

# **Analysis and Design of Metrics for Aspect-Oriented Systems**

***A Thesis***

*Submitted in fulfillment of the  
requirements for the award of the degree of*

**Doctor of Philosophy**

**Submitted by**

**Avadhesh Kumar**  
(Registration No. 9051451)

**Under the supervision of**

**Dr. P S Grover**

Professor of Computer Science,  
Guru Tegh Bahadur Institute of Technology,  
Guru Gobind Singh Indraprastha University,  
Delhi – 110006.

**Dr. Rajesh Kumar**

Associate Professor,  
School of Mathematics and Computer  
Applications, Thapar University,  
Patiala –147004.



**School of Mathematics and Computer Applications  
Thapar University,  
Patiala–147 004 (Punjab), India.**

**June 2010**

***To***

***My Family***

# CERTIFICATE

I hereby certify that the work which is being presented in this thesis entitled **ANALYSIS AND DESIGN OF METRICS FOR ASPECT-ORIENTED SYSTEMS**, in fulfillment of the requirements for the award of degree of **DOCTOR OF PHILOSOPHY** submitted in school of Mathematics and Computer Applications (SMCA), Thapar University, Patiala, Punjab, is an authentic record of my own work carried out under the supervision of Dr. P.S. Grover and Dr. Rajesh Kumar, and refers the work of other researchers, which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.



(AVADHESH KUMAR)

Registration No. 9051451

This is to certify that the above statement made by the candidate is the correct and true to the best of my knowledge and belief.



(P S Grover)

Professor of Computer Science,  
Guru Tegh Bahadur Institute of Technology,  
Guru Gobind Singh Indraprastha University,  
Delhi - 110006.

Supervisor



(Rajesh Kumar)

Associate Professor,  
School of Mathematics and Computer  
Applications, Thapar University,  
Patiala -147004.

Supervisor

## ACKNOWLEDGEMENTS

---

The work leading to this doctoral thesis has been carried out by me as a research scholar at the School of Mathematics and Computer Applications (SMCA) of this University. These five years (2005-10) have been very stimulating and instructive for me.

At the outset, I am highly indebted to my revered supervisors, Dr. Rajesh Kumar, Associate Professor and Professor (Dr.) P.S. Grover, for their untiring support, encouragement and able guidance at each and every step throughout this research endeavour. I admire the knowledge of Professor P.S. Grover. It is really fortunate that I had such an opportunity to learn the ABC's of research from him, which will go a long way in my professional career. Further more, I find no words to express my heartfelt thanks to Dr. Rajesh Kumar for supervising this work so diligently and delightfully. It has indeed been a privilege to carry out this research work under their guidance.

I am grateful to the Director, Thapar University, Patiala for providing me University's resources and facilities necessary to carry out this research work. I express my gratitude to the Doctoral Committee comprising Professor (Dr.) S. S. Bhatia, Professor (Dr.) R. K. Sharma and Professor (Dr.) Seema Bawa for monitoring the progress and providing valuable suggestions for improvement of my Ph. D. research work from time to time. I am also thankful to the entire faculty and staff of SMCA for healthy discussions, suggestions and necessary cooperation to complete this work.

I wish to thank a great personality who greatly influenced my life and career, Dr. Ashok K. Chauhan, Founder President, RBEF (Amity University). I am thankful to Maj. Gen. K. Jai Singh (Retd.), Vice Chancellor, Amity University, Uttar Pradesh and my Director at AIIT for allowing me to use the Institute's resources, required for this research.

Special thanks are due to my friend and motivator Dr. Arun Sharma, Professor at Amity University, Noida, for his valuable suggestions and providing the support in all respect. I am thankful to friends working in corporate for providing valuable suggestions and code to evaluate and validate some of the metrics. I am also thankful to my colleague Mr. O. P. Sangwan for his active suggestions and advise on several occasions.

### *Acknowledgements*

My hearty thanks to all the anonymous referees of several international research journals in the domain of aspect-oriented systems, software quality and soft computing for their constructive comments and new ideas for improvement of the research work presented in Chapters 2–6 of this thesis.

I wish to thank many acclaimed researchers across the world for timely contributing the reprints of their valuable research papers for my reference apropos this dissertation. These include Dr. Will Tracz, Dr. Richard Wiener, Dr. S.K. Chang, Dr. Tharam S. Dillon, Dr. Elizabeth Chang, Dr. N. S. Gill, and several others.

Finally I would like to dedicate this thesis to my family. I thank my parents who gave me the courage and blessing to augment my education, and supported me in all achievements throughout my life. I also thank my in-laws for all type of supports during this project. I must acknowledge the constant cooperation and moral support of my loving wife, Mrs. Shrddha Sagar along with our daughters Pavani and Paurvi throughout this project without which it could have not been possible to achieve this target so contentedly.

Above all, I pay my reverence to the almighty God.

