
0	0.733614	0.797012	0.950178	0.734211	0.725044	0.759658	0.734211
0.733614	0	0.861653	0.970767	0.707111	0.7081	0.713095	0.707111
0.797012	0.861653	0	0.866415	0.862362	0.850352	0.887258	0.862362
0.950178	0.970767	0.866415	0	0.970957	0.967657	0.977276	0.970957
0.734211	0.707111	0.862362	0.970957	0	0.708232	0.712792	0
0.725044	0.7081	0.850352	0.967657	0.708232	0	0.718746	0.708232
0.759658	0.713095	0.887258	0.977276	0.712792	0.718746	0	0.712792
0.734211	0.707111	0.862362	0.970957	0	0.708232	0.712792	0

Figure 5. Component Magnitude for each Component with rest Components.

Coh	Coup	W_coh	W_Coup	Component_1
-0.230139791	1.735997884	-0.152829667	1.152829667	-0.94469226

Figure 6. Evaluate Component Metrics.

Component_0		
-0.053966221	0.947503528	
-0.030595804	0.969136832	
-0.053966221	0.947503528	
-0.030595804	0.969136832	
0.170918592	0.860541553	
-0.030387648	0.969336832	
-0.034022355	0.965863482	
-0.030387648	0.969336832	
	Iteration_2 =	
	maxMagnitude =	0.969336832
	Component_2 =	[org.apache.commons.lang3.mutable.package-info, org.apache.commons.lang3.mutable.MutableObject, org.apache.commons.lang3.mutable.MutableShort]

Figure 7. Merge_1 Class within Component.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

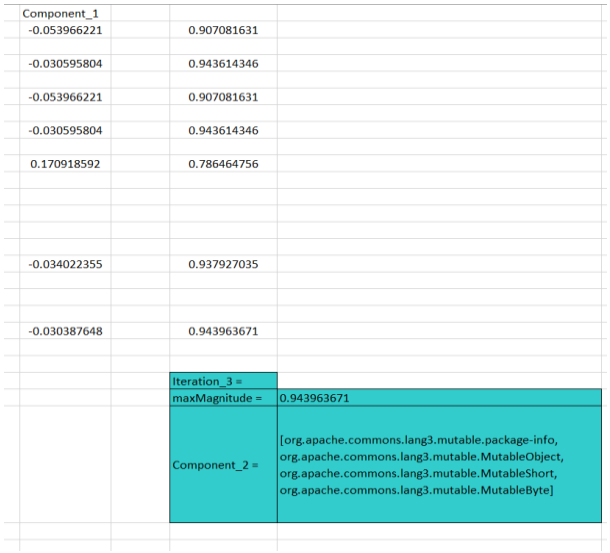

Figure 8. Merge_2 Class within Component.

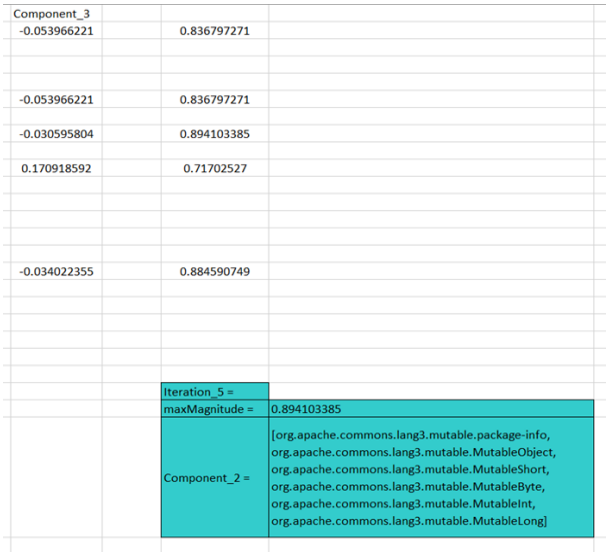

Figure 9. Merge_n Class within Component.

One of the goals of this stage is to restructure classes building and construct an optimized structure that achieves the least number of paths, which reflects the highest number of couplings between classes. The highest classes coupling within a component describes the outer coupling classes, and at the same time, it is considered the interior cohesion of the component. We search for the coupling power with the shortest distance between the classes, and we aim to build a structure optimization. The following figure 10 represent class's arrangement in the original structure. The second fig 11 is structure optimized by implementing our method.

