Finding the Design Pattern from the Source Code for Developing Reusable Object Oriented Software

Department of Computer Science and Engineering, Khulna University of Engineering and Technology, Khulna 9203, Bangladesh

***Abstract* — One of the principal goals of object-oriented software is to improve the reusability of software components. In this paper the identification of design patterns from source code is described to improve the reusability of software. The methodology for parsing scheme to recover the design pattern from the source code is described by defining interactions between objects. Three industrial softwares are use in case study to verify the methodologies and sufficient experimental results are presented.**

***Index Terms* — Coupling, design methodology, design pattern, software metrics, software reusability.**

I. INTRODUCTION

Object-oriented (OO) system development is gaining wide attention both in research environment and in industry. It is difficult for the software engineers to produce quality software in a short time as the customers demand. 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 the 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. Unfortunately, the benefits associated technology has their price [3]. Poor documentation of the code makes studying and understanding details painful and time consuming for reusing that software. Abstracting design details from the source code help to understand the implementation details reuse it in efficient manner. Sometimes it becomes even impossible to understand particular document without the design details. In this paper a methodology is proposed to find the design pattern of the software from the source code. This would provide several goals in software construction leading to better values for external attributes such as maintainability, reusability, and reliability. To find the design pattern it important to know the interactions between the internal components of the software. Different interactions are pointed out and described to get the interactions.

In OO paradigm coupling describes the interdependency between methods and between object classes, respectively [4]–[5] . 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" . Braind et al. defined some properties to be satisfied by the coupling measure for empirical validation. 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. As the polymorphic method invocation is determined by run time, the coupling on this method belongs to dynamic coupling [6]. The Coupling between Object classes (CBO) as "CBO for a class is a count of the number of other classes to which it is coupled". CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another.

Eder et al. identify three different types of relationships [7]. These relationships, interaction relationships between methods, component relationships between classes, and inheritance between classes derive the different dimensions of coupling. Hitz and Montazeri [8] 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. An approach to measure coupling in object-based systems [9] such as those implemented in C++ by expanding it to include inheritance and friendship relations between classes. This framework concentrates on coupling as caused by interactions that occur between classes.

In this paper interaction for coupling measure for object oriented system design is defined based on different interactions of the classes of the system. And using these interactions a parser is developed named "*Design*

978-1-4244-4457-1/09/$25.00 ©2009 IEEE

Analyzer" to find the design pattern of the program. An analysis was performed using the *Design Analyzer* on some software that are developed in Object Oriented Programming paradigm. In this paper three case study softwares are described to show how *Design Analyzer* works. The selected softwares were 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 descriptions of these softwares are 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 of contains 19 user defined classes and the Server side contains 4 user defined classes.

Software 2-Com Chat: This is also chatting software, which sends and receives data through the communication port of a personal computer. The software uses Java Communication Application Programming Interface (API). The main purpose of this software is to simulate the seven layers of the OSI Model. The software implements Broadcasting, CheckSum and Routing techniques. This software contains 10 user defined classes.

Software 3-Admission Test Management System: This is mainly database software which can be used for examination management of any university by storing information of the applicants, information of allowed candidate for test, merit list of selected candidates, waiting list and so on. This software contains 13 user defined classes.

II. Finding the Design Pattern for Java Based object Oriented System

Before we go any further, it is imperative to first discuss the concept of a design pattern. Gamma et al. 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. But beyond a description of the problem and its solution, software developers need deeper understanding to tailor the solution to their variant of the problem. Hence a design pattern also explains the applicability, trade-offs, and consequences of the solution. It gives the rationale behind the solution, not just a pat answer. In this paper the proposed *Design Analyzer* finds the interactions, roles, collaborations and relationships of the software in a graphical format.

In the subsequent sub sections the interactions between objects is found out for Java based programs. In Section 3 these interactions are used to develop the algorithm for *Design Analyzer*. The term object is used to mean a module through out the paper.

A. Types of Inter-module Interactions That Occur in Java

There different ways that are commonly found for interactions between the classes in a Java based object-oriented software are described in this section. 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:

a) Operation-Operation (O-O) interaction
b) Class-Class interaction (C-C) interaction

Operation-Operation (O-O) interaction: The first two categories (relationship through return type and argument passing) above are included in O-O interaction. Hence we defined O-O interaction as follows

Definition 1: (Operation-Operation Interaction): The Operation-Operation interaction (O-O) is defined as the interaction between two operations of two or more different objects or classes. Let O_C 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 O_C or class D is the return type of O_C.