**Date: June, 2010.**

**Place: Thapar University, Patiala.**

  
(Avadhesh Kumar)

# ABSTRACT

---

Aspect-Oriented Software Development (AOSD) is a relatively new programming paradigm, which encapsulates concerns that crosscut the modularity of traditional programming mechanisms such as Module-Oriented (MO) and Object-Oriented (OO). The concern that crosscuts the modularity of traditional programming mechanism is known as a crosscutting concern (Garcia *et al.*, 2004). Examples of crosscutting concerns include *logging, tracing, caching, security, resource pooling* etc. The ability to modularize such concerns is expected to improve parallel development, comprehensibility, reuse, ease of change, reducing development costs, increasing dependability and adaptability. According to Alexander (2003), among the claimed benefits of this new technology is the reduction in the amount of code written and higher cohesion. There are many Aspect-Oriented Programming languages available in the industry, for example- AspectJ (a Java extension), AspectC (a C extension), AspectC++ (a C++ extension), AspectXML (a XML extension), CaesarJ and Hyper/J (being used by IBM).

However, AOSD still faces several problems. Quantitative assessment of some of the important characteristics such as modularity, maintainability, reusability, complexity and the overall quality of Aspect-Oriented (AO) technology is fairly unexplored. OO technology has matured enough and many quantitative models are available to determine various characteristics of this technology. But very few existing evaluations have been performed at qualitative and quantitative levels in AOSD. Since AO has some additional features from OO technology, there is a need to develop quantitative models to determine above said important characteristics of AO technology.

In present study, our focus is on those AO languages that have features of Java and AO technology. In this category, we have selected AspectJ, CaesarJ and Hyper/J AOP languages. The proposed metrics in present work will be applicable to Java,

AspectJ, CaesarJ and Hyper/J. The term generic/unified has been used which covers all these programming languages.

## Objectives of the Proposed Work

In this research work, the main emphasis is on quantitative evaluation and analysis of various features that are important in the context of AOSD. The following are issues and problems that have been explored in this study:

1. Static and dynamic metrics for measuring the performance of aspect-oriented software designed with Java and AspectJ, CaesarJ and Hyper/J.
2. Reusability and maintainability of aspect-oriented software.
3. Metrics for measuring the complexity of aspect-oriented software designed with AspectJ, CaesarJ and Hyper/J from various viewpoints, such as code complexity and interaction complexity.
4. Propose a quality model to bring out the relationship among quality characteristics and validate the suggested model.

The issues stated above have been accomplished as follows:

**Objective 1:** *To investigate and propose static and dynamic metrics for measuring the performance of aspect-oriented software designed with Java, AspectJ, CaesarJ and Hyper/J.*

Metrics have been thoroughly studied and explored for several quality characteristics of Aspect-Oriented Systems (AOS). These features include *cohesion*, *coupling*, *complexity*, *changeability* (a sub-characteristic of maintainability), *reusability* and others. The new proposed metrics are for *cohesion*, *coupling*, *complexity*, *size*, and *changeability* of generic AO systems. These proposed metrics include both static as well as dynamic metrics. These metrics can be applied on software designed by using Java, AspectJ, CaesarJ and Hyper/J.

**Objective 2:** *To assess the reusability and maintainability of aspect-oriented software.*

A model has been proposed to determine how to broaden the applicability of class/aspect. Using this proposed model, reusability can be determined by direct measurable metrics for *cohesion*, *coupling*, *complexity*, and *size*. In order to predict maintainability of AOS, we have proposed a metric for changeability. In AOS, when a crosscutting concern is encapsulated in an aspect, it makes changes easy. So, here the focus is on changeability sub-characteristic of maintainability.

**Objective 3:** *To design metrics for ascertaining the complexity of aspect-oriented software designed with AspectJ, CaesarJ and Hyper/J from various viewpoints, such as code and interaction complexity.*

Metrics have been defined for measuring code complexity and interaction complexity of generic aspect-oriented systems implemented in Java, AspectJ, CaesarJ and Hyper/J. We have used fuzzy logic for defining software complexity. Complexity metrics have been used as input variables in a fuzzy inference engine. The motivation has been to understand and explain the difficulties faced in calculating the total complexity of an AO component in generic aspect-oriented systems. Here, a component is defined in new terminology as a class in all four programming languages, an interface in all four programming languages, and finally an aspect in AspectJ, CClass in CaesarJ and Hyperslice in Hyper/J. Using this fuzzy logic tool, automation of measuring complexity of generic AO system has been achieved.

**Objective 4:** *To propose a quality model to bring out the relationship among quality characteristics in aspect-oriented software and validate the suggested model.*

A quality model for aspect-oriented system has been proposed, which includes quality characteristics like *reusability*, *complexity*, *code-reducibility* and *modularity*, along with others, which are present in ISO/IEC 9126 quality model. In order to evaluate the quality of proposed the model as single unit, Analytic Hierarchy Process (AHP) has been applied. Pair-wise relative weights of characteristics and sub-characteristics have been collected through a survey conducted on professionals working in the field of aspect-oriented technology. A case study from one of the projects developed in AspectJ has also been considered for validating the suggested model.