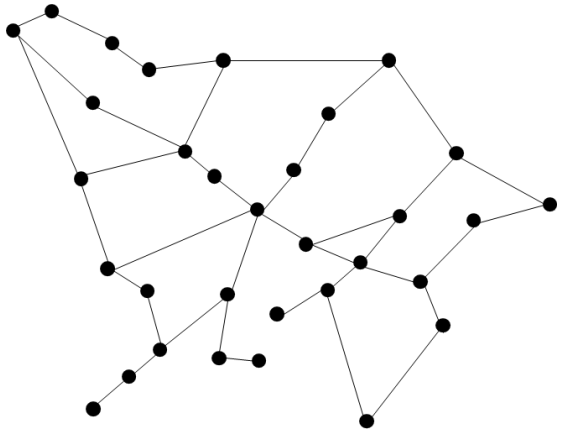

Figure 1. Actual Component

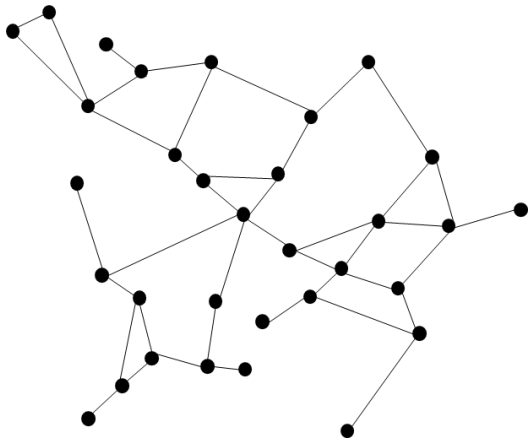

Figure 11. Mould Component

The component is small, independent units that perform a specific task, and are designed to be independent and interact with the external environment via the API, as well as interact with the rest of the components via the API.

Fifth Step:

The component is designed to be reusable, and the component resulting from the previous steps, includes many interconnected classes. We have to calculate the degree of reusability of the component so that it is eligible for deployment and work on it. To calculate the component reusability, we calculate the degree of customizability (DC) for the access field and the Component Complexness Interface (CCI). The degree of customizability (DC) is calculated from equation 10.

The weight of the class is calculated by equation 11.

Now, we calculate the $DC\_CCI\_Magnitude$ vector as the equation 13.

Finally, we estimate the component's reusability using Equation 15.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

4. Theoretic Validation of the GGODWM suite

The methodology suggested provides validity and accuracy of the proposed approach of the Good Global Optimization Dynamics Weight Metrics suite. Thus, we analyze the theoretical approach and its mathematical characteristics. Many verifications have been made by various researchers, the most one is Briand et al. [6]. In this research, we were to evaluate the study according to Briand's framework. The framework Briand et al. chose in our study because the framework used complex metrics only in its verification. Briand's framework provides mathematical principles such as system quality, code size, coupling, cohesion, and robustness.

4.1 Experimentation

In this section, the suggested metrics give main goal is to measure the Java code reusability. The standard metrics, called ISO/IEC 9126-1, provide metrics considered standard and depict outer quality features. Thus, we perform an experimental estimation of the metrics, which had been proposed in this article. The evaluation that had experimented helped mainly to validate the proposed approach. The evaluation is experimented on three principles models (Goal, question, and metrics). These principles provide a systematic execution for our proposed approach. The model GQM Goal, Question, and Metrics have five targets to be achieved, the next is illustrated:

1. The purpose of this study: Monolithic Java programming component.
2. The goal: study and analyze the Java program, particularly from the CBRS.
3. The metrics focus: on component reusability.
4. Prospect: Software designer.
5. Surrounding: Monolithic, components, and Java program.

