Analysis of Coupling between Class Inheritance and Interfaces

Mr. Siddharth Jain, Mr. Gajendra Singh

***Abstract*— Object oriented systems play an important role in real world environment. Many coupling measures have been introduced invarious surveys to identify and measure the design complexity of object oriented systems. We analyze different coupling metrics in this paper which also identifies complexity between inheritance and interface programming. This paper presents a wide discussion on measure coupling between object (CBO), number of associations between classes (NASSocC), number of dependencies in metric (NDepIN) and number of dependencies out metric (NDepOut) in object oriented programming. A measurement is done for UML class diagrams and interface diagrams. The metric values of class and inheritance diagrams have been compared to prove which concept is good to use and beneficial for developers. A suite of measures is presented that addresses two problem areas within contemporary object-oriented software measurement theory and practice, i.e. the lack of OOA measures and the lack of measures for behavioral aspects of software. We also analyze a different measure which is based on a formally defined model of object-event interaction, called the object-event association matrix.**

***Index Terms*— OOP, OOA, CBO, Inheritance**

I. INTRODUCTION

Today's market much more emphasize on software quality. This has led to an increasingly large body of work being performed in the area of software measurement, particularly for evaluating and predicting the quality of software. In turn, this has led to a large number of new measures being proposed for quality design principles such as coupling. High quality software design, among many other principles, should obey the principle of low coupling. Stevens et al., who first introduced coupling in the context of structured development techniques, define coupling as "the measure of the strength of association established by a connection from one module to another" [1]. Therefore, the stronger the coupling between modules, i.e., the more inter-related they are, the more difficult these modules are to understand, change, and correct and thus the more complex the resulting software system. Some empirical evidence exists to support this theory for structured development techniques; [2], [3].

Test-driven development (TDD) is not, despite its name, a testing technique but rather a development technique in which the tests are written prior to the source code [4]. The tests are added gradually during the implementation process and when the tests are passed, the code is re factored to improve its internal structure. This incremental cycle is repeated until all the functionality is implemented. [5]. The idea of TDD was popularized by Beck [6] in the Extreme Programming (XP) method. Therefore, although TDD seems to have just recently emerged, it has existed for decades; an early reference to the use of TDD features in the NASA Project Mercury in the 1960s [7].

Basically there are two different kinds of abstractions namely classes and interfaces. The most important difference is that a class can hold functional logic and an interface is used to organize source code and it will also provide the boundary between the levels of abstraction. According to object oriented programming, the class provides encapsulation and abstraction and the interface provides abstraction and cannot inherit from one class but can implement multiple interfaces. The above said differences are minor and they are very similar in structure, complexity, readability and maintainability of source code [8]. Here, the difference in usage of class inheritance and interface concepts are measured for class diagrams by coupling metrics proposed by Chidamber and Kemrer and Brian.

Complexity of source code directly relates to cost and quality. Many coupling models are presented in the literature to measure the possible interactions between objects and to measure design complexity. High coupling between objects increases complexity and cost. Low coupling is good for designing object oriented software. Inheritance introduces more interactions among classes [9]. This will increase the complexity. This paper presents a comparison between object oriented interfaces and inheritance class diagrams.

The remaining of this paper is organized as follows. We discuss class and object in Section 2. In Section 3 we discuss about Inheritances. In section 4 we discuss about Evolution and Recent Scenario. In section 5 we discuss about the Challenges. The conclusions and future directions are given in Section 6. Finally references are given.

II. CLASS AND OBJECT

A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object. Methods are nothing but members of a class that provide a service for an object or perform some business logic. Java fields and member functions names are case sensitive. Current states of a class's corresponding object are stored in the object's instance variables. Methods define the operations that can be performed in java programming.

Syntax:
class classname
{Methods + variables;}

Manuscript received June 08, 2014.
Mr. Siddharth Jain, M.Tech Scholar, SSSIST, Sehore
Mr. Gajendra Singh, HOD, SSSIST

An object is an instance of a class created using a new operator. The new operator returns a reference to a new instance of a class. This reference can be assigned to a reference variable of the class. The process of creating objects from a class is called instantiation. An object encapsulates state and behavior.