## Thesis Outline

Thesis is divided into seven chapters:

**Chapter 1** covers the basic issues and our understanding of aspects and aspect-oriented software development (AOSD). It consists of a detailed literature survey of aspect-oriented systems (AOS) and metrics. It covers internal characteristics like *cohesion*, *coupling*, *complexity* and *size* as well as external characteristics like *maintainability*, *understandability*, *usability*, *modularity*, *reusability* and others in context of AOS. Some of the metrics discussed in this survey are derived from object-oriented systems (OOS) with no or minor changes, others are exclusively specific for AOS. This Chapter also covers various quality models such as McCall, Boehm, FURPS, Dromey, ISO/IEC 9126, Bertoa, Adnan, and others. Major findings of these models in context of AOS are also been presented here and discussed.

**Chapter 2** proposes a metrics suite for measuring internal characteristics such as *cohesion*, *coupling*, *complexity*, and *size* of generic aspect-oriented systems. Here, generic means, the metrics that are applicable to most of the aspect-oriented programming languages such as AspectC, AspectC++, AspectXML, AspectJ, CaesarJ and Hyper/J. The present study is focused on those AOP languages that have most of the features of the Java programming language and AO technology. In this category, we have selected AspectJ, CaesarJ and Hyper/J. So our metrics suite is applicable to Java, AspectJ, CaesarJ and Hyper/J programming languages. In order to define metrics for internal characteristics, new terminology and formulism have been defined to present unambiguous, standardized and operational terminology. These proposed direct measurable metrics for internal characteristics can be used to determine external characteristics such as *maintainability*, *understandability*, *usability*, *adaptability*, and *modularity*. This terminology and metrics have been used in rest of the Chapters.

**Chapter 3** proposes a fuzzy logic approach to measure complexity of aspect-oriented systems. The complexity of AOS has been divided in two categories: *code*

*complexity* and *interaction complexity*. Metrics for the complexity have already been defined in Chapter 2. Code complexity is due to complexity of attributes, complexity of operations and complexity of nested components. The complexity values of attributes, operations, and nested operations have been normalized between 0-1. These complexity metrics have been used as input variables in a fuzzy inference engine. Attributes have been divided into three categories: low (L), medium (M) and high (H) as member functions. NIL variable for attribute means there is no attribute in the component. Operations have also been divided into three categories: L, M, and H as member functions. NIL variable for operation represents that there is no operation in the component. Nested components and components have been divided into five categories-very low (VL), L, M, H and very high (VH) as member functions. NIL variable for a nested component represents that there is no nested component present in the component. NIL variable for a component means complexity of component is zero. A similar approach can be applied to measure interaction complexity of AO components. Using this fuzzy logic tool, the complexity of AO system can be calculated automatically.

**Chapter 4** proposes an approach to predict maintainability of aspect-oriented systems. In aspect-oriented technology, changes in crosscutting concerns become easy. Here, we have focused in measuring changeability, a sub-characteristic of maintainability in ISO/IEC 9126. A metric for changeability has been defined using terminology and formalism defined in Chapter 2. Possible types of changes in AOS have been determined. Using these feasible changes, an experiment is performed on projects designed in AspectJ and CaesarJ. Changeability of generic AO system is assessed by making changes at code level. Impact of these changes has been evaluated. High change impact of an AO component means, it will absorb less change. In other words, high change impact means low changeability. In order to validate proposed changeability measurement technique, dependency on a component of other components has also been measured. Number of dependencies has been measured with the help of adjacency matrix, which is one of the existing techniques. Correlation analysis between average change impact and number of dependencies gives a correlation factor value. Correlation factor in CaesarJ based system is 0.87 and in AspectJ based system is 0.89. These values are high enough to indicate that

in high dependent AO system, change impact will be high. That means, a highly dependent AO system will absorb low changeability. It has also been seen from the existing literature that highly dependent systems lead to poor maintainability, which is also proved by our work.

**Chapter 5** proposes a model to assess reusability of AO components of AOS. Here, reusability means how wider an AO component can be reused in other AO components. In this proposed model, we determined relationship of reusability with external and internal characteristics of AOS. In order to determine the external characteristics on which reusability of AO components depends, a survey was conducted on experts of the domain. From the survey, it has been observed that there are four main external characteristics, *understandability*, *modularity*, *maintainability*, and *adaptability* on which reusability of AO components depends. These external characteristics cannot be measured directly. In order to predict how wider an AO component can be reused, we can also use directly measurable metrics for internal characteristics such as *cohesion*, *coupling*, *complexity*, and *size* proposed in Chapter 2. A correlation analysis among indirect measurable external characteristics, *understandability*, *modularity*, *maintainability*, *adaptability*, and all direct measurable metrics for internal characteristics has been performed. Through this correlation analysis, it has been concluded that (i) metrics  $CMPX_{M(c)}$ , NOO, LOC, and CoOI (when component is an aspect), are directly proportional to the reusability of AO component; (ii) metric CoI is inversely proportional to the reusability of AO component; and (iii) there is no strong relation among external characteristics and CoAT, CoPT, CoAR, NOA, and CoOI (when component is a class) metrics. These metrics have been defined in Chapter 2 for generic AO systems.