One of the ultimate robust issues in this study regarding application measurement is the solidity of experimental estimation. Almost all experimental estimations of many measures depend only on moderate empirical support. However, we concluded a vast estimation using our suggested metrics approach. We pick two groups of application systems to apply empirical validation. The first group constructs many of the open-source Java source code. These source codes are covered in reusable components, but the method proposed targets an optimized redesign of these components. These source codes are readily available under the Apache Commons application. These source codes are prepared to be autonomous and less dependent on other applications. The other group comprises of much programs that were written without taking into account reusability, which is unlike open-source application systems. Table 1 grants and figures details regarding the various programs dealt with in the dual selected sets. These facts are the measures of the component, the number of classes for each unit, the packages that have undergone the component's environment, and a depiction of each component. Eventually, in this paper we will reply to the research questions, to justify the metric set in the suggested approach established on the experiment willing:

RQ1: What was your role in the connection between multiple metrics kits, coupling, cohesion, degree of customization, and complex interface complexity?

The response to these regard questions in our research is by locating the powerful point of the correlation between many non-parametric variables. We action the well-celebrated Spearman correlation coefficient. This correlation coefficient mostly helps us set whether all feature components are serious to the study or not.

RQ2: Does the Turbo_MQ metrics volume the identical measure used to estimate CR, or does it use other, various dimensions?

The collection of metrics suggested for our research measures the whole metrics of the component whether it has reusability or not via the CR metric. In the compose works, Turbo_MQ is a well-celebrated software fitness estimation and has an ordinary use for estimated core component modularity. This research question was demanded to limit whether the proposed CR approach estimates some of the features of the Turbo_MQ measure or not. Therefore, this article question helps us study whether the proposed CR measure gives better benefits or not.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

Table 1: Instantaneous of several components/ software.

#	Component Name	Size (KB)	Total No. Of Classes/Packages	Description
C-1: Apache Commons Components.				
1	Lang	1334	141/ 12	A library that provides many utilities for the java.lang API
2	Math	5311	265/ 22	This library provides many statistical and mathematical components that can be called within Java programs.
3	FileUpload	1614	67/ 5	It facilitates the possibility of uploading files to servlets and web applications.
4	Text	466	45/ 4	Provides many libraries with algorithms that operate text strings.
5	RNG	1144	124/ 10	It offers multiple applications for generating random numbers that are more accurate than java. util.Random
C-2 :Open-Source Software Systems.				
1	Junit 4.5	60	27/ 4	Unit testing for Java programming language.
2	Easy Mock 2.4	163	63/ 3	Java-based simulation systems, using a test panel of Java source code applications.
3	JDOM 1.1.3	230	62/ 6	Several libraries parse XML documents.
4	XGen Source Code Generator 0.3.8	290	30/ 6	Generate Java source from bound input files.
5	Servlet API 2.4	89	41/2	library that provides server-side functionality to the application

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

4.2 Results Analysis and Interpretation

The results in this section will be illustrated in detail, we show the investigational outcomes that had been studied and analyzed. The obtained results are an evaluation of our proposed approach way in reusability components. However, the outcomes were acquired, a study, analyzed, and interpreted. Many suggested metrics are calculated via automatic analysis and global dynamic weight optimization, and various metrics values are estimated for the source code. Table 2 gives information in detail about the results acquired after studying the source code and analyzing it and experimental analysis.

The obtained values for dynamic weight coupling and cohesion are necessary to estimate the total reusability component degree. Dynamic cohesion and coupling weight are present in Table 2. The results shown in Table 2, describe the structure of software that was analyzed and experimentally these analysis results in the first group of programs, which offers a reusability component score of 72.8%. After applying the proposed method and redesigning these components into structures that form an optimized structure, the outcomes indicate a significant enhancement reaching 92.4%. Finally, the second group of programs has an average component reusability is nearly 54.2%. This result indicates that the reusability component score low since this program is not dedicated to be reusing components in a manner.

The obtained results indicate that the programs in the second group have wide coupling and lower cohesion, and the coupling and cohesion results are not good after the opinion of component attributes. Therefore, the developers when designing software code do not adopt design principles according to component reusability for this software code. Most developers design components with restricted reusability. In this research, the suggested method GGODWM, studies and analyzes the monolithic program that was designed with no reusability, this approach redesigns the monolithic classes into optimized structures that meet many features such as independent component, isolated, and high reusability metric score. The results that are shown in Table 2 conclude the GGODWM approach is capable of redesigning a monolithic program and extracting components from several related classes with high cohesion and low coupling and achieving competent reusability.