An object reference provides a handle to an object that is created and stored in memory. In Java, objects can only be manipulated via references, which can be stored in variables. Creating variables of your class type is similar to creating variables of primitive data types, such as integer or float. Each time you create an object, a new set of instance variables comes into existence which defines the characteristics of that object. If you want to create an object of the class and have the reference variable associated with this object, you must also allocate memory for the object by using the new operator. This process is called instantiating an object or creating an object instance.

The class diagram is the main building block in object oriented modeling. It is used both for general conceptual modeling of the systematic of the application, and for detailed modeling translating the models into programming code. The classes in a class diagram represent both the main objects and or interactions in the application and the objects to be programmed. In the class diagram these classes are represented with boxes which contain three parts:

- The upper part holds the name of the class
- The middle part contains the attributes of the class
- The bottom part gives the methods or operations the class can take or undertake.

The example of BankAccount class is shown in fig 1

.

Fig 1 Class Diagram

An object diagram in the Unified Modeling Language (UML), is a diagram that shows a complete or partial view of the structure of a modeled system at a specific time.An Object diagram focuses on some particular set of object instances and attributes, and the links between the instances. A correlated set of object diagrams provides insight into how an arbitrary view of a system is expected to evolve over time. Object diagrams are more concrete than class diagrams, and are often used to provide examples, or act as test cases for the class diagrams. Only those aspects of a model that are of current interest need be shown on an object diagram(fig2).

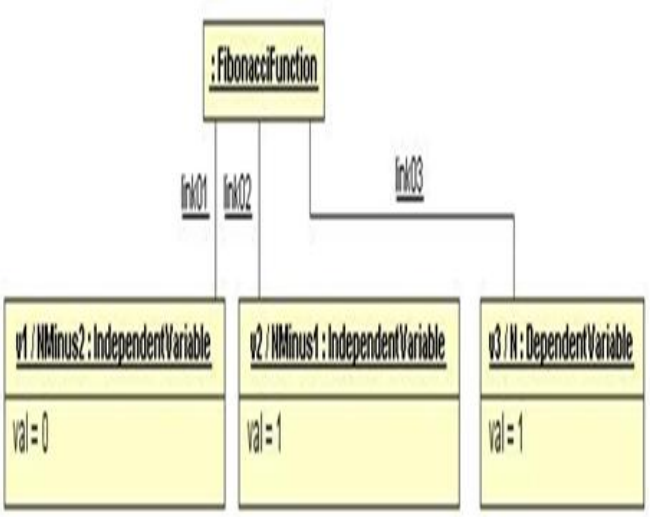

Fig 2 Object Diagram

III. INHERITANCE

In object-oriented programming (OOP), inheritance is a way to compartmentalize and reuse code by creating collections of attributes and behaviors called objects which can be based on previously created objects. In classical inheritance where objects are defined by classes, classes can inherit other classes. The new classes, known as subclasses (or derived classes), inherit attributes and behavior of the pre-existing classes, which are referred to as super classes (or ancestor classes). The inheritance relationships of classes gives rise to a hierarchy. In prototype-based programming, objects can be defined directly from other objects without the need to define any classes. The inheritance concept was invented in 1967 for Simula. Sometimes inheritance-based design is used instead of roles. A role, say Student role of a Person describes a characteristic associated to the object that is present because the object happens to participate in some relationship with another object (say the person in student role -has enrolled- to the classes). Some object-oriented design methods do not distinguish this use of roles from more stable aspects of objects. Thus there is a tendency to use inheritance to model roles, say you would have a Student role of a Person modelled as a subclass of a Person. However, neither the inheritance hierarchy nor the types of the objects can change with time. Therefore, modelling roles as subclasses can cause the roles to be fixed on creation, say a Person cannot then easily change his role from Student to Employee when the circumstances change. From modelling point of view, such restrictions are often not desirable, because this causes artificial restrictions on future extensibility of the object system, which will make future changes harder to implement, because existing design needs to be updated. Inheritance is often better used with a generalization mindset, such that common aspects of instantiable classes are factored to superclasses; say having a common superclass 'LegalEntity' for both Person and Company classes for all the common aspects of both. The distinction between role based design and inheritance based design can be made based on the stability of the aspect. Role