Class-Class interaction (C-C) interaction:
The last two categories above (relationship through object declaration and inheritance) are included in C-C interaction. Hence we defined C-C interaction as follows

Definition 2: (Class-Class 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 a C-C interaction between the classes C and D, if an object O_d of D is declared inside class C or D is derived from class C through inheritance.

B. The Parsing Scheme to Find the Design Pattern

In this section the two types of interactions defined in Section 2 (C-C and O-O) is parsed in java based programs to develop the *Design Analyzer*. Although the examples are in Java but it can easily be extended in any object oriented language.

Parsing for the O-O Interaction

Relation through Argument Passing

Before proceeding we have to know the format of how classes can be inter-related through argument passing in functions. Let's consider two classes Class A and Class B. Let a is the object of Class A which is declared in the scope of Class B. In java this can happen in one of the following fashion:

(1) Class B{
access-modifier static A function-name (argument list)
}

(2) Class B{
access-modifier final A function-name (argument list)
}
(3) Class B{
access-modifier A function-name (argument list)
}

Here access-modifier sits for indicating public, private or protected and argument list represents variables of any data type. So from here we see that Class B is the container class because it contains a function that uses the object of another class as argument. During parsing the source codes of Class B if we find a statement like:
access-modifier static A function-name (argument list) or
access-modifier final A function-name (argument list) or
access-modifier A function-name (argument list);
then we can come to the decision that Class B is related to Class A through the object a and c as argument of the function function-name.

Relation through Return Type of function

Let's consider two classes Class A and Class B. Let a is the object of Class A which is declared in the scope of Class B. In java this can happen in one of the following fashion:

(4) Class B{
access-modifier static return-type function-name (A a, A c)
}

(5) Class B{
access-modifier final return-type function-name (A a, A c)
}

(6) Class B{
access-modifier return-type function-name (A a, A c)
}

So from here we see that Class B is the container class because it contains a function that uses the object of another class as argument. During parsing the source code of Class B if we find a statement like
access-modifier static return-type function-name (A a, A c) or
access-modifier final return-type function-name (A a, A c) or
access-modifier return-type function-name (A a, A c);
then we can come to the decision that Class B is related to Class A through the object a and c as argument of the function function-name.

Parsing for the C-C interaction

Relation through object declaration

Before proceeding we have to know the format of how classes can be inter-related through object declaration in Java. Let's consider two classes Class A and Class B. Let a is the object of Class A which is declared in the scope of Class B. In java this can happen in the following fashion.
(7) Class B {
A a = new A( );
}
So from here we see that Class B is the container class because it contains the object of another class. During parsing the source code of Class B if we find a statement like A a = new A ( ); then we can come to the decision that Class B is related to Class A through the declaration of the object a.

Relation through Inheritance

Let's consider two classes Class A and Class B. Let a is the object of Class A which is declared in the scope of Class B. In java this can happen in one of the following fashion:

(8) Class B extends A{
// body of Class B
}
(9) Class B implements A {
// body of Class B
}

So from here we see that Class B inherits Class A. During parsing the source code of Class B if we find a statement like Class B implements A or Class B extends A then we can come to the decision that Class B is related to Class A through inheritance.