Now, to prove the perfection of the results given in Table 2, the writers doing the paper [29]. In this article, a group of experts in the area of software engineering and information technology and several developers specified the reusability rate for the components. A group of 20 expert developers was randomly generated, to give an unbiased investigation way. Each developer member is supplied with software components that are variant from the other of the members, and the breaking out is based on each member's developers in the area, and he estimates the potential of reuse of the components given to him. Moreover, the research paper provided a flexible and fair comparison, as it used fixed criteria to achieve the possibility of reusability for different components, explicitly cohesion, coupling, outer dependency, portability, functional completeness, and ease of component configuration and maintenance. These criteria are very similar to the criteria chosen in our proposed GGODWM approach for many metrics. An evaluation score is assigned to the metrics from 1 to 5, where separately team participant evaluates the metric score within a range of 1 means toughly un accept, and 5 toughly accept. Finally, these values were calculated as an average by the evaluators and compared to the reusability degree of diverse software modules as shown in Table 2. The research paper indicates that practitioners manually calculated the reusability average score of the components, and these values are compared with the CR metric proposed for our method, as shown in Table 2. Moreover, in our study, this claim is verified by using statistical tools and conducting a tow-tail t-check. The p-check value stands at 2.1042×10^{-12}, so this value is much less than the near of importance, $\alpha = 0.05$. Founded on this assessment, we conclude that the proposed method GGODWM, achieves the process of scientific standards.

Table 2: Investigational Outcomes for diverse Components/software.

#	Component Name	Size (KB)	Total No. Of Classes/Packages	Description
C-1: Apache Commons Components.				
1	Lang	1334	141/12	A library that provides many utilities for the java.lang API
2	Math	5311	265/22	This library provides many statistical and mathematical components that can be called within Java programs.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

* 3	* FileUpload	* 1614	* 67/ 5	* It facilitates the possibility of uploading files to servlets and web applications.
* 4	* Text	* 4 66	* 45/ 4	* Provides many libraries with algorithms that operate text strings.
* 5	* RNG	* 1 144	* 124/ 10	* It offers multiple applications for generating random numbers that are more accurate than java. util.Random
* C-2 :Open-Source Software Systems.				
* 1	* Junit 4.5	* 6 0	* 27/ 4	* Unit testing for Java programming language.
* 2	* Easy Mock 2.4	* 1 63	* 63/ 3	* Java-based simulation systems, using a test panel of Java source code applications.
* 3	* JDOM 1.1.3	* 2 30	* 62/ 6	* Several libraries parse XML documents.
* 4	* XGen Source Code * Generator 0.3.8	* 2 90	* 30/ 6	* Generate Java source from bound input files.
* 5	* Servlet API 2.4	* 8 9	* 41/2	* library that provides server-side functionality to the application

5. Discussion

This research question aims to illustrate the relationship among various component metrics. It adopts many metrics, including coupling, cohesion, and degree of customization, interface complexity, and component reusability. To determine the score and track the relationship between any pairs of metrics, a statistical tool, the nonparametric Spearman rank coefficient is used. Here, in our study, Spearman was used instead of using Pearson, because the values of the metric measures are not normally distributed. In the normal case, the Shaparo-WiIk trial is used. p_value measured and gained in the Shaparo_WiIk regularity check is 0,0014998 and this rate is appropriately fewer than 0.05.

We used the Spearman trial by contrast to the Pearson trial in this study with the Pearson check on common data spread, different from the Spearman test. Table 3 offers the correlation of the spearman estimation among various metrics physical features. Table 3 shows the features of the cohesion and coupling suite metrics that have a power linear negative relationship. This relation reflects that when one of the metric values increases which opposite decreases to another metric. This study evaluates the component based on the high level of functional cohesion. However, if a component has a cohesion degree, to interpret this, means that the component has highly tight functionality, on the other side the coupling of the component is lower in an automatic manner. This confidence supports the Spearman correlation outcomes and offers the power correlation that is obtained from the comparison between coupling and cohesion. In another comparison, the relationship between component complexity interface and degree of customization is linear. Therefore, the Spearman correlation between other metrics such as degree of customization, cohesion, coupling, completeness, and complexness is

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

stated in a low correlation. Figure 12, shows the relationship in detail. The picture reflects that the components reusability suite metric is power positively about the cohesion possession and power negative about the coupling possession. Component reusability grows as cohesion grows, and component reusability too growth as coupling reduction, and vice versa. On the other hand, the component reusability is negative relative to the degree of customization features; growing hardness in customizing components is the opposite reduction in reusability.