**Chapter 6** proposes a new quality model, Aspect-Oriented Software Quality Model (AOSQUAMO), by adding some new sub-characteristics, *reusability*, *complexity*, *code-reducibility* and *modularity* to the existing ISO/IEC 9126 quality model, which are important in the context of AOS. In order to assign weights to characteristics and sub-characteristics of proposed model, we conducted a survey on software professionals working in the field of aspect-oriented technology. The survey form consists seven tables

for filling pair-wise relative weight values of characteristics and sub-characteristics. These data are analyzed by using Analytic Hierarchy Process (AHP), wherein weight values are calculated for quality characteristics and sub-characteristics. The weight values obtained will help developers to select only those characteristics and sub-characteristics which are important and relevant as per their quality requirement in that domain. A case study has also been considered to validate the proposed model. This experimentation leads to the useful conclusion related to the capability of measuring quality of aspect-oriented systems as a single unit.

This thesis concludes in **Chapter 7** with major finding of the present work. It also suggests some future work in this direction.

# CONTENTS

|                                                                                          |             |
|------------------------------------------------------------------------------------------|-------------|
| <b>CONTENTS</b>                                                                          | <b>i</b>    |
| <b>LIST OF FIGURES</b>                                                                   | <b>vii</b>  |
| <b>LIST OF TABLES</b>                                                                    | <b>ix</b>   |
| <b>ABBREVIATIONS</b>                                                                     | <b>xi</b>   |
| <b>LIST OF PUBLICATIONS BY THE AUTHOR</b>                                                | <b>xv</b>   |
| <b>CHAPTER 1: ASPECT-ORIENTED SYSTEMS - AN INTRODUCTION<br/>AND REVIEW OF LITERATURE</b> | <b>1-24</b> |
| 1.1 Introduction                                                                         | 1           |
| 1.2 Aspect-Oriented Software Development                                                 | 3           |
| 1.2.1 Aspect Weaver-Composer of the Final System                                         | 4           |
| 1.2.2 Aspect-Oriented Programming Languages                                              | 5           |
| 1.2.2.1 AspectJ                                                                          | 5           |
| 1.2.2.2 CaesarJ                                                                          | 6           |
| 1.2.2.3 Hyper/J                                                                          | 7           |
| 1.3 Literature Review                                                                    | 7           |
| 1.3.1 Software Metrics                                                                   | 7           |
| 1.3.1.1 Metrics for Internal Characteristics for Legacy Systems                          | 8           |
| 1.3.1.2 Metrics for Internal Characteristics for Aspect-Oriented Systems                 | 10          |
| 1.3.2 Software Maintainability                                                           | 16          |
| 1.3.2.1 Maintainability of Legacy Systems                                                | 16          |
| 1.3.2.2 Maintainability of Aspect-Oriented Systems                                       | 16          |
| 1.3.3 Software Reuse and Reusability                                                     | 17          |
| 1.3.3.1 Reusability of Legacy Systems                                                    | 18          |

|         |                                            |    |
|---------|--------------------------------------------|----|
| 1.3.3.2 | Reusability of Aspect-Oriented Systems     | 19 |
| 1.3.4   | Software Quality Models                    | 20 |
| 1.3.4.1 | Quality Models for Legacy Systems          | 20 |
| 1.3.4.2 | Quality Models for Aspect-Oriented Systems | 22 |
| 1.4     | Conclusion                                 | 23 |

## **CHAPTER 2: METRICS SUITE FOR GENERIC ASPECT-ORIENTED SYSTEMS** **25-60**

|         |                                         |    |
|---------|-----------------------------------------|----|
| 2.1     | Introduction                            | 25 |
| 2.2     | Terminology and Formalism               | 28 |
| 2.2.1   | Structure                               | 28 |
| 2.2.2   | System                                  | 29 |
| 2.2.3   | Components                              | 29 |
| 2.2.4   | Operations                              | 30 |
| 2.2.5   | Operation Invocations                   | 32 |
| 2.2.6   | Indirect Operation Invocations          | 32 |
| 2.2.7   | Attributes                              | 33 |
| 2.2.8   | Attribute References                    | 33 |
| 2.2.9   | Types                                   | 34 |
| 2.3     | Related Work                            | 34 |
| 2.4     | Unified Frameworks                      | 40 |
| 2.4.1   | Unified Framework for Cohesion Measures | 40 |
| 2.4.1.1 | Type of Cohesion Connections            | 40 |
| 2.4.1.2 | Domain of Measure                       | 41 |
| 2.4.1.3 | Direct or Indirect Connections          | 42 |
| 2.4.1.4 | Inheritance                             | 43 |
| 2.4.1.5 | Access Operations and Constructors      | 44 |
| 2.4.2   | Unified Framework for Coupling Measures | 45 |
| 2.4.2.1 | Type of Coupling Connections            | 46 |
| 2.4.2.2 | Locus of Impact                         | 48 |
| 2.4.2.3 | Granularity of the Measure              | 48 |