based design should be used when it's conceivable that the same object participates in different roles at different times, and inheritance based design should be used when the common aspects of multiple classes (not objects!) are factored as superclasses, and do not change with time(fig3).

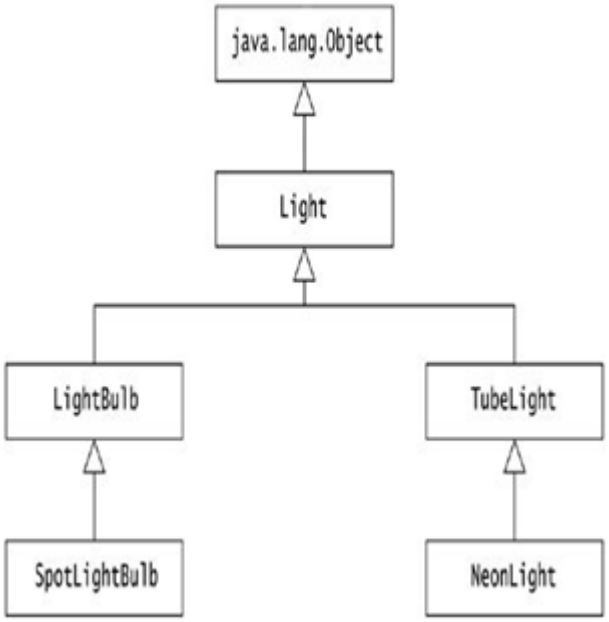

Fig 3 Inheritance

IV. Evolution Recent Scenario

Object-oriented software is based on the notions of class, encapsulation, inheritance, and polymorphism. These notions make it more challenging to design metrics for the characterization of OO-based software vis-a-vis what it takes to do the same for the purely procedural code [10], [11]. An early work by Coppick and Cheatham [12] attempted to extend the then popular program-complexity metrics,such as the Halstead [13] and the McCabe and Watson complexity measures [14], to OO software. Subsequently,other works on OO software metrics focused mostly on the issue of how to characterize a single class with regard to its own complexity and its linkages with other classes.

In 2003, G. McGraw et al.[15] proposed about extract sensitive classes i.e. classes having both data members and methods attack prone. Hence these classes are suspected to be attacked. Second step attempts to secure these sensitive classes using security mechanisms. At the end, classes are assumed to be secure merely just applying security mechanisms. But, as a matter of fact, sensitive classes even after shielded with security mechanisms may not be completely secure. The simple reason is that no mechanism can guarantee absolute security.

In 2006, S. Ardi et al. [16] describe about the loss due to Code Red Worm has been estimated to $2.6 billion and due to Nachi Worm, operations at Air Canada and CSX railroad were affected very badly.Efforts in this direction have been started but the statistics shows that the problem is still growing

In 2004 Rajib et al.[17] proposed a tool to improve software products and process, measurements which are essential in many fields. Software measurement plays an important role in finding the software quality, performance, maintenance and reliability of software products. The concept of measurement requires appropriate measurement tools to measure, to collect, to verify and validate relevant metric data.

In 2005, Marcela Genero et al. [18] proposed about measure coupling in class diagrams there are three types of metrics used in this diagram. First CK metric is added to measure coupling performance. A measure of coupling is more useful to determine the complexity. The higher the inter object coupling, the more rigorous the testing needs to be. In this paper, four metrics are used to validate the proposed approach.

Object oriented programming is more recent and more important in quality software programming than that of the old style procedural programming. In the last two decades object oriented software engineering receives much attention because object oriented technology is wide spread [19].