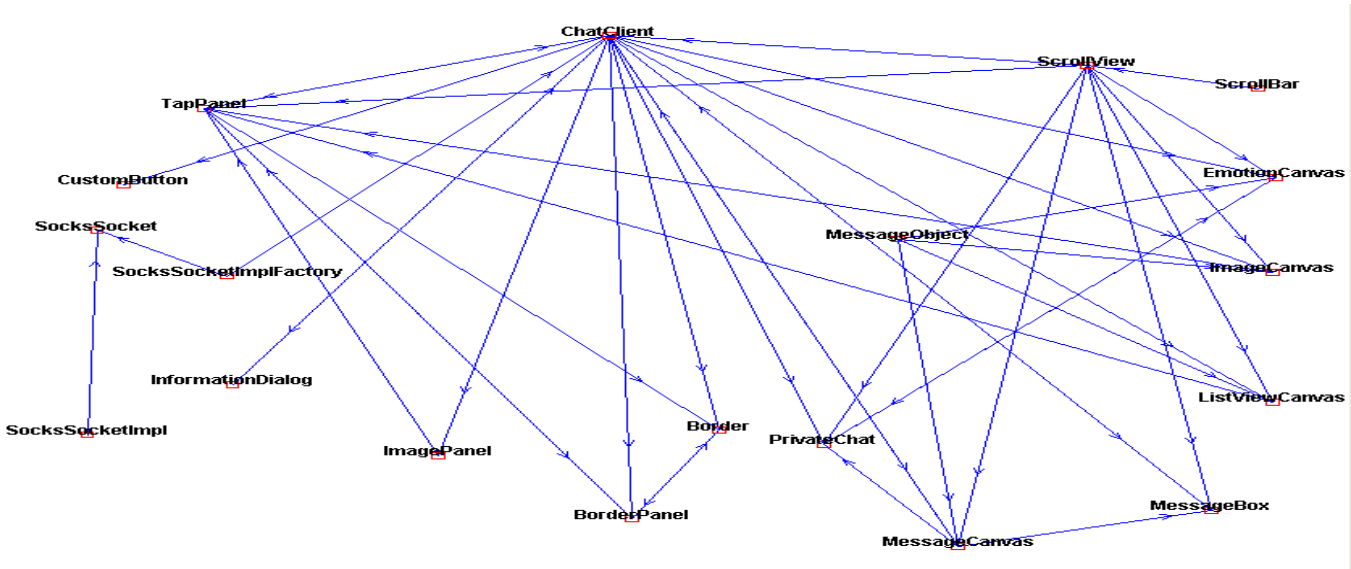
III. Analysis of Design Pattern Using design Analyzer

We have developed tool using java for analyzing the source code of an object oriented system to get the design pattern of the system. We named it *Design Analyzer*. Through out the paper where we used the word *Design Analyzer* we mean the developed software. The *Design Analyzer* implements the parsing scheme described in Section 2. The input of the system is the source code of an Object Oriented program and output is the graphical representation (See Fig. 1) of the design pattern of the software. In the design pattern each node represents a class of the system and there is an edge 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.

Fig.1 shows the design pattern found using the *Design Analyzer* for software 1 described in Section 1 from the figure we can see that there are 19 user defined classes in the client part of 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. Fig. 2 shows the design pattern of software 2. Fig. 3 shows the design pattern of software 3. From the figure we see that the software 3 is poorly designed. Almost all the classes are coupled with one class namely *Admission*. If this class fails or has a bug then the whole software will work poorly.

In conclusion 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.

Fig. 1. Graphical representation of the relationship among the user-defined classes of client part of Software 1.



