Computer Science Journal of Moldova, vol.**21**, no.2(62), 2013

Design and code coupling assessment based on defects prediction. Part 1

Arwa Abu Asad, Izzat Alsmadi

Abstract

The article discusses an application of code metrics at object-oriented software design. Code metrics give an additional method to avoid errors except the obvious ones like thorough requirements, design, programming, testing, and consumer's feedback. Software metrics try to collect values and measurements from the software and predict possible current or future problems. This paper includes the development, analysis and evaluation of several software code metrics. The paper also investigates how could coupling metrics be utilized as early indicators of fault proneness. A tool is developed to parse through code projects and automatically collect those metrics. A case study of Scarab project is selected to evaluate coupling metrics ability to predict fault proneness. Results showed that the value of the evaluated metrics can vary in terms of their ability to judge the software design and fault proneness. Results showed also that CBO, RFC, MPC and ICP have more correlation with reported bugs in comparison with other collected and evaluated coupling metrics.

Keywords: Design metrics, Object-Oriented Designs, Coupling metrics, software faults.

1 Introduction

Software quality is affected significantly by design phase problems. Typically, high quality software will be the outcome of software with a good design. Therefore, designers should have the vision for quality of software in earlier stage of design process by making software easier to

understand and change later as well. It is largely acknowledged that software design quality attributes can be directly related to software code or implementation attributes. Thus, it is important to recognize measuring reliability, efficiency, modularity and quality in early phases prior to implementation of the code. How could we control design to produce robust, maintainable and reusable software? Many metrics were proposed to measure quality of design identifying potential problems as early as possible. Metrics provide an important assistance to developers for design assessment.

Developers must recognize the factors that will improve their design in terms of understandability and future change. One of the important design quality factors is the coupling or type and nature of connection between components that may impact the way those components interact or will respond to future modifications.

In this paper several design coupling metrics proposed by Briand et al (1999) [1–3] are assessed using a large set of open source code projects. One of the major design goals is to minimize coupling. Therefore, it is important to calculate this attribute before the implementation phase. Goals or objectives of this research can be summarized as:

- To develop a tool to collect several design or object oriented coupling metrics.
- To find the relationship between code and design coupling metrics.
- To empirically verify the validity of coupling metrics by means of a comparative evaluation of open source codes.
- To assess the correlation between coupling metrics and fault proneness.

The remainder of this paper is organized as follows: Overview of related work is described in Section 2. A measurement coupling model is discussed in Section 3. Metrics calculation is presented in Section

4. In Section 5 we presented coupling correlation with fault proneness and Section 6 presents the conclusion.

We divided the paper into two parts due to size limitation. In the first part, we will describe all metrics, their equation and how can they be measured with small examples. The second part will mainly focus on the introduction, evaluation and analysis of the case study that is conducted as part of this research. The case study includes the collection and analysis of a large set of open source code applications. After the automatic collection of all metrics described in the first part, the second part includes an evaluation and assessment stage of the collected metrics using statistical analysis to find possible values or predictions based on the collected metrics and data.

2 Related Work

Metrics related to each other. They provide overlapping quality information. Complexity and coupling are related to each other. Many researchers studied the relation between them. Chidamber and Kemerer (1994) [4] used group of coupling metrics to estimate the complexity of OO systems such as RFC, CBO DIT and NOC. Fenton and Pfleeger (1997) [6] proved that software metrics such as complexity, coupling, and cohesion (CCC) could be used to evaluate software quality. Badri et al. (2009) proposed multi-dimensional metrics that captures coupling between classes using different internal software attributes, such as complexity.

Yadav and Khan (2011) [11] proposed coupling complexity normalization metric that minimizes complexity by controlling coupling of software system. Misra et al. (2011) [8] proposed a new metric, Cognitive code complexity (CCC), which is based on inheritance and basic control structure. They observed that CCC is related to RFC, DIT, NOC, CBO and WMC.