The class provides encapsulation and abstraction and the interface provides abstraction and cannot inherit from one class but can implement multiple interfaces. The above said differences are minor and they are very similar in structure, complexity, readability and maintainability of source code [20].

In 2010, V. Krishnapriya, et al. [21] proposed about the measurement to measure coupling between object (CBO), number of associations between classes (NASSocC), number of dependencies in metric (NDepIN) and number of dependencies out metric (NDepOut) in object oriented programming. A measurement is done for UML class diagrams and interface diagrams. The metric values of class and inheritance diagrams have been compared to prove which concept is good to use and beneficial for developers.

In 2007 , Maria Siniaalto et al. [22] reports the results from a comparative case study of three software development projects where the effect of TDD on program design was measured using objectoriented metrics. The results show that the effect of TDD on program design was not as evident as expected, but the test coverage was significantly superior to iterative test-last development.

In 2010, Simon Allier et al.[23] express existing definitions of coupling metrics using call graphs. We then compare the results of four different call graph construction algorithms with standard tool implementations of these metrics in an empirical study. Our results show important variations in coupling between standard and call graph-based calculations due to the support of dynamic features.

In 2010, Hongyu Pei Breivold et al.[24] primary studies for this review were identified based on a pre-defined search strategy and a multi-step selection process. Based on their research topics, we have identified four main categories of themes: software trends and patterns, evolution process support, evolvability characteristics addressed in OSS evolution, and examining OSS at software architecture level. A comprehensive overview and synthesis of these categories and related studies is presented as well.

In 2009, Yuming Zhou et al.[25] describe about the OO metrics that are investigated include cohesion, coupling, and inheritance metrics. Our results, based on Eclipse, indicate

that: 1) The confounding effect of class size on the associations between OO metrics and change-proneness, in general, exists, regardless of whichever size metric is used; 2) the confounding effect of class size generally leads to an overestimate of the associations between OO metrics and change-proneness; and 3) for many OO metrics, the confounding effect of class size completely accounts for their associations with change-proneness or results in a change of the direction of the associations. These results strongly suggest that studies validating OO metrics on change-proneness should also consider class size as a confounding variable.

In 2009, Alka Agrawal et al. [26] suggest an approach to identify vulnerable classes in object oriented design. The method proposed also investigates whether transitive nature of Inheritance contributes to propagation of vulnerabilities from one class to another or not. An algorithm for computing Vulnerability Propagation Factor (VPF) has been developed, which measures number of vulnerable classes because of the Vulnerability in some classes of an object oriented design.

In 2011, Narendra Pal Singh Rathore & Ravindra Gupta [27] presented an approach to measure complexity between class inheritance and interface on object oriented source code.

V. CHALLENGES

Effort of measuring vulnerability of an Inheritance hierarchy and hence an object oriented design is at very young stage. So, lacunas are obvious. One of the major limitations of the work is its applicability to only object oriented software. Also, only one aspect of object oriented design has been considered when Calculating. Inheritance, while other aspects including Encapsulation, Polymorphism, and Coupling etc. should also be taken into account.

Moreover, it is important to pay attention that the approach reached almost the half error rate than the regression for the sample, proving the advantage of to use several beniciary approaches against statistics multivariate regression.

Such studies require effort be collected on a per-class basis in a consistent and reliable manner. This is more difficult than accounting only for the total project cost. From a practical perspective, such a fine granularity may not be needed, as the typical application of a cost model is to estimate, at an early stage, the cost and risk associated with entire projects.

Chidamber, Darcy, and Kemerer have investigated the six of the design measures proposed in . Their aim was not to build accurate prediction models, but rather to test the ability of the measures to identify high effort and low productivity classes.

VI. CONCLUSIONS AND FUTURE DIRECTIONS

This paper presents discuss several concepts and on how to reduce coupling in object oriented programming. Due to the reduction in coupling, developers can produce quality rograms. When CBO is reduced reusability will be increased. High coupling will support low encapsulation and produce more faults. Due to the reduction in values of coupling metrics the stability of the structure will be good. When the coupling measures are reduced, the classes can function more independently.