Table 3: Spearman's Relationship Coefficient Standards.

*	* CCoh	* CCup	* DC	* CCI	* CR
* Co3h	* 1	*	*	*	*
* Co3p	* -0.92504	* 1	*	*	*
* PC	* -0.44366	* 0.392565	* 1	*	*
* CCI	* 0.424413	* -0.37364	* -0.49332	* 1	*
* CR	* 0.977086	* -0.95618	* -0.40904	* 0.467412	* 1

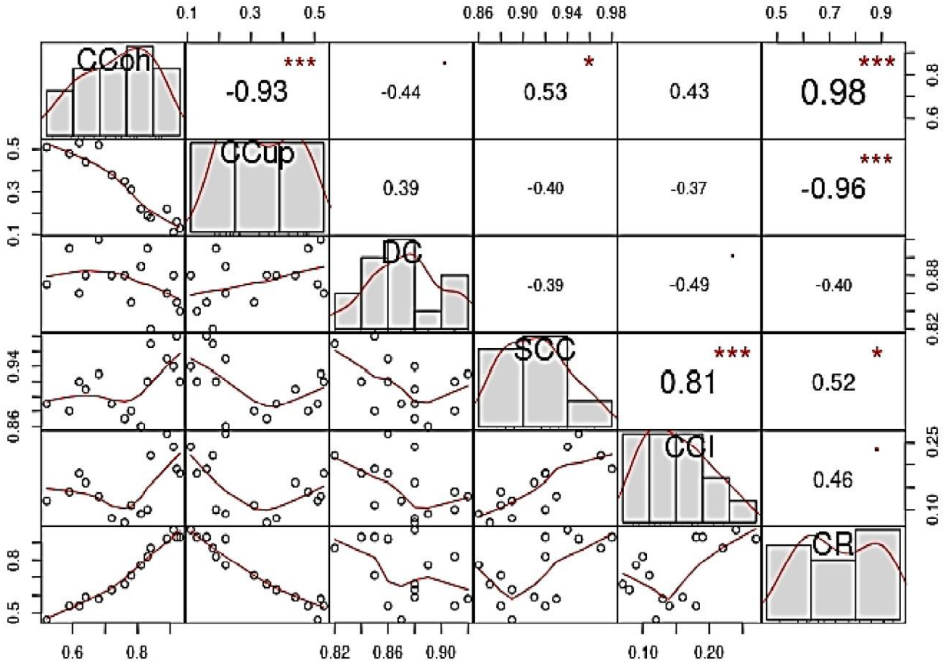

Figure 12. Pairwise Spearman Correlation Coefficient values.

Turbo_MQ is a common metric that is used by many developers from various fields, it is used to estimate the score of modules in the application system [30]. The Turbo_MQ works to measure and assess the quantity based on the nature of the components that have inner relation and outer dependency correlation. Table 4 reveals the gain of the Turbo_MQ and CR metrics for a lot of multiple software systems studied in our investigation. Therefore, the Turbo_MQ degree value shown in Table 4 illustrates the metric average of the component and fitting to the software source code. The Turbo_MQ score does not have a maximum value, and if this metric is high, it is considered better. The score is good if the metric score is high and the component is considered to have a high reusability. The result is presented in Table 4. We observed that several components such as (Servlet API, and JUnit) have an up Turbo_MQ value, so, their reusability has a small CR score. Therefore, to illustrate the correlation among dual variables, study, and analysis have been taken of the many variables by on interconnection the correlation coefficient of Spearman, then the results obtained from this relationship are depicted via the distribution map in Figure 13. The schema plainly shows that the relationship between Turbo_MQ and CR is non-linear. The R_square mark amount of the plot is 0.4798. See the plain that the cover between the Turbo_MQ and CR metrics is solely 48%. The algebraic scanning shows that the line state prototypical between the intentional Turbo_MQ