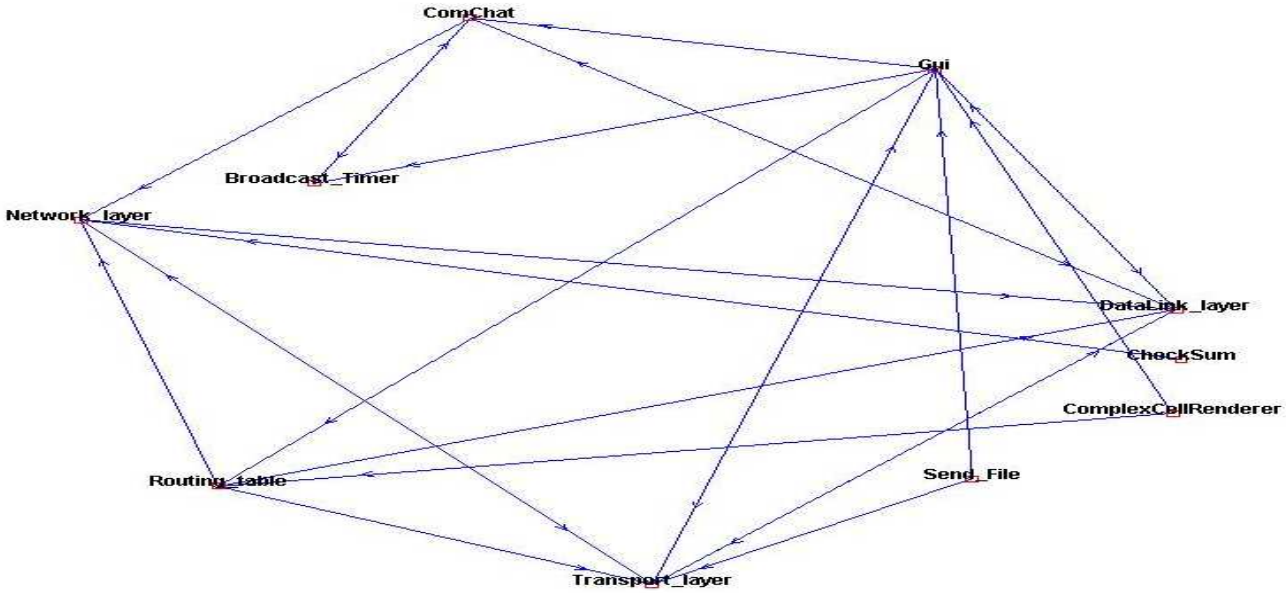


Fig. 2. Graphical representation of the relationship among the user-defined classes of Software 2

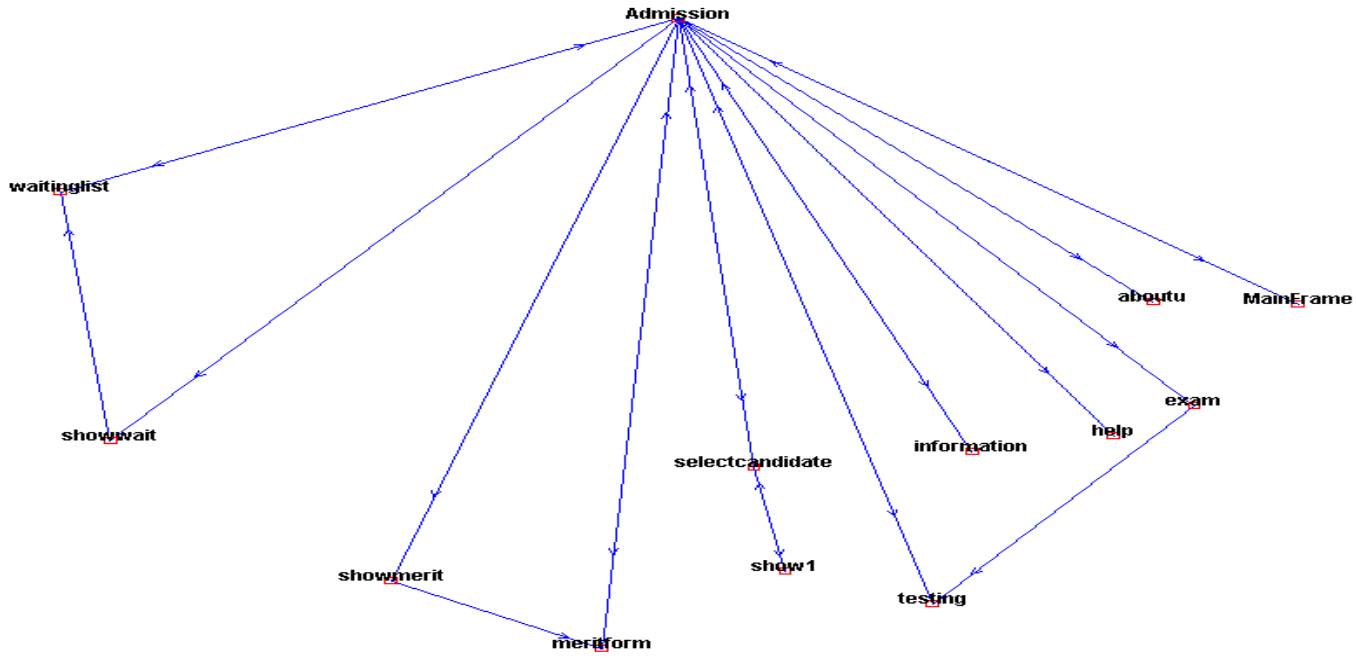


Fig. 3. Graphical representation of the relationship among the user-defined classes of Software 3.

A. Adding New Module

From this 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. Here in this we show one example of software 1 (Section 1) where one new module is added to make the existing software reusable. From the graphical analysis of Software 1 the decision that can be taken is: a new module can be added beside the classes which are not highly coupled such as: Border, ImagePanel, InformationDialog, ListViewCanvus, ScrollBar. After adding a new module StatusArea beside the class InformationDialog, the graphical representation that is obtained is shown in Fig. 4.