Classes in object-oriented systems, written in different programming languages, contain identifiers and comments which reflect concepts from the domain of the software system. This information can be used to measure the cohesion of software. To extract this information for cohesion measurement, Latent Semantic Indexing can be used in a manner similar to measuring the coherence of natural language texts.

REFERENCES

[1] W. Stevens, G. Myers, and L. Constantine, "Structured Design,"IBM Systems J., vol. 13, no. 2, pp. 115-139, 1974.

[2] R.W. Selby and V.R. Basili, "Analyzing Error-Prone SystemsStructure," IEEE Trans. Software Eng., 1991.

[3] P.A. Troy and S.H. Zweben, "Measuring the Quality of Structured Designs," J. Systems and Software, 1981.

[4] Beck, K., Test-Driven Development By Example, Addison-Wesley, Boston, MA, USA, 2003.

[5] Astels, D., Test-Driven Development: A Practical Guide,Prentice Hall, Upper Saddle River, USA, 2003.

[6] Beck, K., Extreme Programming Explained, Second Edition:Embrace Change, Addison-Wesley, USA, 2004.

[7] G. Larman and V.R. Basili, "Iterative and Incremental Development: A Brief History", 2003, IEEE.

[8] Mathew Cochran,"Coding Better: Using Classes Vs Interfaces",January 18th, 2009.

[9] Mohsen D. Ghassemi and Ronald R. Mourant,"Evaluation of Coupling in the Context of Java Interfaces", Proceedings OOPSLA, ACM 2000.

[10] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Trans., 1994.

[11] N. Churcher and M. Shepperd, "Towards a Conceptual Framework for Object-Oriented Software Metrics," 1995.

[12] C.J. Coppick and T.J. Cheatham, "Software Metrics for Object-Oriented Systems," , 1992.

[13] M.H. Halstead, Elements of Software Science. Elsevier, 1977.

[14] T.J. McCabe and A.H. Watson, "Software Complexity," Crosstalk,J. Defense Software Eng. 1994.

[15] G. McGraw, "From the ground up: The DIMACS software security workshop," IEEE Security & Privacy, vol. 1, 2003.

[16] S. Ardi, D. Byres and N. Shahmehri, "Towards a structured unified process for software security,", ACM, 2006.

[17] Rajib Mall,"Fundamentals of Software Engineering", Chapter 1, Pg.No:1-18,2nd Edition, April 2004.

[18] Marcela Genero, Mario Piattini and Coral Calero," A Survey of Metrics for UML Class Diagrams", 2005.

[19] Pradeep Kumar Bhatia, Rajbeer Mann, " COIT-2008,RIMT-IET,Mandi Gobindgarh.

[20] Mathew Cochran,"Coding Better: Using Classes Vs Interfaces",January 18th, 2009

[21] V. Krishnapriya and Dr. K. Ramar, 2010 International Conference on Advances in Computer Engineering,IEEE.

[22] Maria Siniaalto and Pekka Abrahamsson, First International Symposium on Empirical Software Engineering and Measurement,IEEE 2007.

[23] Simon Allier, St´ephane Vaucher, Bruno Dufour, and Houari Sahraoui, 2010 Working Conference on Source Code Analysis and Manipulation,IEEE.

[24] Hongyu Pei Breivold, Muhammad Aufeef Chauhan and Muhammad Ali Babar, 2010 Asia Pacific Software Engineering Conference,IEEE.

[25] Yuming Zhou, Hareton Leung and Baowen Xu, IEEE TRANSACTIONS , SEPTEMBER/OCTOBER 2009.

[26] Alka Agrawal, Shalini Chandra and Raees Ahmad Khan, 2009 International Conference on Availability, Reliability and Security,IEEE.

[27] Narendra Pal, Ravindra Gupta 2011, WICT, ISBN no. 978-1-4673-0127-5.