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

and the proposed GGODWM CR metrics greet to the data with only 48% of the variation of the answers variable [30]. The residual 52% conflict in the answer rate data outcomes is inexplicable. Then, it was defined in this article that the proposed GGODWM method can overlay additional characteristics that the Turbo_MQ metric cannot cover alone in general. Hence, the metric of our proposed method GGODWM to estimate the component score is justified. Therefore, developers can use the proposed method by adopting the reusability metric to guide the whole quality of the component, extract components from object-oriented programs, and redesign them with an optimized structure.

Table 4: Turbo_MQ V/S CR metric.

* #	* Category	* Component Name	* TurboMQ	* CR
* 1	* C-1	* Lang	* 0.65	* 0.71
* 2		* Math	* 0.87	* 0.89
* 3		* FileUpload	* 1.43	* 0.82
* 4		* Text	* 0.83	* 0.66
* 5		* RNG	* 0.65	* 0.77
* 6	* C-2	* Junit 4.5	* 1.25	* 0.54
* 7		* Easy Mock 2.4	* 2.04	* 0.54
* 8		* JDOM 1.1.3	* 1.56	* 0.59
* 9		* XGen Source Code Generator 0.3.8	* 1.56	* 0.46
* 10		* Servlet API 2.4	* 1.05	* 0.58

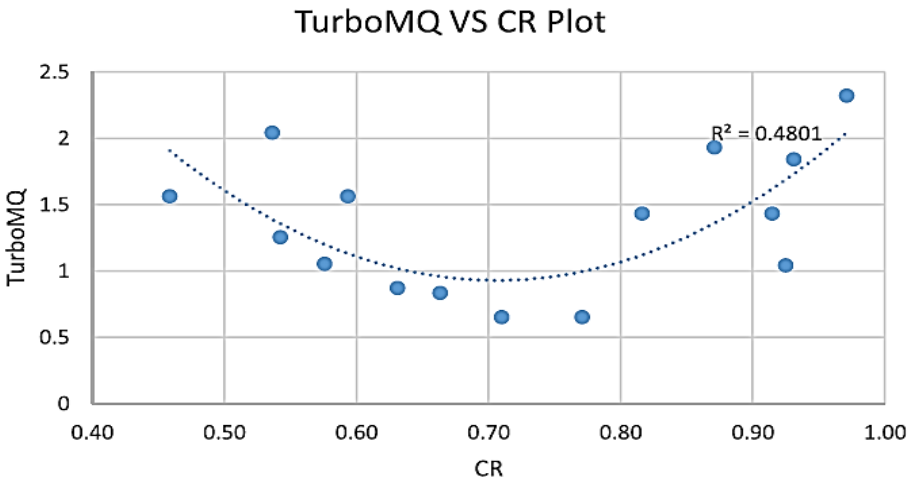

Figure 2. Dot Scheme between Turbo_MQ and CR metric.

6. Conclusion and Future Work

In the modern technology industry and the increasing use of CBRS for information technology, component libraries have a rising demand for multiple uses and compatibility with different functions. These components greatly contribute to the architecture level's reusability during software's technological development. However, this requires redesigning the software and components into an optimized architecture and measuring them to compare them with similar components. From the proposed method, the most suitable CSBD component can be selected. Therefore, the goal of our study is to propose a dynamic component identification method with an optimized structure and a set of metrics to estimate component reusability designed for the Java language.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

An automatic method has been proposed to measure reusability through component redesign and extracting a component with an optimized structure from monolithic programs through many metrics' properties, including cohesion, coupling, customization, and complexness of the component interface. Founded on the values of the proposed metrics, the reusability of the final component was estimated. Our study presents a method with a set of metrics against two categories of Java programs. These two categories contain a wide range of programs, containing ready-made components, and object-oriented programming systems designed without taking into account the basic standards for building the component. The proposed method was tested experimentally and statistically. The results give very good indicators of the components extracted from monolithic programs intended in the Java language.