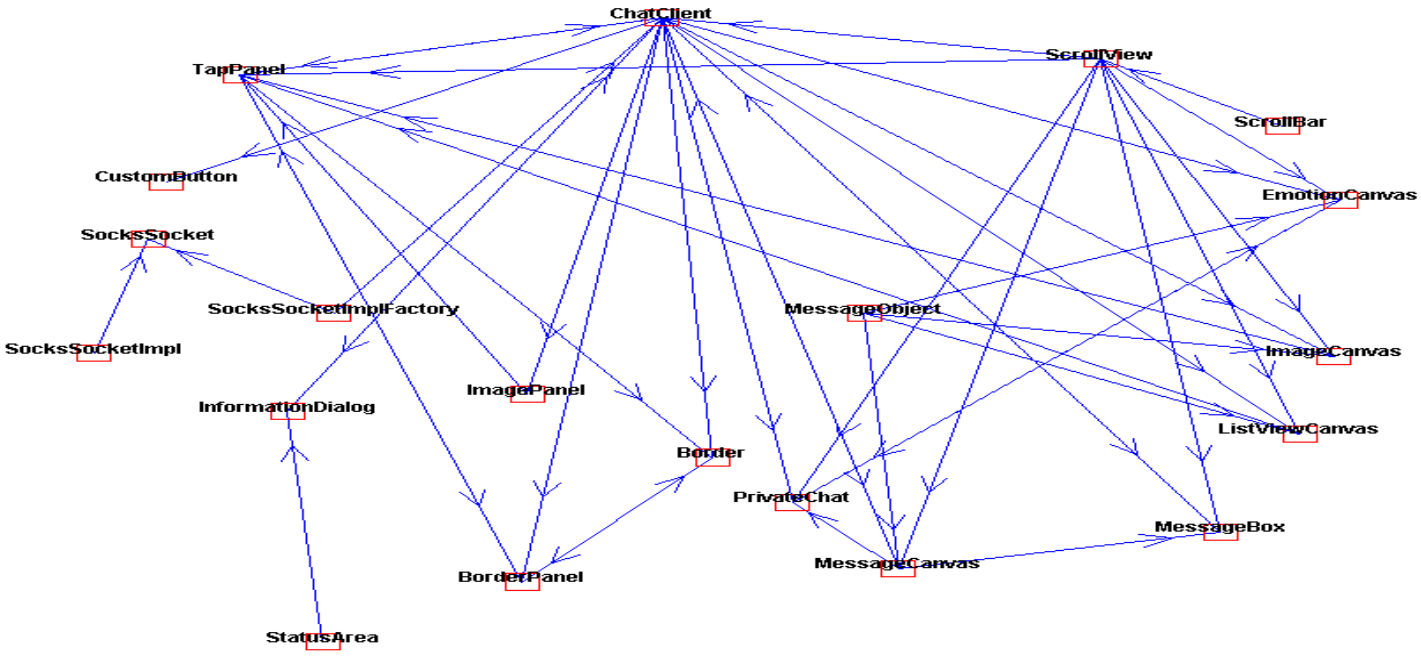


Fig. 4. Graphical view of the inter class relationship for Software 1 after adding a new module.

IV. Conclusion

Identification of design patterns from source code is one of the most promising methods for improving software maintainability and reusing design experience. 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 softwares for selecting the most effective component and the highly coupled class. Here a parser named *Design Analyzer* has been introduced which generates the graphical representation of the inter-class relationship and three software have been analyzed by this software for case study and after analysis, the highly coupled class for software 1 has been detected. 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 as minimum as possible and by ensuring the reduction of development cost. After detecting the highly coupled class a new module has been added in Software 1.

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,* pp. 150-162, 1994.

[3] W. Pree, H. Sikora, "Design patterns for object-oriented software development (tutorial)", *International Conference on Software Engineering, Proceedings of the 19th international conference on Software engineering,* pp. 663 – 664, 1997.

[4] S. R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering* 20 (6), pp. 476-493, 1994.

[5] P. Coad and E. Yourdon, "Object-Oriented Analysis", *prentice-Hall*, second edition, 1991.

[6] K. M. Azharul Hasan and D. N. Batanov, "Measuring Coupling for Developing Object-Oriented Systems", *In Porc. ICT*, pp. 325-330, 2003.

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

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

[9] D. N. Batanov and Somjit Arch-int, "Business Objects and Components for Web-based Information Systems Development", *Proceedings of IRMA'2002 Seattle, 2002.*