International Journal of Modern Engineering Research (IJMER)
www.ijmer.com Vol.2, Issue.4, July-Aug. 2012 pp-2601-2606 ISSN: 2249-6645

# Reviewing Testability of Object Oriented Systems for Non-Functional Specifications

Vaishali Chourey[1], Dr. Meena Sharma[2]
*(Department of Computer Science, Medi-Caps Institute, Indore, M.P. India)
** (Department of Computer Science, IET-DAVV Indore, M.P. India)

***ABSTRACT:*** *The object oriented programming proves to be the most beneficial paradigm for scalable and maintainable software development. An object characterizes special features like encapsulation, inheritance, modularity and polymorphism. The processes in Test Driven Design closely relate the agile methodology and strengthen the need of testing during the development stages. The UML specifies the architecture of the system. The design follows the specification and hence the implementation. The whole process adopts the language of UML from beginning of software through requirements specification till the deployment. The evaluation of software so as to be testable needs additional efforts. The weaker or ignored issues like the testing for non-functional requirements still need to be accommodated in the test design. The paper reviews the weaknesses of the object oriented systems and the models to be testable. It also includes metrics that quantifies the structural complexity of system to test its understandability and maintainability.*

***Keywords:*** *Models, Object Oriented Testing, Model Based testing, metric based evaluation, Weighted Complexity.*

## I. INTRODUCTION

Software engineering bridges its strengths to design and document the software development process through the use of various models. Booch, Jacobson and Rambaugh[1] conceptualized the behavioral, functional and implementation specific models that were sufficient to describe the elements and their relations in any object oriented software systems. Their contribution to the software engineering field for the same has been widely exercised in industry. The models have been the curious issue in the testing phenomenon whereby the testing is not delayed till the implementation phase but it goes simultaneously as it proceeds in the life cycle. This makes model based testing an interesting and exploratory research area. Model driven methodology proved its worth with the variety of software developed in varied areas. Any degree of complexity can be easily expressed with a set of diagrams. Whether it is structural or architectural specification or functional description, the use of objects and object design languages like UML has facilitated the tasks of project managers. It has even groomed the language of communication amongst the team of developers in any organization. Along with these the software industry benefits with the automated tools for programming high order languages. Altogether this has summed to an approach for robust and manageable code being developed quickly and efficiently. Software Testing is also a major phase in the development of the system for assuring its reliability and behavioral compliance to the requirements specified. A fact that 40-50% of the software development efforts are shared by testing makes it an important aspect.

The work in this paper compiles the major contributions in the areas of object oriented paradigm and its importance in the development phases. The section II contains an overview of the OO system and their characteristics. The section III contains the modeling language UML and the various diagrams with their importance. Section IV summarizes the issues that restrict testability of the designs. Section V contains the realization of theories related to class design metrics to map it to testing process. It brings the quantification of non-functional parameters like understandability and such attributes to contribute to testing results.

## II. OBJECTS AND OBJECT ORIENTED DESIGN

All the literature pertaining to the objects and object oriented design addresses the software projects commendable threats namely inadequate and unstable requirement, inadequate customer communications, poor team communications, unnecessary complexity and ineffective team behavior. Software projects are governed by the list of requirements specified by the customer who is the end user of the whole process. Thus the whole process is based on requirements that are actually the specifics of the system and the needs that are captured. The requirements once finalized do not change often, but the specification for building a piece of implementation of software are changed frequently and added throughout the development cycle. The challenge is to cope up with this frequent updates. Secondly the requirements must be verified with the customer before the design starts. Thus the interactions must conform to the initiation of the project. The next problem begins when the requirements that are verified are not “exactly” communicated to the developer team and an ambiguity occurs. The same requirement must flow properly to the developers. So, we have to make the models that are descriptions to the requirements graphically, thus lessening the probability of ambiguity in documentation if it was textual. As the analysis completes, the design of core transactions and implied operations have to be prepared. The behavior has to be formally or “pseudo-conventionally” specified with models so that they can be codified easily. This defines the system in the form of subsystem, components, classes, objects and their inter-relations like association, hierarchy and collaborations. Thus the system is added with another issue of its structural complexity because of its inter-related artifacts. The task of project

management is to clearly pass through these phases and bring up the system in its entirety.

Not only limited to the planning and development, but testing also is the requirement of the system and theories indicate the fact that effectiveness of object oriented technology lies in testing that goes parallel to development. Whether it is agile development process or any other technology, that depends on objects and specifies that the testing of models be made simultaneous. This reduces the risks from getting accumulated. There are many justifications to the development of models and their usages.