|                                                                                                          |                                                                    |              |
|----------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|--------------|
| 2.4.2.4                                                                                                  | Direct or Indirect Connections                                     | 50           |
| 2.4.2.5                                                                                                  | Stability of Server                                                | 50           |
| 2.4.2.6                                                                                                  | Inheritance                                                        | 50           |
| 2.4.2.7                                                                                                  | Aggregation                                                        | 50           |
| 2.4.2.8                                                                                                  | Dynamic Issues                                                     | 51           |
| 2.4.2.9                                                                                                  | Language Specific Considerations                                   | 51           |
| 2.5                                                                                                      | AOS Metrics Suite                                                  | 51           |
| 2.5.1                                                                                                    | Cohesion Metric                                                    | 52           |
| 2.5.1.1                                                                                                  | Selection Criteria                                                 | 52           |
| 2.5.1.2                                                                                                  | Formal Definition of Cohesion Metric                               | 52           |
| 2.5.2                                                                                                    | Coupling Metrics                                                   | 53           |
| 2.5.3                                                                                                    | Complexity Metrics                                                 | 54           |
| 2.5.3.1                                                                                                  | Selection Criteria                                                 | 55           |
| 2.5.3.2                                                                                                  | Formal Definition of Complexity Metrics                            | 55           |
| 2.5.4                                                                                                    | Size Metrics                                                       | 58           |
| 2.6                                                                                                      | Conclusion                                                         | 59           |
| <b>CHAPTER 3: ASSESSMENT OF COMPLEXITY OF GENERIC ASPECT-ORIENTED SYSTEMS USING FUZZY LOGIC APPROACH</b> |                                                                    | <b>61-74</b> |
| 3.1                                                                                                      | Introduction                                                       | 61           |
| 3.2                                                                                                      | Complexity Concepts                                                | 62           |
| 3.2.1                                                                                                    | Complexity Metrics for Module-Oriented and Object-Oriented Systems | 62           |
| 3.2.2                                                                                                    | Fuzzy Logic and Complexity Metrics for Aspect-Oriented Systems     | 64           |
| 3.3                                                                                                      | Proposed Complexity Model                                          | 65           |
| 3.4                                                                                                      | A Fuzzy-Logic Approach to Complexity Metrics and Model             | 66           |
| 3.4.1                                                                                                    | Membership Functions For Input Parameters                          | 67           |
| 3.4.2                                                                                                    | Fuzzy Rules for the Proposed Model                                 | 71           |
| 3.5                                                                                                      | Conclusion                                                         | 74           |

**CHAPTER 4: PREDICTING MAINTAINABILITY OF ASPECT-ORIENTED SOFTWARE SYSTEMS** **75-90**

|       |                                                |    |
|-------|------------------------------------------------|----|
| 4.1   | Introduction                                   | 75 |
| 4.2   | Maintainability                                | 76 |
| 4.2.1 | Changeability in Traditional Software Systems  | 77 |
| 4.2.2 | Changeability in AO Systems                    | 78 |
| 4.3   | Change Impact Analysis                         | 80 |
| 4.3.1 | Code Level Changes                             | 80 |
| 4.3.2 | Change Impact Evaluation                       | 82 |
| 4.4   | Case Study                                     | 82 |
| 4.4.1 | Validation                                     | 86 |
| 4.5   | Cohesion Measure as Indicator of Changeability | 86 |
| 4.5.1 | Results                                        | 87 |
| 4.6   | Conclusion                                     | 90 |

**CHAPTER 5: REUSABILITY ASSESSMENT IN ASPECT-ORIENTED SOFTWARE SYSTEMS** **91-105**

|     |                                |     |
|-----|--------------------------------|-----|
| 5.1 | Introduction                   | 91  |
| 5.2 | Reusability Metrics and Models | 92  |
| 5.3 | Aspect-Oriented Metrics Suite  | 94  |
| 5.4 | Proposed Reusability Model     | 95  |
| 5.5 | Empirical Assessment           | 96  |
| 5.6 | Experimental Case Study        | 97  |
| 5.7 | Results and Discussions        | 103 |
| 5.8 | Conclusion                     | 104 |

**CHAPTER 6: QUALITY MODEL** **107-130**

|     |              |     |
|-----|--------------|-----|
| 6.1 | Introduction | 107 |
|-----|--------------|-----|

