iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 10 **Issue:** IX **Month of publication:** September 2022

DOI: https://doi.org/10.22214/ijraset.2022.46808

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue IX Sep 2022- Available at www.ijraset.com

An Experimental Assessment on Effects of Solid Design Principles on the quality of Software using CKJM Metric Analysis

Bhaumik Tyagi[1], Yusra Beg[2]
[1, 2]*Department of Computer Science and Engineering Sharda University*

Abstract: ***Design is one of the imperative segments of the software development life cycle, which directly influences the project's entire life cycle. If the design is proficient, all the other stages of the Software development life cycle like coding, support, and maintenance will be hassle-free. The design has a substantial role that precisely affects the performance and quality of software. Today, the realm of Agile Methodology has steered the developer to contend with the newest tech stacks and features in the marketplace. Though, if there are no standard guidelines available it is intricate for the raw consumers to uphold the design quality. Software Design with appropriate patterns and principles can augment the software maintainability, reusability, and scalability. To maintain standards of software design, some principles are introduced with consideration of concepts like Cohesion and Coupling. This research paper emphasizes experimental scrutiny to attest to the SOLID design principles instructions by applying the design principles to a working prototype and then assessing the prototype using CKJM (Chidamber & Kemerer Java Metrics).***

Keywords: CKJM metrics, Software design, SOLID Principles, Software quality, Design Principles.

I. INTRODUCTION

Software design aids to visualize the inclusive system and decreases the cost implicated in developing the project. While developing software, it is not an easy task to identify the feasibility of the actual requirements at the very start of the project, therefore the design should support scalability which will allow the induction of the new requirements into the software architecture. To support scalability there are a few important factors like Rigidity (which specifies the difficulty measure of changing the software), Immutability (Inability to reuse software from other projects or parts of the software from the same project), Viscosity (Inability to preserve the design of the system), Fragility (Tendency of the software to break every time it is changed). The presence of these four factors results in poor architecture. Any application that demonstrates these factors is actually suffering from a bad design. To handle these factors we have a set of guidelines introduced by Robert Martin in 2000 called “The Design Principles”. The SOLID acronym was introduced later, around 2004, by Michael Feathers.

In this research paper, the Automobile system has been implemented with and without solid design principles. Also, there is a comparison of both designs using CKJM metric analysis.

A. Outline Of Solid Principles

1) *The Single-Responsibility Principle:* "There should never be more than one reason for a class to change." In other words, every class should have only one responsibility or a single responsibility. Suppose, you have a class that is reading and writing information from databases or validating inputs, logging events, and also performing business logic, so that class is one with a lot of responsibilities, this is something we call a direct violation of the Single Responsibility Principle. How can you recognize a class that might be violating the Single Responsibility Principle? By observing class properties like Tight Coupling, No separation of concerns, and Low cohesion. Tight coupling means altering one class’s outcomes by having a modification of other classes to get the code working again. Low Cohesion means methods and functions which are not related to each other in any meaningful way but are enclosed within a class. A good way to spot this is if methods in a class don't reuse the same fields. Each method is using different fields from the class. No Separation of Concerns distinguishes the program into segments that deal with each concern.

   A class should concentrate on doing one & only one thing. Suppose, we have an interface Modem with two responsibilities:


```
Interface Modem{
    public void dial(String pro);
    public void hangup();
    public void send (char c);
    public char recv();
}
```

The dial and hangup have responsibility for Connection management & send and receive have responsibility for Data Communication. We'll separate the same interface into two interfaces Data Communication and Communication Management:

```
interface DataChannel{
    public void Send(char c);
    public char recv();
}


interface Connection{
    public void dial(String phn);
    public char hangup();
}
```

Some benefits of using SRP are" The class is easier to understand, The class is easier to maintain, The class is more reusable, and Frequency and Effects of Change."

2) *The Open-Closed Principle:* "Software entities should be open for extension, but closed for modification. "You should be able to change the behavior of a method without changing it's source code. Code that we don't alter is less likely to create bugs due to unforeseen side effects. Here's an example of some code that is not closed for modification. We'll use a switch statement that will perform something different for each transport type. If a new transport type needs to be handled by our program then we need to modify the switch statement; violating the Open/Closed Principle. In order to achieve Open Closed Principle in our code, we would probably use Parameters, Inheritance, and Composition.

```
class CoffeeMachine{
    ...
    //Settings are dynamic
    void roastbySetting(RoastSetting Setting)
    {
        ...
        roastingService.roast(&setting)
    }
}
```

This class implements the open-closed principle. When a new RoastSetting is added, the CoffeeMachine class code does not need to update. If the RoastSetting type changed, the CoffeeMachine class would still not need to be changed.