In addition, they showed that CCC could estimate OO design quality.

Stevens et al. (1974) [10] defined the first definition for coupling metrics in procedural programming. Myers (1978) [12] defined six met-

rics for procedural coupling among pairs of modules, which are content, common, external, control, stamp and data coupling. Offutt et al. (1993) [9] extended the previous level of coupling by introducing the no-coupling level, and redefining it; so, it algorithmically quantified these metrics on C-program. Chidamber and Kemerer (1994) [4] proposed the first formal definition of coupling between classes and defining number of metrics related to coupling that is CBO and RFC.

Eder et al. (1994) [5] redefined coupling types of procedural programming for object-oriented systems, which are the following:

- Content Coupling occurs when one class has a direct access to private method values (i.e. instance variables) of another class where relation between classes allows such access, like friend classes in C++ or internal in CSharp. Some special cases like "Jump" statements represent also examples of content coupling. "Goto" and "Jump" keywords are used in low level and structural languages and restricted in the object oriented one.
- Common coupling occurs when one class has a public access for instance variables or if such variables are accessed through an inheritance relation.
- Control coupling occurs when one module controls the execution of another module under certain conditions.
- Stamp coupling occurs in message passing scenarios where one parameter type of the complete message is of type object (i.e. not a primitive data type).
- Data coupling occurs in message passing where all parameters are of primitive data types.

In addition, they identified three types of relationships in object-oriented systems:

- Interaction relationships between methods,
- Component relationships, and

- Inheritance relationships between classes.

3 Coupling measurement model

3.1 Terminology

This section defines the terminologies and the abbreviations that are used in the later sections. Table 2 shows the term and the definition adapted for it in this paper as well as the abbreviation of terms.

3.2 Coupling Metrics model

Coupling Between objects (CBO)

The original definition of CBO excludes inheritance relations. However, this definition is revised to include inheritance relationship. Coupling between objects metric is defined for Instance attributes, and for method calls including inheritance relationship. The approach of Briand et al. [1-3] deals with various method invocations with no consideration for the type of call. However, in this paper polymorphism call – due to interface method calls – is not considered as coupling.

There are different types of method calls:

- Direct calls: when method calls method in another class – it is considered as coupling.
- Polymorphism calls: when method calls method of interface – it is not considered as coupling between the class and the class that implements the interface.
- Inherited method from parent class: when inherited method calls from child – it is considered as coupling with parent class.
- Overriding method calls: they contribute to coupling between all classes that define the method since metrics collection is statically collected. Finding out exactly the class of overriding method requires execution of the code.

Table 1. Terminology Abbreviation

Term	Definition	Abbreviation
Defined methods	Methods declared within class C	M_{def}
inherited methods	Methods declared within parent class and inherited and (not overridden) within child class C	M_{inh}
Polymorphic method	Methods defined within an interface	M_{poly}
new methods	Methods declared within class C that do not override inherited ones	M_{new}
Overriding methods	Methods declared within class C that override (redefine) inherited ones	M_{over}
invoked methods	Methods that can be invoked in association with class C	M_{inv}
External calls	Invocations to a method defined in other classes	$Call_{ex}$
Internal calls	Invocations to a method in the same class	$Call_{int}$
polymorphic calls	invocations to a method defined in an interface	$call_{poly}$
inherited calls	Invocations to a method defined in parent class through object of child class	$call_{inh}$
defined attributes	attributes declared within class C	$Attr_{def}$
overriding attributes	attributes declared within class C that override (redefine) inherited ones	$Attr_{over}$
Inherited attributes	attributes inherited (and not overridden) in class C	$Attr_{inh}$
used attributes	attributes that can be manipulated in association with class C (those external public attributes from other classes used in method of class C)	$Attr_{use}$
instance attributes	attributes of type class	$Attr_{inst}$
Instance parameter	Parameter of type class	arg_{ins}

Calls are counted in both directions. Therefore, CBO of a class C is the number of other classes that class C references and classes that reference class C. Since CBO counts each class only one time, so if C references C‘ and C‘ references C, then C‘ will be counted one time, hence there is no need for counting in both directions.