|         |                                                                    |     |
|---------|--------------------------------------------------------------------|-----|
| 6.2     | Software Quality Characteristics and Models                        | 109 |
| 6.2.1   | McCall et al. Model                                                | 109 |
| 6.2.2   | Boehm et al. Model                                                 | 110 |
| 6.2.3   | ISO/IEC 9126 Software Quality Model                                | 110 |
| 6.2.4   | Dromey's Model                                                     | 112 |
| 6.2.5   | Other Models                                                       | 112 |
| 6.3     | Aspect-Oriented Software Quality Model (AOSQUAMO)                  | 113 |
| 6.3.1   | Definition of Proposed New Characteristics Added                   | 116 |
| 6.3.1.1 | Modularity                                                         | 116 |
| 6.3.1.2 | Code-Reducibility                                                  | 116 |
| 6.3.1.3 | Complexity                                                         | 116 |
| 6.3.1.4 | Reusability                                                        | 117 |
| 6.3.2   | Definition of Existing Characteristics in Context of AO Technology | 117 |
| 6.3.2.1 | Functionality                                                      | 118 |
| 6.3.2.2 | Reliability                                                        | 119 |
| 6.3.2.3 | Usability                                                          | 120 |
| 6.3.2.4 | Efficiency                                                         | 120 |
| 6.3.2.5 | Maintainability                                                    | 121 |
| 6.3.2.6 | Portability                                                        | 121 |
| 6.4     | Evaluation of AOSQUAMO using Analytic Hierarchy Process            | 122 |
| 6.4.1   | Allocating the Weights to Characteristics and Sub-Characteristics  | 124 |
| 6.4.2   | Determining Eigen Vector and Eigen Values                          | 125 |
| 6.4.3   | Evaluating Quality of AO Projects                                  | 129 |
| 6.5     | Conclusion                                                         | 129 |

## **CHAPTER 7: CONCLUDING OBSERVATIONS AND FUTURE SCOPE 131-134**

|     |              |     |
|-----|--------------|-----|
| 7.1 | Introduction | 131 |
|-----|--------------|-----|

|                   |                                                |                |
|-------------------|------------------------------------------------|----------------|
| 7.2               | Review of the Results Obtained from this Study | 131            |
| 7.3               | Limitations and Future Scope of the Study      | 133            |
| <b>REFERENCES</b> |                                                | <b>135-152</b> |

## LIST OF FIGURES

---

| <b>Figure No.</b> | <b>Figure Name</b>                                                     | <b>Page</b> |
|-------------------|------------------------------------------------------------------------|-------------|
| 1.1               | Aspect Weaver                                                          | 4           |
| 2.1               | Components of an Aspect-Oriented System                                | 29          |
| 3.1               | Fuzzy Model for Complexity Measurement of a Component                  | 66          |
| 3.2               | Member Functions of Input Variable CMPXAtt                             | 68          |
| 3.3               | Member Functions of Input Variable CMPXOp                              | 71          |
| 3.4               | Member Functions of Input Variable CMPXNested                          | 71          |
| 3.5               | Rule Viewer of AOPComplexity                                           | 73          |
| 3.6               | Surface Viewer of AOPComplexity                                        | 74          |
| 5.1               | Relationship of Reusability with External and Internal Characteristics | 96          |
| 5.2               | Line Graphs of External Characteristics                                | 101         |
| 5.3               | Line Graphs of Internal Characteristics                                | 101         |
| 5.4               | Line Graphs of Internal Characteristics                                | 102         |
| 6.1               | Boehm Quality Model                                                    | 110         |



## LIST OF TABLES

---

| <b>Table No.</b> | <b>Table Title</b>                                      | <b>Page</b> |
|------------------|---------------------------------------------------------|-------------|
| 2.1              | Domain of Measures for Cohesion                         | 42          |
| 2.2              | Types of Coupling Connections                           | 46          |
| 2.3              | Domain of Measures for Coupling                         | 49          |
| 3.1              | Halstead Complexity Measures                            | 63          |
| 3.2              | Complexity Weight Values to Individual Attribute        | 67          |
| 3.3              | Type of Operations                                      | 69          |
| 3.4              | Complexity Weight Values of Operations                  | 70          |
| 3.5              | Some Sample Rules of the Complexity Fuzzy Model         | 72          |
| 4.1              | Criteria for Minimum Number of Random Changes           | 81          |
| 4.2              | CaesarJ Project- Tested for Changeability               | 83          |
| 4.3              | AspectJ Project- Tested for Changeability               | 85          |
| 4.4              | CaesarJ Project- Tested for Cohesion and Changeability  | 87          |
| 4.5              | AspectJ Project- Tested for Cohesion and Changeability  | 88          |
| 5.1              | Metrics Values for Internal Characteristics             | 98          |
| 5.2              | Metrics Values for External Characteristics             | 100         |
| 6.1              | Quality Characteristics of the ISO/IEC 9126             | 111         |
| 6.2              | AOSQUAMO Model                                          | 115         |
| 6.3              | Eigen Vectors and Eigen Values for Main Characteristics | 126         |
| 6.4              | Eigen Vectors and Weights for Sub-Characteristics       | 127         |