## III. UNIFIED MODELLING LANGUAGE OR THE UML

The UML is defined [1] as *"A language to specify, visualize, construct and document the artifacts of a software- intensive system."* It is a standard language for writing the blueprints of software. Ranging from the enterprise solutions to distributed web based applications; UML conceptualizes the artifacts of the system developed in object oriented languages. Thus, UML is inherently applicable to architecture centric, iterative and incremental project development. This language has rules and vocabulary for physical and conceptual representation of the software. The language has a collection of diagrams and relative specifications to handle the complexity of the system. For the enumeration, we have:

Table 1: Diagrams and Relevance in Software Project Management

| Diagram | Purpose |
|---|---|
| Use case Diagram | This diagram models and organizes the behavior of the system through its functionalities and services *to-through-for* actors of the system. |
| Class Diagram | This represents the set of classes, interfaces and their relationships. This diagram addresses the static and process view of the system. |
| Object Diagram | This diagram emphasises on the objects and their relationships. |
| Sequence Diagram | This is and interaction diagram that emphasises the time ordering of messages. |
| Collaboration Diagram | This is and interaction diagram that emphasises the structural organization of the objects that send and receive messages. |
| State-chart Diagram | This diagram addresses the dynamic view of the system especially useful in modelling reactive systems. |
| Activity Diagram | This is a diagram to model the functions of the system and emphasises flow of control among objects. It represents the sequence, concurrency and synchronization of various activities performed by the system. |
| Component Diagram | This diagram expresses the organization and dependencies among a set of components. |
| Deployment Diagram | This diagram shows the configuration of run-time processing nodes and corresponding components. |

The generality in the models and their *express*-ability makes it applicable to various areas like production, deployment and maintenance of software. However in the software development organization, the diagrams are conveniently adopted by analysts and end users for specifying the requirements, structure and behavior of the system. The architects who design the systems to satisfy the requirements specified and the developers who code the architecture into executables use the modeling conventions for communication and documentation. This is equally benefiting the quality assurance personnel who verifies and validates the system for its structure, behavior, functionality and other requirements. The monitoring of the development is emulsified with the process. This is the strength of UML widely acceptable by researchers and used by industries.

The UML has been an immensely popular issue in industry and research for Model Based Testing (MBT). [6] Models are the simplified version and representation of the systems and so are easily amenable for automated test case generation. Models can be classified into formal, semi-formal and informal models. Formal models are mathematically derived [3] from techniques of calculus theory, logic, state machines, markov chains etc., semi-formals combine the diagrams in ad-hoc conventions and are used in industries. Behavioral models are very significant for the test case generation [10, 12] as the bugs are indicated during test of a specific run or implementation of specific functionality of the system. Several research work and industry cases record the diagrams [5] with the Object Oriented Testing Strategies to test various aspects of the software. The table below describes it as:

| UML Diagram | Test Coverage | Type of Test | Fault Model |
|---|---|---|---|
| Class Diagram State Diagram | Code | Unit | Error Handling, correctness, |
| Class Diagram Interaction Diagram | Functional | Functional | Functional Behavior Integration Issues API Behavior |
| Usecase Diagram Activity Diagram Interaction Diagrams | Operational Scenarios | System | Contention Synchronization Workload Recovery |
| Class Diagram Interaction Diagram | Functional | Regression | Unexpected Behavior through system alterations |
| Usecase Diagram Deployment Diagrams | Inter-System Communications | Deployment Solution | Interoperability issues |

Table 2: Diagrams and Associated Tests

## IV. ISSUES IN OBJECT ORIENTED TESTING

The Object Oriented Testing requires additional techniques for its execution apart from the conventional ones. The testing takes two broad forms of Functional testing or Black box testing where the fulfillment of functional requirements is tested. Another is the Structural testing that tests for the structure of classes, their interactions and their states during execution of any method or activity. Structural testing is the white box testing of the OO systems.

The characteristic of object orientation makes testing rigorous over each iteration and phase of development [3-5]. The increments that are models like the analysis model is tested for requirement specific documentation and use cases. The design model is tested with the corresponding class diagram, interaction diagrams and activity diagrams.