3) *The Liskov Substitution Principle:* "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." The Liskov Substitution Principle is basically stating that this 'is-a' relationship is not good enough for maintaining a clean code. We should examine the relationship further and explore if we can slot in 'is-substitutable-for' instead. The type "A" is derived from type "B" then you should be able to substitute objects of type "B" for objects of type "A". Some benefits of applying Liskov Substitution Principle are:" Flexibility, Well-defined abstractions, Reusable code."

```
class Roaster{
    public:
    virtual void roast();
}
class CoffeeRoaster: Roaster{
    public:
    void roast(){
        //Specific coffee implementation
    }
}
class EspressoRoaster: Roaster{
    public:
    void roast(){
        //Specific espresso implementation
    }
}
.... Usage
void roast(Roaster roaster){
    roaster.roast()
    //Doesn't care about type
}
```

The roaster is Base class, two specific implementations of Coffee and Espresso, Outside functions can unknowingly use either the coffee or espresso implementation.

4) *The Interface Segregation Principle:* "Clients should not be forced to depend upon interfaces that they do not use."
Some benefits of applying Interface Segregation Principle are:" Possible reduction in compile time, Maintainability, Proper separation of concerns."

```
class AllInOneCoffeeMachine: RobustMachine{
    public:
    void roast()_
    void pour()_
    void grind()_
}
class SimpleCoffeeMachine: Pourer{
    public:
    void pour()_
}
```

This class can inherit from multiple interfaces to bring in all the functionality that it needs. Individual interfaces allow clients to use only what they need.

5) *The Dependency Inversion Principle:* "Depend upon abstractions, not concretions." High-level modules (classes that depend upon other, low-level classes of a program) should not depend on low-level modules directly. They should both depend upon an abstraction. High-level and Low-level modules should depend on abstractions rather than being dependent on each other. Some benefits of the Dependency Inversion Principle are "Loose coupling of software, Huge benefit to code reusability, Proper separation of concerns."
Inversion of Control (IoC) means to create instances of dependencies first and latter instance of a class (optionally injecting them through constructor), instead of creating an instance of the class first and then the class instance creating instances of dependencies. Inversion of control is a design used for dissociating components and layers in the system. The design is implemented by injecting dependencies into a component when it is constructed. These dependencies are generally imparted as interfaces to extend decoupling and support testability.
Although the SOLID principles apply to any object-oriented design, they can also form a core philosophy for methodologies such as agile development or adaptive software development.

II. METHODOLOGY

A. *Case 1: Software Design without Solid Design Principles*

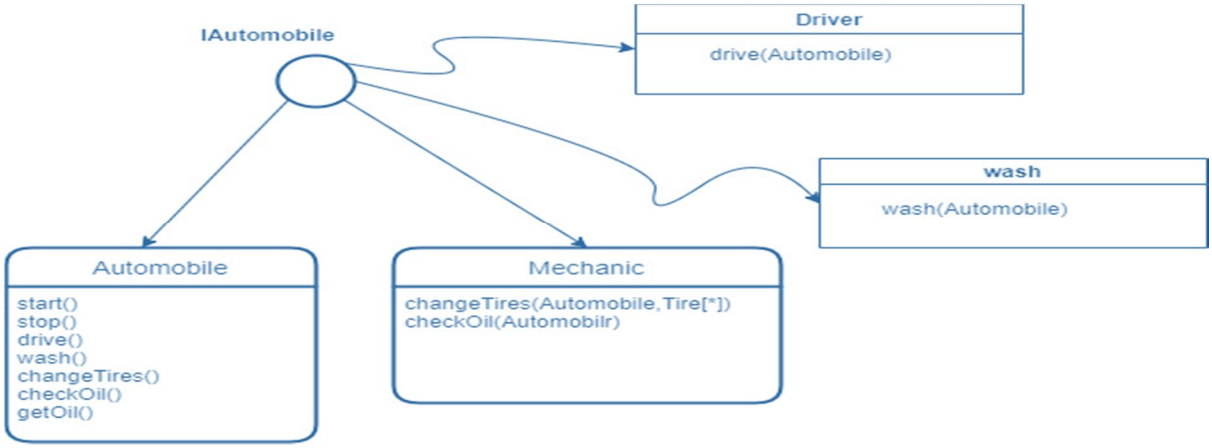


Fig 1: Without SOLID Design Principles

The first design implementation is arbitrary and does not comprise the SOLID principles.