The relationship between child and parent was not considered in Chidamber & Kemerer definition [4]. Nevertheless, this definition had been revised to include inheritance coupling by Briand et al. framework.

Instance variable is determined as:

- Instance attribute that is declared in C. Inherited, polymorphism, and overriding attributes are not discussed by Briand et al. [1-3]. In this framework, they will be dealt with like the corresponding method.

All these points are considered in the parsing and analysis process. As such, the calculation of these metrics has been made directly from the database using SQL.

CBO equation:

$$CBO = \sum unique(Ci),$$

$$where\ Ci = C - c \mid c\ invoke\ M(C)\ or\ c\ defined\ Attr_{inst}$$

Pseudo code for select statement:

```
Result1= Select used_class_num from call_table where Call_int = 'false'
and call_poly = 'false' and current_class_num = ?
Result2= Select used_class_num from attribute_table
where Attr_inst = 'true' and current_class_num = ?
CBO = count (distinct Result1, Result2)
```

Coupling between objects metric is defined as the number of classes to which a class is coupled. However, for comparison reason in this paper we defined average of CBO as follows:

$$ACBO = \frac{Number\ of\ classes\ to\ which\ a\ class\ is\ coupled}{Total\ number\ of\ classes}$$

Response for a Class (RFC)

The response for a class is the set of methods that have the possibility to be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set and number of calls in that class. There are two versions of RFC, which are:

RFC = $M + R$ (First-setp measure)

RFC' = $M + R'$ (Full measure),

where M = represents the number of methods in the class. Counted class methods are:

- Declared methods.
- Inherited methods.

R = represents the number of remote methods that are directly called by methods of the class. Here are the counted calls:

- Calls to methods in other classes. Internal calls of class methods are not counted.
- Calls to library methods are not counted.
- Polymorphism calls through interface are not counted because they are not considered as coupling calls.
- Calls to inherited methods through child classes are considered as in a coupling relation with the parent class not the child one.
- Each called method is only counted once no matter how many times the method is called.

R' = represents the number of remote methods called recursively through the entire call tree.

RFC represents the first level or direct calls outside of the class. RFC may also consider recursive counts and all potential responses set recursively until entire call tree is included. On the other hand, polymorphic method calls that include all possible remote methods executed are included either in R or R'. A given method is counted only once in R (and R') even if it is executed by several methods. RFC' that considers recursive calls is not considered in our developed tool.

Pseudo code for select statement:

```
M = select count (method_num) from method_table
where current_class_num = ?
R (First-step measure) = select count (DISTINCT
invoked_method_num) from call_table
where Call_int = 'false' and call_poly = 'false' and current_class_num = ?
RFC = M + R (First-step measure)
```

Message passing coupling (MPC)

MPC is originally defined as : The number of send statements defined in a class. MPC counts the number of method calls defined in methods of a class to methods in other classes excluding inheritance relation. MPC counts only method calls to non-descendent or ancestor classes. On the other side, Briand et al. [1-3] redefined MPC as the number of method invocations in a class including invocations to methods in inheritance relation. Since one of the purposes of this paper is to investigate Briand framework, the redefined MPC is adapted.

```
MPC = select count (*) from call_table where Call_int= 'false' and
callpoly = "false"
and current_class_num = ?
```

Class Data Abstraction Coupling(DAC)

DAC counts the number of instantiations of other classes within the given class or the number of attributes of type class. This type of coupling is not caused by inheritance. Therefore, if a class has an

attribute variable that is an instantiation (object) of another class, this is considered as data abstraction coupling.

$$\sum \text{ attr } \mid \text{ type(attr)} \in Ci - c$$

This formula is translated into SQL for DAC‘:

DAC‘= select count (DISTINCT Attr_{inst}.class_num)
from attribute_table where $\text{Attr}_{inst=}$ ’true’ current_class_num =?