Our objective in this labor is to provide approved metrics to estimate the reusability of a component, and it was designed for the Java language. Conversely, our projected method has a broader scope, and developers and researchers can use this method to develop programs and systems.

Our upcoming labor in this area omits evolving metrics and other methods that calculate a set of metrics proposed for components of the JavaBeans framework. Many methods and metrics can be used with search engines for programs and tools that are arranged according to their reusability, as it helps developers to study and analyze components of JavaBeans in the shortest possible time.

Funding: "This research received no external funding"

Conflicts of Interest: "The authors declare no conflict of interest."

References

[1] O. Of, "Empirical Study of Object-Oriented Metrics," vol. 5, no. 8, pp. 149–173, 2006.
[2] A. Ampatzoglou, S. Bibi, A. Chatzigeorgiou, P. Avgeriou, and I. Stamelos, "Reusability Index: A Measure for Assessing Software Assets Reusability," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), R. Capilla, B. Gallina, and C. Cetina, Eds., *Cham: Springer International Publishing*, 2018, pp. 43–58. doi: 10.1007/978-3-319-90421-4_3.
[3] R. Anguswamy and W. B. Frakes, "A Study of Reusability, Complexity, and Reuse Design Principles," *International Symposium on Empirical Software Engineering and Measurement*, pp. 161–164, 2012, doi: 10.1145/2372251.2372280.
[4] G. H. Anthes, "Software Reuse Plans Bring Paybacks," ComputerWorld, vol. 27, no. 49, pp. 73–76, 1993.
[5] M. F. Bertoa, J. M. Troya, and A. Vallecillo, "Measuring the Usability of Software Components," *Journal of Systems and Software*, vol. 79, no. 3, pp. 427–439, 2006, doi: 10.1016/j.jss.2005.06.026.
[6] S. Bi, X. Dong, and S. Xue, "A Measurement Model of Reusability for Evaluating Component," 2009 1st *International Conference on Information Science and Engineering, ICISE* 2009, no. 2, pp. 20–22, 2009, doi: 10.1109/ICISE.2009.58.
[7] M. A. S. Boxall and S. Araban, "Interface Metrics for Reusability Analysis of Components," *Proceedings of the Australian Software Engineering Conference, ASWEC*, vol. 2004, pp. 40–51, 2004, doi: 10.1109/ASWEC.2004.1290456.
[8] H. Zuse, "Reply to: Property-Based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, p. 533, 1997, doi: 10.1109/32.624309.
[9] I. Crnković, S. Sentilles, V. Aneta, and M. R. V. Chaudron, "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2011, doi: 10.1109/TSE.2010.83.
[10] W. B. Frakes and G. Succi, "An Industrial Study of Reuse, Quality, and Productivity," *Journal of Systems and Software*, vol. 57, no. 2, pp. 99–106, 2001, doi: 10.1016/S0164-1212(00)00121-7.
[11] A. B. Furht, Handbook of Internet Computing. CRC Press, 2019. doi: 10.1201/9781351072632.
[12] A. Gosain and G. Sharma, "Object-Oriented Dynamic Complexity Measures for Software Understandability," *Innovations in Systems and Software Engineering*, vol. 13, no. 2–3, pp. 177–190, 2017, doi: 10.1007/s11334-017-0304-3.
[13] A. Rathee and J. K. Chhabra, "Reusability in Multimedia Softwares Using Structural and Lexical Dependencies," *Multimedia Tools and Applications*, vol. 78, no. 14, pp. 20065–20086, 2019, doi: 10.1007/s11042-019-7382-1.
[14] G. Gui and P. D. Scott, "Measuring Software Component Reusability by Coupling and Cohesion Metrics," *Journal of Computers*, vol. 4, no. 9, pp. 797–805, 2009, doi: 10.4304/jcp.4.9.797-805.
[15] A. Rathee and J. K. Chhabra, "Metrics for Reusability of Java Language Components," Journal of King Saud University - *Computer and Information Sciences*, vol. 34, no. 8, pp. 5533–5551, Sep. 2022, doi: 10.1016/j.jksuci.2022.05.010.
[16] M. D. Papamichail, T. Diamantopoulos, and A. L. Symeonidis, "Software Reusability Dataset Based on Static Analysis Metrics and Reuse Rate Information," *Data in Brief*, vol. 27, p. 104687, 2019, doi: 10.1016/j.dib.2019.104687.