## ABBREVIATIONS

---

|          |                                        |
|----------|----------------------------------------|
| ACI      | Average Change Impact                  |
| AHP      | Analytic Hierarchy Process             |
| ANC      | Actual Number of Connections           |
| AO       | Aspect-Oriented                        |
| AOA      | Aspect-Oriented Approach               |
| AOD      | Aspect-Oriented Design                 |
| AOP      | Aspect-Oriented Programs               |
| AOS      | Aspect-Oriented Systems                |
| AOSD     | Aspect-Oriented Software Development   |
| AOSQUAMO | Aspect-Oriented Software Quality Model |
| AR       | Attribute References                   |
| BT       | Built-in Types                         |
| CAE      | Coupling on Advice Execution           |
| CBC      | Coupling Between Components            |
| CBO      | Coupling Between Objects               |
| CBSD     | Component-Based Software Development   |
| CC       | Cyclomatic Complexity                  |
| CDA      | Crosscutting Degree of an Aspect       |
| CFA      | Coupling on Field Access               |
| CIM      | Coupling on Intercepted Module         |

|        |                                                |
|--------|------------------------------------------------|
| CLC    | Class Level Coupling                           |
| CMC    | Coupling on Method Calls                       |
| CoAR   | Coupling on Attribute Reference                |
| CoAT   | Coupling on Attribute Type                     |
| CoHA   | Coupling on High Level Association             |
| CoI    | Coupling on Inheritance                        |
| CoOI   | Coupling on Operation Invocation               |
| CoPT   | Coupling on Parameter Type                     |
| CT     | Component Types                                |
| DIT    | Depth of Inheritance Tree                      |
| EOI    | Explicit Operation Invocation                  |
| IOI    | Implicit Operation Invocation                  |
| JP     | Join Point                                     |
| LCC    | Lack of Concern-based Cohesion                 |
| LCOM   | Lack of Cohesion in Methods                    |
| LCOO   | Lack of Cohesion in Operations                 |
| LOC    | Lines of Code                                  |
| MMRE   | Mean Magnitude Relative Error                  |
| MO     | Module-Oriented                                |
| MOA    | Module-Oriented Approach                       |
| NAO    | Number of Affected Operations                  |
| NFFPCM | Neuro-Fuzzy Function Point Calibration Model   |
| NIST   | National Institute of Standards and Technology |

|      |                             |
|------|-----------------------------|
| NOA  | Number of Attributes        |
| NOC  | Number of Children          |
| NOO  | Number of Operations        |
| NOSC | Number of System Components |
| OLC  | Object Level Coupling       |
| OO   | Object-Oriented             |
| OOA  | Object-Oriented Approach    |
| OOP  | Object-Oriented Programming |
| OOS  | Object-Oriented Systems     |
| PCD  | Pointcut Descriptor         |
| PSS  | Physical Source Statements  |
| RFC  | Response for a Class        |
| RFM  | Response for a Module       |
| UDT  | User-Defined Types          |
| WMC  | Weighted Methods per Class  |



## LIST OF PUBLICATIONS BY THE AUTHOR

---

### International Journals:

1. Rajesh Kumar, P.S. Grover, and **Avadhesh Kumar** “A Fuzzy Logic Approach to Measure Complexity of Generic Aspect-Oriented Systems”, **Journal of Object Technology (JOT)**, Volume 9, No. 3, pp: 43-57, May/June 2010.
2. **Avadhesh Kumar**, Rajesh Kumar, and P.S. Grover, “Unified Cohesion Measures for Aspect-Oriented Systems”, accepted for publication in **International Journal of Software Engineering and Knowledge Engineering (IJSEKE)**, Volume 20, No. 6, September Issue 2010.
3. **Avadhesh Kumar**, P.S. Grover, and Rajesh Kumar, “A Quantitative Evaluation of Aspect-Oriented Software Quality Model (AOSQUAMO)”, **ACM SIGSOFT Software Engineering Notes**, Volume 34, Issue 5, pp: 1-9, September 2009.
4. **Avadhesh Kumar**, Rajesh Kumar, and P.S. Grover, “Generalized Coupling Measure for Aspect-Oriented Systems”, **ACM SIGSOFT Software Engineering Notes**, Volume 34, Issue 3, pp: 1-6, May 2009.
5. P.S. Grover, Rajesh Kumar, and **Avadhesh Kumar**, “Measuring Changeability for Generic Aspect-Oriented Systems”, **ACM SIGSOFT Software Engineering Notes**, Volume 33, Issue 6, pp: 1-5, November 2008.
6. **Avadhesh Kumar**, Rajesh Kumar, P.S. Grover, “An Evaluation of Maintainability of Aspect-Oriented Systems: a Practical Approach”, **International Journal of Computer Science and Security**, Volume 1, Issue 2, pp: 1-9, August 2007.

### **National Journals:**

7. **Avadhesh Kumar**, Rajesh Kumar, and P.S. Grover, “*A Comparative Study of Aspect-Oriented Methodology with Module-Oriented and Object-Oriented Methodologies*”, **ICFAI Journal of Information Technology**, Volume 2, No 4, pp: 7-15, December 2006.

### **International Conferences:**

8. **Avadhesh Kumar**, Rajesh Kumar, and P.S. Grover, “*Towards a Unified Framework for Complexity Measurement in Aspect-Oriented Systems*” , 2008 International Conference on Computer Science & Software Engineering (CSSE 2008), Wuhan, China, pp: 98-103, **IEEE Computer Society**, December 12-14, 2008.
9. **Avadhesh Kumar**, Rajesh Kumar, and P.S. Grover, “*Towards a Unified Framework for Cohesion Measurement in Aspect-Oriented Systems*”, 19th Australian Software Engineering Conference, 2008 (ASWEC 2008) Perth, Western Australia, pp: 57-65, **IEEE Computer Society**, March 26-28, 2008.
10. **Avadhesh Kumar**, Rajesh Kumar, and P.S. Grover, “*A Change Impact Assessment in Aspect-Oriented Software Systems*”, International Software Engineering Conference Russia 2006 (SECR-2006), pp: 83-87, November 16-17, 2006.