Information-flow-based coupling (ICP)

The original definition of ICP counts for a method m of a class c, the number of polymorphically invoked methods of other classes, weighted by the number of parameters of the invoked methods (Lee et al., 1995) [7]. We used Briand definition for ICP that it measures the amount of information flow to and out from the class via parameters through method invocation as the number of method invocations in a class. This is weighted by the number of parameters of the invoked methods, where the weight is number of parameters plus 1.

The equation is then calculated as:

$$ICP = \frac{\sum Call}{\sum Call + \sum par(m_{inv}) + 1}$$

This formula is translated into SQL as:

$$\sum Call$$

Result1 = ”select count (*) from call_table where Call_{int} = ’false’ and Call_{poly} = ’false’ and current_class_num = ?

Result2 = ”select count (argument_num) from call, method, argument
where call.invokedmethod_num = method.method_num and
method.method_num = argument.method_num
and current_class_num = ?

So,

ICP = Result1 / (Result1+Result2+1)

Briand Suite Measures [1-3]

Some of measures defined in this suite are based on friendship relation, which is dependent on C++ language terminology. To generalize the issue to other programming languages, we considered only common aspects among the different languages evaluated: Java, CSharp and C++. We considered only measurements that rely on inheritance relationship. The meaning for these measures are explained in Section 3.

Class-attribute interaction (CA)

Class-attribute interaction measures:

$$ACAIC(c) = \sum_{d \in Ancestors(c)} CA(c,d)$$

ACAIC= select Attr_{inst}.class_num from attribute_table where Attr_{inst} = 'true' and current_class_num =? and Attr_{inst}.class_num = ancestors_class

$$DCAIC(c) = \sum_{d \in Descendents(c)} CA(d,c)$$

DCAEC = select Attr_{inst}.class_num, count (*) from attribute_table where Attr_{inst} = 'true' and current_class_num =? And Attr_{inst}.class_num = descendent_class group by Attr_{inst}.class_num

Class-method interaction (CM)

Class-method interaction measures:

$$ACMIC(c) = \sum_{d \in Ancestors(c)} CM(c,d)$$

$$DCMEC(c) = \sum_{d \in Descendents(c)} CM(d, c)$$

ACMIC = select Parameter_table.class_num, count (arg_num)
from Parameter_table, method_table
where Parameter_table.method_num = method_table.method_num
and arg_{ins} = 'true' and method_table.modifiers NOT LIKE 'override'
and method_table.class_num =? Parameter_table.class_num
€ancestors_class GROUP BY Parameter_table.class_num

DCMEC = select method.class_num, count (arg_num) from
Parameter_table, method_table where:
Parameter_table.method_num = method_table.method_num
and arg_{ins} = 'true' and method_table.modifiers NOT LIKE 'override'
and method_table.class_num <>? and Parameter_table.class_num =?
and method_table.class_num €descendent_class
GROUP BY method_table.class_num

Method-method interaction (MM)

Method-method interaction measure (newly defined or overrided). The pseudo code for the two measures is summarized as follows:

$$AMMIC(c) = \sum_{d \in Ancestors(c)} MM(c, d)$$

AMMIC = select called_class_num, count (invoked_method_num)
from call_table where current_class_num =? And called
class €ancestors_class GROUP BY called_class_num

$$DMMEC(c) = \sum_{d \in Descendents(c)} MM(d, c)$$

DMMEC = select current_class_num, count (invoked_method_num)
from call_table Where used_class_num = ? and current_class €
descendent_class GROUP BY current_class_num

4 Metrics Demonstration

4.1 Metrics Demonstration

Previously, coupling metrics were discussed in theory. This section provides examples of CSharp code used to help in clarifying the coupling metrics calculation. The following CSharp code has four classes: A, B, C, and D, where coupling relation is shown in the source code.