1) *Violation of SRP:* The Automobile class should not be concerned about the Driver and wash. Instead, it should be responsible for stop, start, drive, wash, changeTires, checkOil, getOil. Interface IAutomobile should not be responsible for checking Automobile and Mechanic together.
2) *Violation of OCP:* Both the class and the interface code for IAutomobile need to be changed while adding a new Mechanic or Automobile object in the system.
3) *Violation of LSP:* Since the base class Automobile is tightly coupled with the Mechanic types, we cannot use this logic for any other type of method. Such base class behavior does not make any sense for the derived class and thus we cannot use the derived class object as a base class reference in the current system.
4) *Violation of ISP:* The IAutomate interface contains the different contractual APIs in a single interface. Instead, the data of drive and wash should be segregated in different contracts.
5) *Violation of DSP:* The Mechanic classes are tightly coupled with Automatic dependency since all Mechanic classes are creating a new instance of the Automobile class. Thus, any property change in the Mechanic class requires a change in all classes where the dependency has been tightly coupled.

B. *Case 2: Design with Solid Design Principles*

The second approach has been designed by inducing SOLID design principles resulting in the addition of a few interfaces and classes. In the new design, there is an IMechanic interface, which defines the contract for any type of method in the system.

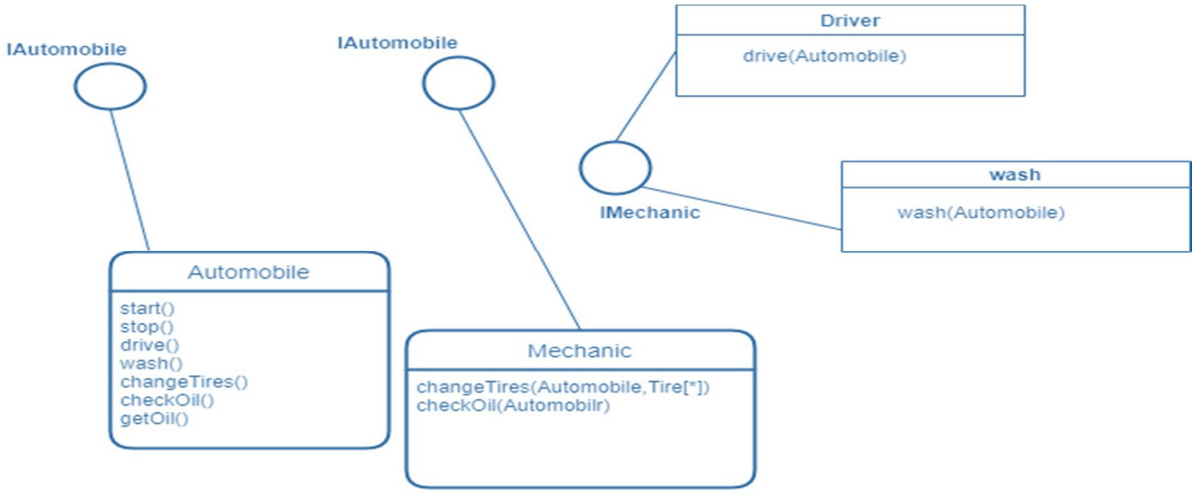


Fig 2: With SOLID Design Principles

1) *Reflection of SRP:* Every class in the system has its separate responsibilities. Driver class is only responsible for drive calculation without being concerned about the type of wash.
2) *Reflection of OCP:* The addition of a new mechanic does not require any changes in the Automatic class except for the addition of a new class of IMehcanic type.
3) *Reflection of LSP:* If we want to use the same Automatic logic for any other type of object, then the current base class can be used as a pointer to the derived classes as it is not dependent on the type of object. The method should only be of IMechanic interface type.
4) *Reflection of ISP:* The interfaces have been segregated as per the responsibilities. The IAutomatic interface has been defined for Automatic and the IMechanic interface has been defined for Mechanic, giving clear meaning to the methods.
5) *Reflection of DSP:* The classes dealing with Mechanic do not need to worry about the objects of Automatic or driver. The IMechanic removes this dependency and uses a single instance of Mechanic.

III. EXPERIMENTATION

A. CKJM Metric Analysis for both Cases

CKJM is an open-source command line tool that calculates CK metrics for Java programs. The CKJM tool calculates object-oriented metrics by processing the byte code of compiled Java files. The following six metrics proposed by Chidamber and Kemerer are calculated for each Java class.