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025

[17] R. Harrison, S. J. Counsell, and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491–496, 1998, doi: 10.1109/32.689404.
[18] L. X. Hqj et al., "3DUDPHWHU (VWLPDWLRQ 0HWKRG RI ([SHULPHQWDO' DWD % DVHG RQ UH \' LVWDQFH 0HDVXUH," pp. 438–441, 2020.
[19] J. Guerrero-García, J. M. González-Calleros, J. Vanderdonckt, and J. Muñoz-Arteaga, "A Theoretical Survey of User Interface Description Languages: Preliminary Results," 2009 Latin American Web Congress - *Joint LA-WEB/CLIHC Conference*, pp. 36–43, 2009, doi: 10.1109/LA-WEB.2009.40.
[20] V. Gupta and J. K. Chhabra, "Dynamic Cohesion Measures for Object-Oriented Software," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 452–462, 2011, doi: 10.1016/j.sysarc.2010.05.008.
[21] M. D. Papamichail and A. L. Symeonidis, "A Generic Methodology for Early Identification of Non-Maintainable Source Code Components through Analysis of Software Releases," *Information and Software Technology*, vol. 118, p. 106218, 2020, doi: 10.1016/j.infsof.2019.106218.
[22] M. D. Papamichail, T. Diamantopoulos, and A. L. Symeonidis, "Measuring the Reusability of Software Components Using Static Analysis Metrics and Reuse Rate Information," *Journal of Systems and Software*, vol. 158, p. 110423, 2019, doi: 10.1016/j.jss.2019.110423.
[23] J. F. Hair, C. M. Ringle, and M. Sarstedt, "Partial Least Squares Structural Equation Modeling: Rigorous Applications, Better Results and Higher Acceptance," *Long Range Planning*, vol. 46, no. 1–2, pp. 1–12, 2013, doi: 10.1016/j.lrp.2013.01.001.
[24] A. Pawar and V. Mago, "Calculating the Similarity Between Words and Sentences Using a Lexical Database and Corpus Statistics," *arXiv preprint arXiv*:1802.05667, 2018, [Online]. Available: http://arxiv.org/abs/1802.05667.
[25] J. Zhang, Y. Liu, and T. Wang, "A Framework for Measuring Software Component Reusability Based on Design Patterns," "*Journal of Software Engineering and Applications*", vol. 14, no. 3, pp. 113-126, 2021. doi: 10.4236/jsea.2021.143008.
[26] S. Kumar and P. Sharma, "Evaluating Reusability and Maintainability of Software Components Using Metrics," "*International Journal of Software Engineering and Knowledge Engineering*", vol. 34, no. 2, pp. 145-163, 2023. doi: 10.1142/S0218194023500057.
[27] H. Washizaki, H. Yamamoto, and Y. Fukazawa, "A Metrics Suite for Measuring Reusability of Software Components," *Proceedings - International Software Metrics Symposium*, vol. 2003-Janua, pp. 211–223, 2003, doi: 10.1109/METRIC.2003.1232469.
[28] B. Koteska and G. Velinov, "Component-Based Development: A Unified Model of Reusability Metrics," *Advances in Intelligent Systems and Computing*, vol. 207 AISC, pp. 335–344, 2013, doi: 10.1007/978-3-642-37169-1_33.
[29] J. Sametinger, Software Engineering with Reusable Components. *Springer Science & Business Media*, 1997. doi: 10.1007/978-3-662-03345-6.
[30] A. Sharma, R. Kumar, and P. S. Grover, "Empirical Evaluation and Validation of Interface Complexity Metrics for Software Components," *International Journal of Software Engineering and Knowledge Engineering*, vol. 18, no. 7, pp. 919–931, 2008, doi: 10.1142/S0218194008003957

DOI: https://doi.org/10.54216/JCIM.170108
Received: February 23, 2025 Revised: May 30, 2025 Accepted: July 09, 2025