Class A {
C obj1;
D obj2;
Public methoda1 () {obj1.methodc1 () ;}
Virtual Public methoda2(l obj3)
{obj1.methodc2 (obj3);
Obj2.methodd1 () ;} }
	Class C {
	D obj1;
	Public methodc1(){
	obj1.methodd1();}
	Public methodc2(C
	obj2){obj1.methodd1();
	ebj1.mothodd2();}}
	}

Class B: A {
Public C obj1;
C obj2;
A obj5;
Public methodb1 (A cbj6)
{obj1.methodc1 () ;}
Public methodb2(c obj3)
{obj1.methodc2 (obj3);
Obj2.methodc1 () ;
methoda1(); }
Override methoda2(c obj3) {D obj4;
Obj4.methodd1();
Obj3.methodc2(obj3);} }
	Class D {
	B obj1;
	A obj2;
	Public methodd1(){
	obj1.methodb2(t obj3);}
	Public
	methodd2(){obj1.methodb1();
	Obj1.methoda1();}
	}

CBO:

- Class A is coupled to two classes C and D since it has two attributes of type class C and D as well as invocations to methods: methodc1 and methodc2 in C, and methodd1 in D. Thus, CBO for A is 2.
- Class B is coupled to two classes A, and C since it has attributes of type C, invocations to methods: methodc1, methodc2 and

methodd1 in addition to an invocation to methoda1 of class A. Thus, CBO for B is 3.

- Class C is coupled to class D since it has an attribute of type D and invocations to methods methodd1 and methodd2. Thus, CBO for C is 1.
- Class D is coupled to three classes: A, B and C since it has two attributes of type class: A and B as well as invocations to methods: methodb1, methodb2 and methoda1. Thus, CBO for D is 2. It should be noticed that invocation to method methoda1 through class B is considered as coupling with A because coupling will be with the class that declares the method, where inherited methods will be considered as coupled through parent class.

RFC:

- Class A: The class has two methods: methoda1 and methoda2. Methoda1 has one call to methodc1 and methoda2 has two calls to methodc2 and methodd1. Thus, RFC = 2+ 3 = 5
- Class B: The class has three methods: methodb1, methodb2 and methodb3. It has also one inherited method methoda1. Methodb1 has one call to methodc1. Methodb2 has three calls to: methodc2, methodc1 and methoda1. Methoda2 has two calls to: methodd1 and methodc2. Methoda1 has one call to methodc1. Thus, RFC = 3+ 6 = 9.
- Class C: The class has two methods: methodc1 and methodc2. Methodc1 has one call to methodd1. Methodc2 has two calls to: methodd1 and methodd2. Thus, RFC = 2 + 3=5.
- Class D: The class has two methods: methodd1 and methodd2. Methodd1 has one call to methodb2. Methodd2 has two calls to: methodb1 and methoda1. Thus, RFC = 2 + 3=5.

MPC:

- Class A: The class has three method calls to: methodc1, methodc2 and methodd1. Thus, MPC = 3.
- Class B: The class has five calls to: methodc1, methodc2, methodc1, methodd1, methoda1 and methodc2. Thus, MPC =6.
- Class C: The class has three calls to: methodd1, methodd1 and methodd2. Thus, MPC = 3.
- Class D: The class has three calls to: methodb2, methodb1 and methoda1. Thus, MPC = 3.

DAC:

- Class A has 2 attributes of type class. Thus, DAC= 2 and DAC1=2.
- Class B has 2 attributes of type class. Thus, DAC= 2 and DAC1=1.
- Class C has 1 attribute of type class. Thus, DAC= 1 and DAC1=1.
- Class D has 2 attributes of type class. Thus, DAC= 2 and DAC1=2.

ICP:

- Class A has three calls with methods. One method has one parameter and the others with no parameters. Thus, ICP = (3/5)=0.6.
- Class B has six calls with 2 parameters of method methodc2 and the rest has no parameters. Thus, ICP = (6/9).
- Class C has three calls with no parameters for the three methods. Thus, ICP = (3/4).