1) WMC - Weighted Methods per Class

The WMC metric is simply the sum of the complexities of its methods. As a measure of complexity we can use the cyclomatic complexity, or we can arbitrarily assign a complexity value of 1 to each method.
By default, CKJM assigns a complexity value of 1 to each method, and therefore the value of the WMC is equal to the number of methods in the class.

2) DIT - Depth of Inheritance Tree

The DIT metric provides a measure of the inheritance levels for each class. In Java, the minimum value of DIT is 1 since all the classes inherit the default ‘Object’ class**.**

3) NOC - Number of Children

The NOC metric simply measures the number of immediate descendants of the class.

4) CBO - Coupling Between Object Classes

The CBO metric represents the number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

5) RFC - Response for a Class

The RFC metric measures the number of different methods that can be executed when an object of that class receives a message. CKJM gives a rough approximation of the response set by simply inspecting method calls within a class.

6) LCOM - Lack of Cohesion in Methods

The LCOM metric sums the sets of methods in a class that is not related by the sharing of the class's methods. The original definition of this metric (the one used in CKJM) considers all the method pairs of a class. The lack of cohesion in methods is then calculated by subtracting the number of method pairs that share field access from the number of method pairs that don’t. share field access the number of method pairs that do.

7) NPM - Number of Public Methods

The NPM metric basically sums the approaches in a class that is stated as public. It can be used to measure the size of an API delivered by a package.
The results of the important reproducible empirical research studies have been specified as follows:

Table 1: CKJM Metric without SOLID design principle

Class Name	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM
IAutomobile	4	1	0	4	12	4	4	2
Driver	4	1	0	4	5	3	3	4
Wash	2	1	0	2	3	2	1	3
Mechanic	3	1	0	3	6	5	2	6
Automobile	4	1	0	1	3	4	1	1
ChangeTires	1	1	0	2	6	2	0	0

Table 2: CKJM Metric with SOLID design principle

Class Name	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM
IMechanic	1	1	0	2	1	2	1	1
IAutomobile	2	1	0	2	4	3	2	1
Driver	1	1	0	1	1	4	0	2
Wash	1	1	0	1	2	1	1	2
Mechanic	2	1	0	0	1	2	0	0
Automobile	1	1	0	0	2	1	0	0
ChangeTires	0	1	0	1	2	1	1	0