Structural testing with methods and their code that contains statement, decision and path coverage are tested. All methods that are defined, newly added with the increment in functionality, inherited methods and methods that are redefined needs to be tested. The classes are tested for the state transitions during n activity, transaction flows are tested with messages that the classes share, exception testing for the exceptional behavior and conditions that a class may represent.

Object oriented nature poses difficulties to test a class without additional methods to access all the functionalities defined in the class and access its states. Each new instance for inheritance requires retesting. It is easy to test conditions, decisions, loops and exceptions within a class but it is difficult with the set of interaction amongst classes and requires special techniques. There are hierarchical structures but absence of hierarchical control flow makes the execution testing difficult. Integrated classes are thus tested with techniques like thread based testing, use based testing and cluster based testing. System testing is done for the recovery of systems from faulty conditions, security tests for unauthorized accesses, stress testing for load during execution and performance testing for reliability and availability with optimized execution.

**Heterogeneity in Models**:

Each model describes a different perspective of the same system, thus the testing of the object oriented models take different versions for each model[12]. There are contributions where these models have been used to generate test cases and respective test scripts are generated through automated process. The inputs to these test generators are the set of diagrams associated. Pretschner in his paper [4] presents a detailed discussion reviewing model based test generators. The studies in the area indicate that different test suites with the same coverage may detect fundamentally different number of errors. Also the above table [table 2] indicates that the single diagram alone may not suffice with the exhaustive test of a single type. During the development the industry follows ad-hoc modeling and do not comply with a defined set of diagrams. In such cases where the industry has its own conventions for design and documentation, it is required that the organizations develop their own framework and corresponding tools to build, manage and maintain test models.

**Choice of Models in Test Generation**:

An example system for the GPS navigation system for a car developed has to be tested for its functions and operations. A model based test brings up the test for the vehicle's position hereby ignoring other functionalities for the display and user interaction features [6]. Another test to the model may test a separate aspect of the system like route planning or route display and so on. The crucial factor is that the aspects are independent tests and they do not interact in terms of aspects. Thus not only diagrams but behavior also segregates the tests with diagrams.

**Skills, Audience and Tools**:

The issue arises when the testers need to be educated and trained on modeling practices. So far when the testing was confined to code, developers of the corresponding language who had an expertise could manage the testing. Thus the object oriented- model based testing expects modeling skills for the developers and the tester both. The limitation to the model based testing approach converges to the idea that only trained and technically apt audience can and are expected to create, read, review and maintain models. The concept is insufficient to bridge the gap between the models that are characterized with the quality of being best for human understanding and ones that are optimal for testing. Thus the tools that need to be customized with the testability of models have to be developed within the organization's development framework. There exists the limitation for behavior testing tools available for universal applicability.

**Scope**:

Models have a specific importance when the requirements are being matched to implementation parameters like class, methods or objects. The models however deal only with the superficially expressed behavior of the system which is the high level abstraction. Most of the models confined to the views of the system do not completely fit to the testing essentials. Thus in the early stages of development, the testers end up with almost prohibitive tasks of modelling parts of really large and complex systems. Summarizing, models constructed during the early development process lack several details of implementation that are required to generate test cases.

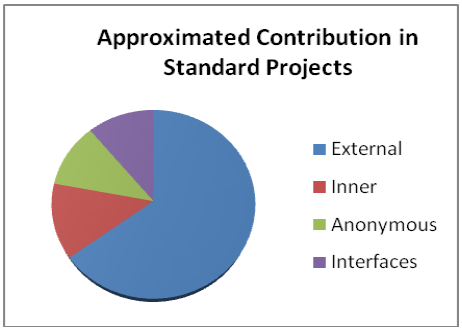
Also for the short development cycles, if there are new releases every week, that also reflects early construction phase of object oriented software development, the diagrams or models do not change accordingly. MBT do not pay off with such projects and more versatile tools are required for testing such typical projects. MBT are fruitful and can be used after releases have achieved a certain degree of stability in its features.

**Features of Programming Languages:**

Encapsulation [4] restricts the visibility of object states and *observ*-ability of intermediate test results. Inheritance causes invisible dependencies amongst hierarchically related classes. The approach that was devised for preventing code redundancy inhibits code dependencies of varied forms. The child classes that inherits parent's methods cannot be tested without testing the parent class. Abstract classes are the serious conditions where they can never be tested. Polymorphism extends to a limit of

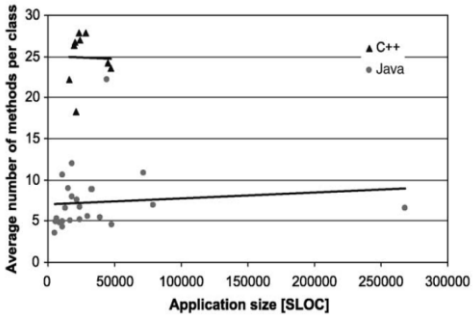
testing all possible conditions, paths for execution and potential errors that it may scope into the classes.

The literature on object orientation defines its strength as Open-Close Principle. The modules are open for extension but close for modification. This, when applied to classes, testing and maintainability is sacrificed. For any new behavior minor-or-major the classes are open for the inheritance. This becomes redundant with series of modifications hereby increasing the complexity of the system. The numbers of classes grow proportionally with the increment in form of new requirements, refined requirements or just additional classes in the development model. A research paper by John D. Mc Groger [17] proves through a formula derived that works as a multiplier function to estimate number of classes after each iteration.




(a)

The number of classes and the relationships amongst them contributes to structural complexity and is referred to in many researches, there by calculating cyclomatic complexity to quantify the attribute.




(b)

Fig 2: (a) Average Metric Values for Percentage of Classes of Each Type
(b) Average Metric value for the Number of Methods per Class

The above facts have been referred from the research [13, 18] done on various projects that estimate the total number of classes coded and the nature of classes along with the number of methods within each class. The average is depicted in statistics as above figures Fig 2 (a) and (b). Thus the number of classes and methods within each class has a vast average estimated and it is technically not feasible to test all the methods and a mid-way taking necessary implementations pass through the tests. Can the un-tested classes create errors and are there more intense testing methods to check the addition of classes and verify them, i.e making each class testable, is still a question to the project managers.

Also it is well proved that object oriented features like polymorphism, inheritance and encapsulation [18] create wide opportunities for the bugs to creep into the system that was less prevalent in traditional systems. It is also well exemplified in many cases like if many server objects function correctly at top level, but there is nothing to prevent a new client class from using it correctly. Thus, not only testing but developing becomes tedious and testing gets extended over a prolonged duration while final implementation gets ready.

The most important part of the analysis of system is calculating the complexity. Complexity is formally defined as the degree to which system or component has a design or implementation that is difficult to understand and verify [21]. This can be included in the system test that validates maintainability and understandability.

The classes may be analyzed with the metrics that measure the aspects of classes and the interactions amongst them[7]. These measures tells us more about our design and help quantify the maintainability. A change in one class will affect code in other classes, it should be minimal and classes with high dependency must be kept in same package. There are some metrics as:

**Intra Class Metrics:**

There are metrics at class level that may be helpful to calculate the complexity of the system. They can be reuse ratio, specialization ratio, number of external methods called, number of methods called in class hierarchy, number of local methods called, number of instance variables, number of modifiers, number of interfaces implemented and number of packages imported.

**LCOM** (Lack of Cohesion Methods): This metric refers the correlation between methods and the local instance variables of the class. High cohesion indicates good class subdivision.

**Unweighted Class Size**: This is calculated as number of methods and attributes of a class.

**Inter Class Metrics:**

This is measured by coupling at class level. Coupling is defined as a representation of the references between classes. If a class refers another class or it is being referenced then we measure it as coupling. There are parameters that still need to be standardized and can be defined as: Coupling between classes, Fan Out, Fan In, Efferent Coupling, Afferent Coupling. (Originally defined by Chidamber & Kemerer)[20].

**Response for Class**: It measures the coupling of classes in terms of method calls. It is the sum of number of methods in the class and the number of distinct method calls made by the methods in the class.

**Message Passing Coupling**: This metric measures the number of number of messages passing among objects of the class. A large value indicates high coupling and classes seem to be more dependent on each other. This increases the complexity of the system.

The above mentioned parameters are non- weighted measures, there are also metrics with weighted parameters like[22]:

**Weighted Class Complexity (WCC)**: The calculation is based on calculating the complexity of operations by considering corresponding cognitive weights. The cognitive weights are used to measure the complexity of the logical structures of the software that reside in the code as methods. They are classified and weighed as sequence (w=1), branch (w=2), iteration (w=3) and call (w=2) [15]. Initially the weight of individual method in a class is calculated by associating a weight with each method (member function) and add all the weights. This is weight due to methods and is called Method Complexity (MC). If there are n methods in a class then total method complexity is given by:

$$= \Sigma^{n} MC_{n}$$

The next step computes the total complexity due to attributes in the class and is denoted by $N_a$.

The complexity of a single class is called Weighted Class Complexity (WCC) and is given by:

$$WCC = N_a + \Sigma^{n} MC_{n}$$

If the total number of classes in the code is x then:

$$TotalWeightedClassComplexity = \Sigma\ WCC_x$$

The above weighted complexity calculation can be explained with the help of an example. The system comprises of following classes:

*Person*
*Student*
*Employee*
*Faculty*
*Administration*

The code exists like the one specified below and at each level, the complexity is calculated simultaneously.

```
/* Person Class is inherited by Student and Employee Class */

 PERSON CLASS
class Person
{
string name;    int age;    char gender;
public:
Person(string="" ,int=0, char='\0'); // W p1=1
Person(const Person &person); //copy constructor W p2=1
void print()const; //Wp3=Wp31+Wp32=2+1=3
string getName(){ // Wp4=1
return name; }
int getAge(){ //Wp5=1
return age;}
char getGender(){ //Wp6=1
return gender; }
};
//Person-default constructor
Person :: Person(string in, int ia, char is)
{ name = in; age = ia; gender = is; }
//Person-copy constructor

Person :: Person(const Person &p)
{ name = p.name; age = page; gender = p.gender; }
void Person :: print()const
{ cout<<"Name\t : "<<name<<'\n' ; //Wp31=1
cout<<"Age\t : "<<age<<'\n' ;
if (gender=='F') //Wp32=2
cout<<"Gender\t : Female" <<'\n' ;
else cout<<"Gender\t : Male" <<'\n' ; }

 STUDENT CLASS
class Student: public Person{ int sid; float gpa;
public:
Student(const Person &p,int student_id,float igpa):
Person(p) //WS1=1
{ sid = student_id;
gpa = igpa; }
void print()const; };
//WS2=WS21+WS22*WS23=1+2*2=5
void Student :: print()const
{ Person :: print();
cout<<"S.ID\t:"<<sid<<"\nGPA\t:"<<gpa<<endl;
//WS21=1
if (gpa>=2.0) //WS22=2
cout<<" Student is successful"<<endl;
else {if (gpa>=1.7) //WS23=2
cout<<"Student must improve GPA" <<endl;
else
cout<<"Student must repeat" <<endl;}}

/* ******** EMPLOYEE CLASS ********** */
class EMPLOYEE: public Person{ float salary;
public: EMPLOYEE::EMPLOYEE(const Person &p, float
sal):Person(p) ,salary(sal){} //WE1=1
EMPLOYEE(const EMPLOYEE
&EMPLOYEE):Person(EMPLOYEE){
salary=EMPLOYEE.salary; } //WE2=1
void print()const; }; //WE3=1
void EMPLOYEE::print() const{ Person::print();
cout<<"salary: "<<salary<<endl; }

FACULTY

class Faculty: public EMPLOYEE{
string branch;.
public: Faculty(const EMPLOYEE &e, string
b):EMPLOYEE(e),branch(b)    //WF1=1
{}
void print()const; }; //WF2=1

/* ******** ADMINISTRATIVE CLASS ********** */
class Administrative: public EMPLOYEE{string duty;
public:
Administrative(const EMPLOYEE &e,string
d="\0"):EMPLOYEE(e){duty=d;} //WA1=1
void print() const; }; //WA2=1
void sendMessage(string msg,Faculty &fac) //WA3=1
{cout<<"The incoming message :"<<msg<<". \nMessage
to"; cout<<fac.getName(); }
```

```
MAIN
int main(void)
{
Person * per[3];
per[0]=new Person ("Aysegul",27,'f');
per[1]=new Person ("Remzi",23,'m');
per[2]=new Person ("Ali",30,'m');
EMPLOYEE EMPLOYEE1(* per[0],1000);
EMPLOYEE1.print();
Student student1(* per[1],9299,3.5);
student1.print();
EMPLOYEE EMPLOYEE2(* per[0],2000);
Administrative
admEMPLOYEE(EMPLOYEE1,"Secretary");
Faculty facEMPLOYEE(EMPLOYEE2,"Computer");
admEMPLOYEE.sendMessage("Today there is a seminar at
your university. You are in
vited",facEMPLOYEE);
}
```

The example is referred from the research of [22] metric based calculation of complexity. This exactly computes the java code for complexity. The idea is to use the same derivations for the design where the classes are decided with its member functions and relations are defined. So far the calculation is based only on methods, attributes and relationship. The structural complexity is majorly due to relations and this can be well defined for class diagram and object (collaboration specifically) diagrams. The same calculations can be made during the iterations when classes grow and at each step the complexity may be curbed.

Thus, the re-arrangement of classes to maintain a proper metric can prove the system to be consistent in terms of growing number of classes and dependencies amongst classes. Lesser the complexity, more manageable is the design.

## V. CONCLUSION AND FUTURE WORK

An object benefits with its features of modularity, abstraction, encapsulation and inheritance but it was never predicted that the growing amount of code and maintenance classes bears loads on testing parameters. All the models have independent importance but require to be modified for testing individual aspects of the system. Several methods to derive a testable version of the UML have to be devised so that testing is not in the span of development but has intermediate phases upon stable models being developed. A future enhancement thereby adding the class complexity metrics to model based testing tools may be a convenient way to validate the understandability and maintainability parameters.

## REFERENCES

[1] Rumbaugh, Jacobson, and Booch., The Unified Modeling Language Reference Manual *Addison-Wesley, Reading MA, 1999.*

[2] Robert V. Binder, Object Oriented Testing: Myth and Reality, article published in *Object Magazine, May 1995.*

[3] Santosh kumar Swain, Model Based Object-Oriented Software Testing, *Journal of Theoretical and Applied Information Technology (JATIT 2005-2010)*

[4] G. Suganya, S.Neduncheliyan, A Study of Object Oriented Testing Techniques: Survey and Challenges, IEEE 2010.

[5] Clay E. Wiliams, Software Testing and the UML, Center of Software Engineering, *IBM T.J. Watson Research Center.*

[6] Mrk Utting, Position Paper: Model Based Testing.

[7] Tong Yi, Fangjun Wu and Chengzhi Gan, A Comparison of Metrics for UML Class Diagrams, *ACM SIGSOFT Volume 29, Number5, September 2004*

[8] Magiel Bruntink and Arie Van Deursen, Predicting Class Testability using Object-Oriented Metrics. *Paper from junit.org*

[9] Richard Torkar, Robert Feldt, and Tony Gorschek Test Cases Generation from UML Activity Diagrams, *Eighth ACIS International Conference on Software Engineering*, Artificial Intelligence and Parallel/ Distributed Computing.

[10] Richard Torkar, Robert Feldt, and Tony Gorschek , Extracting Generally Applicable Patterns from Object-Oriented Programs for the Purpose of Test Case Creation.

[11] Jehad Al Dallal ,Testing Object-Oriented Framework Applications Using FIST2 Tool: A Case Study, *World Academy of Science, Engineering and Technology 63 2010.*

[12] Harry M. Sneed, ANECON GmbH, The Drawbacks of model-driven Software Evolution.

[13] J. Kaczmarek, M.Kucharski, Size and Effort Estimation for applications written in Java, *Information and Software Technology 46(2004) pp 589-601, Sciencedirect and Elesvier Publication.*

[14] Arilo C. Dias Neto, Rajesh subramaniam, Marlon Vieira, Guilherme H. Travassos, A Survey on Model Based Testing approaches: A Systematic Review.

[15] Y. Wang and J. Shao, "A new measure of software complexity based on cognitive Weights." *IEEE Canadian Journal of Electrical and Computer Engineering, 2003.*

[16] Craig Larman, Applying UML and Patterns: *An Introduction to Object Oriented Analysis and Design and the Unified process, Second Edition, pp. 69-74*

[17] John D. McGregor, Managing Metrics in an Iterative Incremental Development Environment.

[18] E. V. Berard, "Issues in the Testing of Object-Oriented Software", Electro'94 International, *IEEE Computer Society Press, 1994, pp. 211–219.*

[19] Ibrahim K. El-Far and James A. Whittaker, Model Based Software Testing, *Encyclopaedia on Software Engineering Wiley 2001.*

[20] Martin Hitz, Behzad Montazari, Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective, *IEEE Transactions on Software Engineering 1996.*

[21] Standard for Software Quality Metrics Methodology, 1998, *IEEE Std. 1061-1998 IEEE Computer Society.*

[22] Sanjay Mishra & Ibrahim Akman, Weighted Class Complexity: A Measure of Complexity for Object Oriented System, *Journal of Information Science and Engineering 24, 2008.*