- Class D has three calls with one parameter of method methodc2 and the rest has no parameters. Thus, ICP = (3/5).

ACAIC:

- Class A has no ancestors. Thus, ACAIC = 0.
- Class B has 1 attribute of parent A type. Thus, ACAIC = 1.
- Class C has no ancestors. Thus, ACAIC = 0.
- Class D has no ancestors. Thus, ACAIC = 0.

DCAEC:

- Class A has 1 attribute of type A in its descendent B. Thus, DCAEC = 1.
- Class B has no descendents. Thus, DCAEC =0.
- Class C has no descendents. Thus, DCAEC =0.
- Class D has no descendents. Thus, DCAEC =0.

ACMIC:

- Class A has no ancestors. Thus, ACMIC = 0.
- Class B has 1 parameter of parent A type in the method methodb1 (A cbj6). Thus, ACMIC = 1.
- Class C has no ancestors. Thus, ACMIC = 0.
- Class D has no ancestors. Thus, ACMIC = 0.

DCMEC:

- Class A has 1 parameter of its type A in method methodb1 (A cbj6) in its descendent B. Thus, DCMEC = 1.
- Class B has no descendents. Thus, DCMEC =0.
- Class C has no descendents. Thus, DCMEC =0.
- Class D has no descendents. Thus, DCMEC =0.

AMMIC:

- Class A has no ancestors. Thus, AMMIC = 0.
- Class B has 1 method invocation of its parent A through methoda1. Thus, AMMIC = 1.
- Class C has no ancestors. Thus, AMMIC = 0.
- Class D has no ancestors. Thus, AMMIC = 0.

DMMEC:

- Class A has 1 method invocation of its methoda1 in its descendent B. Thus, DMMEC = 1.
- Class B has no descendents. Thus, DMMEC =0.
- Class C has no descendents. Thus, DMMEC =0.
- Class D has no descendents. Thus, DMMEC =0.

As it was described earlier, a tool is developed to automate the collection and calculation of those metrics from source codes. Initially, we conducted a manual evaluation for small source codes to make sure that metrics are calculated correctly in the tool in comparison with manual calculation or verification.

4.2 Discussion of Metrics Demonstration

The tool is developed to carry out an evaluation on evaluated coupling metrics, which are: CBO, RFC, MPC, DAC, DAC1, ICP, COF, ACAIC, DCAEC, ACMIC, DCMEC, AMMIC, and DMMEC. These coupling metrics are used to assess the quality of design affecting many high level quality attributes such as: Reusability, maintainability, understandability, complexity, and testability. In most cases, tight coupling will have negative impact on these quality attributes and decrease: reusability, maintainability, understandability, and testability as well as increasing complexity.

High CBO indicates that the class is tightly coupled with other classes in the software, which complicates testing and modification, and limits reusability. The large number of RFC means that many methods are invoked in response to a message. As a result, testing and debugging of the class are then more complex and harder to understand so eventually making it a harder task for tester to pursue an error. As testing and debugging is complicated, quality is decreased and which in turn indicates bad design. MPC indicates also effect on: reusability, maintenance and testing effort. A larger MPC indicates high coupling between the subject class and other classes in the system which means that the class is difficult to change. Higher values of DAC and DAC‘ indicate complexity in data structures and classes of the code. High ICP indicates a high amount of information flow in the class which may complicate maintenance tasks. ACAIC, DCAEC, ACMIC, DCMEC, AMMIC, and DMMEC are indicators of the degrees of inheritance coupling, so high values in those metrics may complicate maintenance and reusability. The degree of coupling determines ability to make changes in design and code. In tightly coupled class, a change in one class may cause eventually several changes on other classes as well. Such ripple effect may go through all or most of the code or the design. Therefore, coupling metrics can play a major role in predicting quality of design showing some symptoms of serious issues.