IV. RESULT

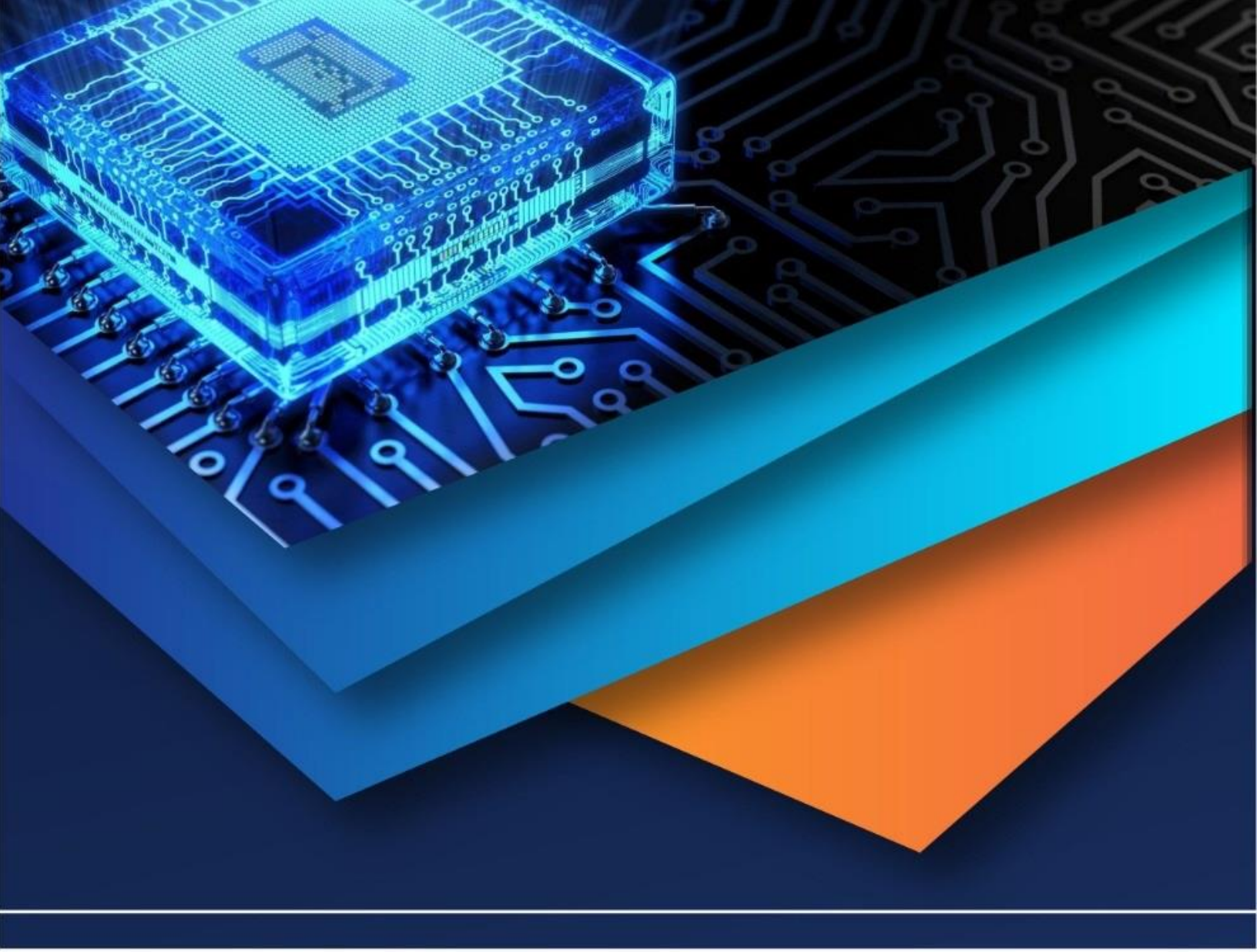
These principles are proposed in order to create remarkable software design and authentication metrics tools. In order to measure these metrics, applications used various tools. In this very research, CKJM metric tool has been used. To assess the results for both cases or types of implementation designs, the aforementioned metric has been used. To structure the software design prediction models, an examination of the calculated metrics has been used for further learning about various amalgamations of design principles. The analysis shows that the implementation of design principles in an application can reduce the dependency factor and can help in developing a scalable architecture. For above-discoursed application, the quality has been enhanced by decreasing the coupling and proposing the cohesion measure. The amalgamation of SOLID and accumulated metrics can be used in further research areas in order to identify the factors and investigate whether they have a statistically substantial impact on all type of designs.

V. CONCLUSION

Do not aim to attain SOLID, use Solid to attain maintainability. Solid design principles are just principles, not rules. It's not a compulsion to apply SOLID principles in even a small codebase. It becomes a necessity while dealing with a large codebase. Always use common sense while applying SOLID. For the sake of SRP, must avoid over-fragmenting of code. As the demand for software has diversified during the last few years leading to the rapid development of the software application, the focus has now shifted to the scalability of the Software design. While working with the latest development tech stacks and methodologies, the software should be able to scale itself as per the market requirements. Software Design should be based on Solid principles so that it would be effortless to reuse and scale the amenities. In this research work, a comparative study on the SOLID Design Principles has been done demonstrating their use in avoiding an immoral design. It can be stated that the application of these SOLID design Principles together could lead us to create a highly maintainable and scalable system. The research demonstrates the empirical assessment of a Software application against the Design approach and evaluates the quality of software using CKJM matrices. For our sample application, we have reduced the coupling by 59% (approx.) and introduced the cohesion by 39% (approx.). As we know in software development, less coupling and high cohesion are what every project demands. As a result, this research supports and promotes the practice of employing Design principles in order to create well-organized software products.

REFERENCES

[1] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, vol. 28, January 2002.
[2] E. D. G. Neha Goyal, "Reusability calculation of object-oriented software model by analyzing ck metric," International Journal of Advanced Research in Computer Engineering and Technology, vol. 3, pp. 2466– 2470, July 2014.
[3] T. H. A. S. Saddam H. Ahmed and A. A. Sewisy, "A hybrid metrics suite for evaluating object-oriented design," International Journal of Software Engineering, vol. 6, pp. 65–82, January 2013.
[4] R. Vir and P. S. Mann, "A hybrid approach for the prediction of fault proneness in object-oriented design using fuzzy logic," Journal Academic Industrial Research, 2013.
[5] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, vol. 20, pp. 476–493, June 1994.
[6] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, April 2003.
[7] R. C. Martin, Design Principles and Design Patterns, 2000.
[8] R. L. Henrike Barkmann and W. L. owe, "Quantitative evaluation of software quality metrics in open-source projects."



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY