Determining the Most Effective Class for Extending Reusability of Object Oriented System

Mohammad Shabbir Hasan
Khulna University of Engineering and Technology
Khulna-9203, Bangladesh
E-mail: shabbir_cse03@yahoo.com

K.M. Azharul Hasan
Khulna University of Engineering and Technology
Khulna-9203, Bangladesh
E-mail: azhasan@cse.kuet.ac.bd

***Abstract*—The object oriented programming paradigm often claimed to allow a faster development pace and higher quality of software. One of the principal goals of object-oriented software is to improve the reusability of software components. Finding the most effective class is one of the most concerned issues to improve reusability. Here effective class means the class with low coupling with respect to others and if a new module is added to this class, minimum modification would be required to achieve reusability. In this paper, a methodology based on various coupling is presented to determine the most effective class in an object oriented software.**

Keywords-Software Reusability; Coupling; Design Pattern; Object Oriented System

I. INTRODUCTION

Object-oriented (OO) system development is gaining wide attention both in research environment and in industry as there is increasing pressure on software developers to produce quality software within a very short time. This necessitates the reuse of previously developed or commercially available software elements to expedite the development process. The most common form of reuse is the reuse of code in a fine-grain manner such as objects in the object-oriented paradigm or a large grain manner such as components in the component oriented paradigm [1]. Reusability of software is considered as crucial technical precondition to improve overall software quality and reduce production and maintenance cost [2].

Software components are supposed to be better reusable and more flexible compared to conventionally developed software and so it is imperative to better understand the relations and interactions of the modules of object-oriented systems. The interactions between the internal components of software provide several goals in software construction leading to better values for external attributes such as maintainability, reusability, and reliability. Coupling is the internal software attribute, which tells how tightly the components of software modules are bound together in design or implementation. It describes the interdependency between methods and between object classes, respectively [3][4]. 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"[5]. Braind et al. defined some properties to be satisfied by the coupling measure for empirical validation [6]. Good OO software should follow the principle of low coupling and high cohesion. Strong coupling makes a system more complex, highly inter-related modules are harder to understand or change. Designing the systems with the weakest possible coupling between modules can reduce complexity. A software designer must, therefore, strive to minimize coupling to reduce the risk of error propagation across modules [7].

There are differences between necessary and unnecessary coupling. The rationale is that without any coupling the system is useless. Consequently, for any given software solution there is a basic or necessary coupling level. Such unnecessary coupling does indeed needlessly decrease the reusability of classes. There is also static and dynamic coupling measure for object oriented systems [8]. As the polymorphic method invocation is determined by run time, the coupling on this method belongs to dynamic coupling.

Eder et al. identify three different types of relationships [9]. These relationships, interaction relationships between methods, component relationships between classes, and inheritance between classes derive the different dimensions of coupling. Hitz and Montazeri [10] characterize coupling by defining the state of an object (the value of its attributes at a given moment at run time), and the state of an object's implementation (class interface and body at a given time in the development cycle). From these definitions, they derive two "levels" of coupling, Class level coupling (CLC), represents the coupling resulting from implementation dependencies between two classes in a system during the development lifecycle and Object level coupling (OLC), represents the coupling resulting from state dependencies between two objects during the run time of a system.

After years of research, many coupling metrics have been proposed to find the inter module dependencies. In this paper a study on software evaluation with the help of coupling metrics has been performed. We have categorized the coupling measures in two categories namely ratio oriented and ratio less. To validate the interactions we have selected an open source software for case study. No modification was done in the software during the analysis presented in this paper. For analyzing the coupling measures of object oriented systems based on different interactions of the classes, a parser has been developed named "Design Analyzer" to find the design pattern of the system. Such

978-1-4244-8035-7/10/$26.00 © 2010 IEEE

design analyzer is useful to get internal software architecture of software developed in Object Oriented Paradigm. Finally, by using principal component analysis [11], the two categories of measures are analyzed to find out the most effective class of an object oriented system based on CLC.

The rest of the paper is organized as follows. Section 2 gives an over view of the case study, Section 3 includes the methodology to find design pattern from source code of an object oriented system, Section 4 has the overview of some existing coupling measures, section 5 presents the issues to be considered for software design taking example of the existing coupling measures, section 6 shows the results and discussion and finally, some conclusions are outlined in section 7.

II. CASE STUDY

In this research work the analysis was performed on an open source software developed in Object Oriented paradigm. A tool named "*Design Analyzer*" is developed as a part of this research work for parsing the source code to get the design pattern. The selected software was developed prior to the analysis and no modification was done during the analysis. For proceeding in an efficient manner the process of analyzing the source codes should follow the syntax of the programming language. Brief description of the open source software used for case study is given here:

Software 1-Turtle Chat: This is chatting software like Yahoo! Messenger. This software can send and receive instant messages over Internet Protocol. There are two parts of this software: Server Part and Client Part. Here the Client side contains 19 user defined classes and the Server side contains 4 user defined classes.

III. FINDING THE DESIGN PATTERN FOR JAVA BASED OBJECT ORIENTED SYSTEMS

Gamma at el [12] defines design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." A design pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Object oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. In this paper, a tool named Design Analyzer is developed that finds the interactions, roles, collaborations and relationships of the software in a graphical format. These interactions can occur in there different ways between the classes in a Java based object-oriented software.

These inter module relationships are as follows:

1. Inter-module relationship through Return Type
2. Inter-module relationship through Argument Passing in a member function
3. Inter-module relationship through Object Declaration
4. Inter-module relationship through Inheritance

From these four types of relationship, we have defined two types of interactions as follows

- **The Operation-Operation interaction (O-O)** is defined as the interaction between two operations of two or more different objects or classes. Let OC be an operation of class C. There is an operation-operation interaction between classes C and D, if class D is the type of a parameter of operation OC or class D is the return type of OC. The first two categories (relationship through return type and argument passing) above are included in O-O interaction.

- **The Class-Class interaction (C-C)** is defined as the interaction between two classes if any one of the above two interaction occurs (i.e. interaction through object declaration or inheritance). Let C and D be two classes of an object oriented system. There is an C-C interaction between the classes C and D, if an object Od of D is declared inside class C or D is derived from class C through inheritance. The last two categories above (relationship through object declaration and inheritance) are included in C-C interaction.

We have developed the tool Design Analyzer using java for analyzing the source code of an object oriented system to get the design pattern of the system. The Design Analyzer implements the interactions of O-O and C-C. The input of the system is the source code of an Object Oriented program and output is an interaction graph of the design pattern of the software. Fig 1 shows the graphical representation of the design pattern of software 1. The design pattern is a graph <G, E> where each node G represents a class of the system and there is an edge E between two nodes if there is an interaction (O-O, C-C) between the two classes. We have considered only the user defined classes that are found in the source code. This is because our next goal is to add a new module in the system so that we can reuse the existing system with minimum modification.

Figure.1 shows the design pattern found using the Design Analyzer for software 1. From the figure we can see that there are 19 user defined classes in the system. Among them the class ChatClient is very much coupled with other classes. The ChatClient class has 16 coupling relations with others. We can see that there is no class isolated in the system. Hence this is a criterion of good design. But the distribution of coupling is based on only three classes namely ChatClient, ScrollView and Tappanel. This is a sign of low maintainability. Because if the class ChatClient fails for any reason most of the classes will be affected. Finally, we can say that to reuse a software, it is very important to know about the design pattern of the software. If it is poorly designed then it might be error prone for reuse in the future.

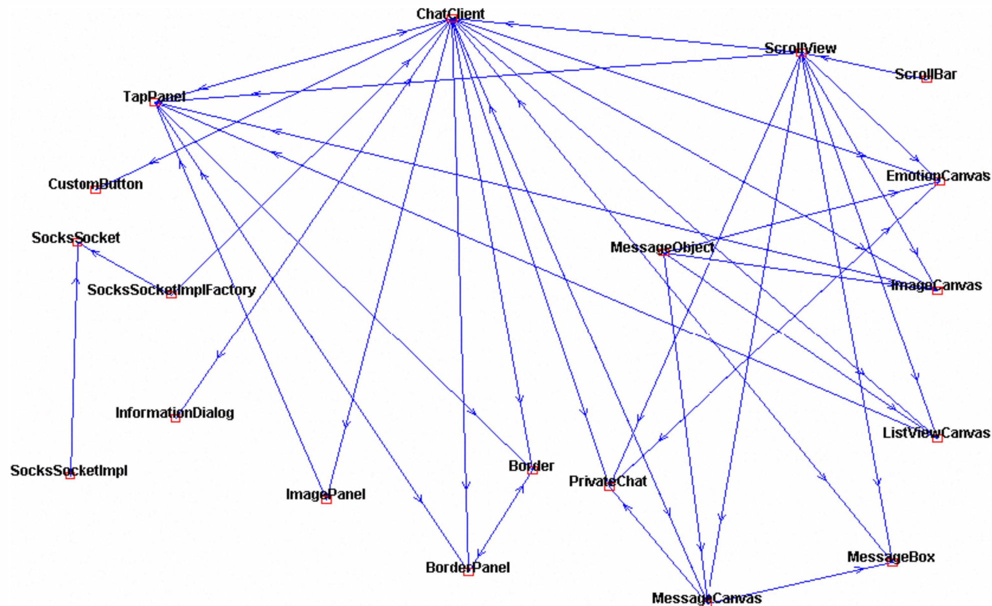

Figure 1. Graphical Representation of the relationship among the User Defined Classes of software 1

IV. THE COUPLING MEASURES

In this research work the following coupling metrics have been chosen for the analysis of inter module dependencies. A brief definition of the measures is given here. All the measures determine the coupling between components.

A. Number of used classes by dependency relation(NUCD)

This measure is used to count the total number of distinct classes with whom a particular class is creating dependency relation caused by any of dependency types (e.g. parameter, local variable, return type) to recognize the dependency between two classes.

B. Total number of evidences for "Used classes by dependency relation" (TNUCD)

This measure is used to count total number of evidences for a particular class of "Used classes by dependency relation." All types of dependencies (e.g. parameter, local variable, return type) will be used to count such evidences.

C. Number of user classes for a class through dependency relation (NUCC)

This measurement represents the total number of distinct classes who are using a particular class through dependency relations.

D. Total number of evidences for "User classes through dependency relation" (TNUCC)

This measurement counts the total number of usage evidences of a particular class by the other classes in OO design.

E. Class Coupling

The Class Coupling (CLC) is the summation of Client Coupling and Server Coupling of the class. It is the measure of the summation of out degree and in degree of a node in Class Interaction Graph (CCIG).

Class Coupling = Client Coupling (CC) + Server Coupling (SC).

The number of Coupling Relations for which a class is a client to other class is called Client Coupling for the class. It is the measure of out degree of a node in CCIG. The number of Coupling Relations for which a class is a server to other class(s) is called Server Coupling for the class. It is the measure of in degree of a node in CCIG.

F. Visible Member

The measure "Visible Member" shows the amount of members (attributes and methods) visible to other class numerically. This measure is used to find the overall members which can be called or used by other classes. Visible members are the most required criteria for direct coupling.

V. CRITERIA OF MEASURING COUPLING

We obtain an Interaction Graph using the tool *Design Analyzer* to represent design pattern of the software and coupling occurs due to this interaction. The methodology of Principal Component Analysis [11] has been adopted to select the most responsive class of a system using the values of various coupling measures. When we want to add a module we need to find a class that is less responsive. From graphical analysis we take decision to add a module in which the interaction graph has fewer edges and in this section we show some experimental result to prove the idea.

A. Experiment Design

To make a study of coupling measure we want to determine the best coupling metrics defined above. We are going to apply these measures on the software under observation. The above measures are applied on this software through principal component analysis.

Principal Component Analysis

Principal component analysis [11] is typically used to reduce the dimensionality and/or to extract new uncorrelated features from the original data. Principal component analysis involves an Eigen analysis on a covariance matrix. If the input data is represented as a matrix X of 'n' rows and 'm' columns:

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ . & . & . & . \\ . & . & . & . \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}$$

Where n = total number of classes and m = total measures.

Then, the sample mean μ_i is computed for each column, where

$$\mu_i = \frac{1}{n}\sum_{i=1}^{n} x_{ij} \quad \text{for } j = 1,2 \ldots m.$$

Then X can be centered to form X*:

$$X^* = \begin{bmatrix} x_{11}-\mu_1 & x_{12}-\mu_2 & \cdots & x_{1m}-\mu_m \\ x_{21}-\mu_1 & x_{22}-\mu_2 & \cdots & x_{2m}-\mu_m \\ . & . & \cdots & . \\ . & . & \cdots & . \\ x_{n1}-\mu_1 & x_{22}-\mu_2 & \cdots & x_{nm}-\mu_m \end{bmatrix}$$

Then covariance matrix $R = (1/n)[X^*]^T X^*$ is computed. An Eigen analysis on the covariance matrix R yields a set of positive Eigen values $\{\lambda_1, \lambda_2, \ldots, \lambda_m\}$ [11]. If the Eigen values are sorted in descending order (i.e., $\lambda_1 > \lambda_2 > \ldots > \lambda_m$), their corresponding Eigen vectors, $\{ v_1, v_2, \ldots, v_m \}$, are the principal components. The first principal component retains the most variance, if the feature vectors are projected onto the first principal component, more variance will be retained than if the vectors are projected onto any other principal component. The second component retains the next highest residual variance, and so on. A smaller Eigen value contributes much less weight to the total variance. In many cases, the first few components can retain nearly all of the variance. If the'd' most significant principal components are selected for projection of the data, then the variance (V) retained by this approximation is [11]:

$$V = \frac{\sum_{i=1}^{d} \lambda_i}{\sum_{i=1}^{m} \lambda_i}$$

V is also called the degree of accuracy for the approximation.

VI. Results and Discussions

From the graphical analysis the designer can take the decision of where to add the new module of the existing software in an efficient manner by keeping the development cost as minimum as possible. This leads to a better reusability of the existing software. From the graphical analysis of Software 1 in Figure 1, the decision that can be taken is: a new module can be added beside the classes which are not highly coupled such as: InformationDialog, CustomButton, ScrollBar, ImagePanel. Also we present a principal component analysis to find a module to interact with for reusing the module.

Table 1 shows the first 3 principal component of software 1. Here first component retains 70.30% variance and first two components retain 95.15% of the total variance. In Table 1 the lower coupling contributing modules are shown in bold face. From the 3 principal components we can see that among the negative impact values only module 9(whose name is InformationDialog and shown as underline) has negative impact to all the principal components. Hence we can take decision if one module is added interacting with only module 9 and then the purpose of the new software serves then it is a good decision to implement it. If the purpose does not serve, then we should select a module having low coupling. Also we can see from design pattern of software 1(see Figure 1) that the class InformationDialog interacts only with 1 class namely ChatClient. We have implemented our approach for software adding a new class StatusArea and we found that the system is working well serving the new purpose.

Figure 2 shows the design pattern after adding the new module StatusArea.

VII. Conclusion

Identification of design patterns from source code is one of the most promising methods for improving software maintainability and reusing design experience. It is hard or even impossible to understand poorly documented legacy

\

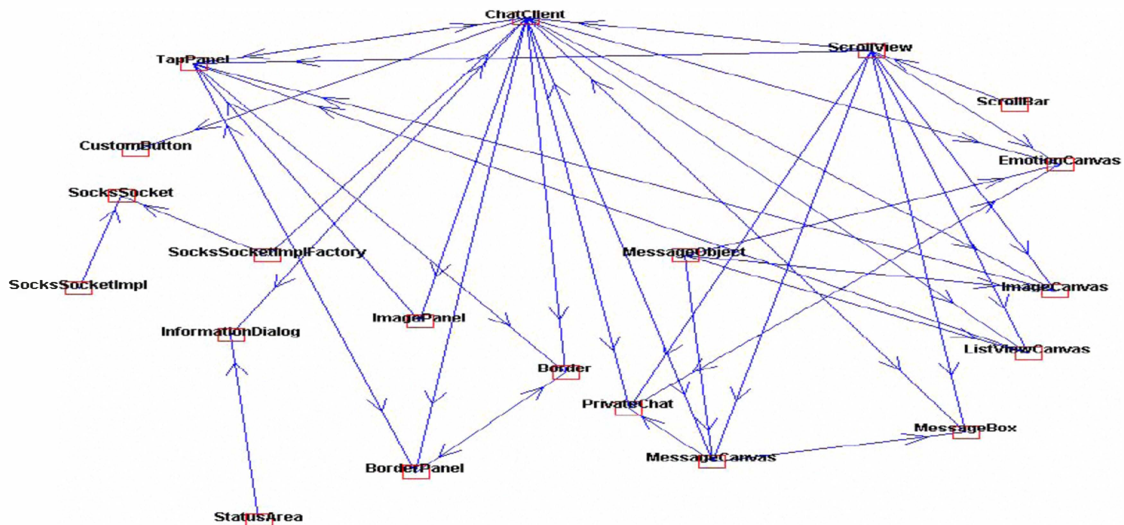

Figure 2. Graphical Representation of the relationship among the User Defined Classes of software 1 (after adding a new module)

TABLE I. PRINCIPAL COMPONENTS FOR FINDING THE MOST EFFECTIVE CLASS OF SOFTWARE 1

Principal component #	Eigen Vector	Eigen Value
1	**-0.2190** **-0.0989** 0.5251 0.5679 0.2232 **-0.0705** 0.1037 0.0043 **-0.0430** 0.016 **-0.0093** 0.0019 0.0167 0.1911 **-0.1806** 0.3981 **-0.0881** 0.1068 0.0081 0.1707	0.9713
2	0.9157 **-0.0559** 0.1064 0.1236 0.0472 0.0021 0.0214 0.0033 **-0.0089** 0.0517 0.0505 0.0059 **-0.0289** **-0.1707** **-0.1957** 0.1786 0.0640 **-0.1361** **-0.0267** 0.0150	0.2244
3	0.0874 0.5140 0.5518 **-0.3007** **-0.1698** 0.0262 0.0652 **-0.0109** **-0.0120** 0.0457 **-0.0055** **-0.1150** 0.2353 0.1326 0.0021 **-0.2258** **-0.0821** 0.0315 **-0.3285** 0.2104	0.0600

systems. Nevertheless, developers try to understand unknown object oriented systems by analyzing the source code to recover the architecture of the system, which is a hard task since the dependencies between the classes cannot be recovered well enough. However when a software of Object Oriented System undergoes the development process, the designer should be concern about the development cost that can be measured with respect to some quality metric of software such as Coupling. In this paper, an approach of detecting design patterns from Java source code is presented and the approach continues with the analysis of source codes of software for selecting the most effective i.e. the highly coupled class. This helps the designer to take the decision that how and where a new module can be added in an efficient manner by keeping the coupling value as minimum as possible and by ensuring the reduction of development cost. We believe, the knowledge about design patterns using the design tool can help developers to understand the underlying architecture faster.

REFERENCES

[1] W. AI-Ahmad, “Object-Oriented Design Patterns for Detailed Design,” Journal of Object Technology, vol. 5 No. 2, pp. 155-169, 2006.

[2] W. Pree, “Meta Patterns-A means for capturing the essentials of reusable object-oriented design”, ECOOP’94, LNCS 821, pp. 150-162. 1994.

[3] S.R. Chidamber and C.F. Kemerer, “A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering 20 (6), 476 493, 1994.

[4] P. Coad and E. Yourdon, “Object-Oriented Analysis”, prentice-Hall, second edition 1991.

[5] W. Stevens, G. Myers, and L. Constantine, “Structured Design”, IBM Systems Journal, 13 (2), 115-139, 1974.

[6] L. Briand, S. Morasca, and V. Basili, “Property-Based Software Engineering Measurement”, IEEE Transactions of Software Engineering, 22 (1), 1996, pp. 68-86.

[7] L. Briand, J. Daly, J. and Wust, “A Unified Framework for Coupling Measurement in Object Oriented Systems”, Technical Report ISERN 96-14, 1996.

[8] K. M. Azharul Hasan and D. N. Batanov, “Measuring Coupling for Developing Object Oriented Systems”, In Porc. On ICT, pp. 325-330, 2003.

[9] J. Eder, G. Kappel, and M. Schrefl, “Coupling and Cohesion in Object-Oriented Systems”, Technical Report, University of Klagenfurt, 1994.

[10] M. Hitz, and B. Montazeri, “Measuring Coupling and Cohesion in Object-Oriented systems”, In Proc. Int. Symposium on Applied Corporate Computing, 1995.

[11] P.C. Wong, R.D. Bergeron, “Multivariate visualization using metric scaling”. Proc., Visualization 97, Phoenix, October 1997, pp. 111-118.

[12] Gamma E. et al, “Design Patterns: Elements of Reusable Object-Oriented software”, Addison Wesley, 1995.