Coupling metrics are then effective predictors for design quality. The tool is first validated manually by comparing tool results with

manual assessment of metrics. The tool is then used in a case study of several open source codes. Further details about the experimental case study and its analysis will be described in the second part of this paper.

5 Conclusion and Future Work

There are several possible characteristics in which software product quality can be evaluated. Part of quality assurance software metrics tools are used to automatically collect and assess the software quality through code and design. Coupling metrics are used to assess the software product in general and the design quality in particular. In this paper, we formulated, developed and evaluated several software coupling metrics. Coupling minimization improves the quality of design. Thus, coupling metrics could be utilized as early indicators of software quality. The coupling metrics that were evaluated in this paper include: CBO, RFC, MPC, DAC, DAC1, ICP, COF, ACAIC, DCAEC, ACMIC, DCMEC, AMMIC, and DMMEC. A program is developed to parse software source code and collect those metrics automatically. In this first part of the paper, we showed examples of all coupling metrics and how can they be calculated.

We evaluated the correctness of the tool in terms of measuring coupling metrics correctly through comparing the tools results with manual calculation of those metrics based on their defined formulas. In the next part of this paper, we will collect several open source codes. We will collect all coupling metrics described earlier from those source codes. We will then conduct several statistical and data mining analysis methods to show the value of the evaluated metrics specially in terms of software or design quality aspects such as bugs detection, maintainability, etc.

References

[1] L. Briand, J.W. Daly, J.K. Wust. *A unified framework for coupling measurement in object-oriented systems*, IEEE Transactions

on Software Engineering, 25 (1) (1999), pp. 91–121.

[2] L. Briand, P. Devanbu, W. Melo. *An investigation into coupling measures for C++*, *Proc. 19th Int'l Conf. Software Eng, ICSE*, (1997), pp. 412–421.

[3] L. Briand, J. Wust, J.W. Daly, D.V. Porter. *Exploring the relationships between design measures and software quality in object oriented systems'*, Journal of Systems and Software, 51(3), (2000), pp. 254–273.

[4] S. Chidamber, C. Kemerer. *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering, 20 (6), (1994), pp. 476–493.

[5] J. Eder, C. Kappel, M. Schrefl. *Coupling and Cohesion in Object-Oriented Systems*. Technical Report, Univ. of Klagenlurt, available at ftp://ftp.ifs.uni-inz.ac.at/pub/publications/1993/0293.ps.gz, (1994).

[6] N. Fenton, S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing Co., (1997), Boston, MA, USA.

[7] Y. Lee, B.S. Liang, S.F. Wu, F.J. Wang. *Measuring the coupling and cohesion of an object-oriented program based on information flow*, Proc. Int'l Conf. Software Quality, Maribor, Slovenia 12, (1995), pp. 81–90.

[8] S. Misra, I. Akman, M. Koyuncu. *An inheritance complexity metric for object-oriented code: A cognitive approach*, Indian Academy of Sciences, 36 (3), (2011), pp. 317–337.

[9] J. Offutt, M.J. Harrold, P. Kolte. *A software metric system for module coupling*, The Journal of Systems and Software, 20(3), (1993), pp. 295–308.

[10] W. Stevens, G. Myers, L. L. Constantine. *Structured design*, IBM Systems Journal, 13 (2), (1974), pp. 115–139.

[11] A. Yadav, R.A. Khan. *Coupling Complexity Normalization Metric-An Object Oriented Perspective*, International Journal of Information Technology and Knowledge Management, 4(2), (2011), pp. 501–509.

[12] G. J. Myers. *Composite Structured Design*, Van Nostrand Reinhold Co, New York, 1978.

Arwa Abu Asad, Izzat Alsmadi Received October 17, 2012

Arwa Abu Asad
Institution: Yarmouk University
Address: CIS department
E–mail: *arwa_abuasad@yahoo.com*

Izzat Alsmadi
Institution : Yarmouk University
Address : CIS department
E–mail: *ialsmadi@yu.edu.jo*