### **National Conferences:**

11. **Avadhesh Kumar**, Rajesh Kumar, P.S. Grover, “*A Critical Review of Cohesion Measures and Measurement Frameworks in Aspect-Oriented Systems*”, National Conference on Computing for National Development (INDIACom-2008), Delhi, India, pp: 415-420, February 8 - 9, 2008.

### **Communicated:**

12. Rajesh Kumar, P.S. Grover, and **Avadhesh Kumar** “*Assessment of Reusability in Aspect-Oriented Systems*”, communicated to **Software Quality Journal (Springer)**.



# CHAPTER 1

## ASPECT-ORIENTED SYSTEMS - AN INTRODUCTION AND REVIEW OF LITERATURE

---

### 1.1 INTRODUCTION

Software development is a highly complex and intellect-oriented activity. In the early days of program development, developers wrote programs using machine language in which programmers spent more time thinking about a particular machine's instruction set than the problem at hand. Gradually, programmers migrated to higher-level languages.

In order to develop software using higher level languages, there are different approaches and selection of software development approach depends on the type of application to be developed. In the seventies Module-Oriented Approach (MOA) was the dominant approach for software development. Modular programming is a programming paradigms based upon the concept of the procedure (also known as subroutine, method and function) call. Any given procedure might be called at a point during program's execution. Modular programming is a better choice than simple sequential or unstructured programming in many situations, which involve moderate complexity or which require significant ease of maintainability. Some important features and limitations of MOA include (i) The ability to reuse the same code at different places in the program without copying it (ii) An easier way to keep track of program flow than a collection of "go to" or "jump" statements (iii) The ability to be strongly modular than structure (iv) To enhance some feature of any module, new code is required to write/replace (v) No

proper binding of data with operations and (vi) A concern scattered in multiple modules cannot be modified easily. But as complexity grew we require better techniques.

Object-Oriented Approach (OOA) aimed at removing some of the flaws encountered with MOA. Now-a-days, OOA (Booch, 1994) has become the mainstream programming paradigm where real world problems are decomposed into objects that abstract behavior and data in a single unit. Object-Oriented Programming (OOP) encourages software reuse by providing design and language constructs for modularity, encapsulation, inheritance and polymorphism (Rumbaugh *et al.*, 1991; Jacobson *et al.*, 1992). Some important features/limitations of OOA include:

- i. Emphasis on data rather than procedure.
- ii. Allows us to view a system as a set of interacting objects.
- iii. Divides system into classes, which allows encapsulation and hides implementation details.
- iv. Objects can communicate with each other through functions.
- v. New data and functions can be easily added whenever necessary.
- vi. Enables polymorphism to implement common behavior and interface for related concepts and
- vii. Maintenance of a big size project, with scattered concerns in multiple objects, is not easy.

Although OOP/OOA has met great success in modeling and implementing complex software systems, it has its limitations. Practical experience with large projects has shown that developers may face some problems with maintaining their code because it becomes increasingly difficult to clearly separate concerns such as readability, security, modifiability, etc. into modules. An attempt to do a minor change in the program design may require several updates to a large number of unrelated modules. Aspect-oriented software development (AOSD) is a new paradigm to support separation of concerns in software development (Bergmans *et al.*, 2001; Kiczales *et al.*, 1997; Lieberher *et al.*, 2001; Tarr *et al.*, 1999). The techniques of AOSD make it possible to modularize crosscutting concerns of a system. Like objects in object-oriented software development, aspects in AOSD may arise at any stage of the software life cycle, including requirements

specification, design, implementation, etc. Some examples of crosscutting concerns are exception handling, synchronization, logging, and resource sharing.

Much research and development work has been carried out on software metrics for module-oriented systems and object-oriented systems, but not much work has been carried out on measuring the characteristics and metrics for the emerging aspect-oriented approach (AOA) until now.

## 1.2 ASPECT-ORIENTED SOFTWARE DEVELOPMENT

Programming codes, which cannot be neatly encapsulated in modules/objects but must be scattered throughout the code are the limitation of OOA. Examples of such codes are *exception handling*, *profiling* (determining where a program spends its time), *tracing* (determining what methods are called when), *special security*, *synchronization issues*, etc. To overcome these limitations, the concept of aspect has been introduced. AOSD approach is a relatively new programming paradigm and emerging technology, which has following salient features:

- i. In AOSD, crosscutting concerns are removed from the modules and implemented separately as aspects, which are modular units designed to implement concerns; it is, in fact, encapsulation of crosscutting concerns into logical units.
- ii. An aspect definition may contain some code and instructions about invoking the aspect – as to where, when, and how - in a program.
- iii. AOSD provides a mechanism to weave the aspects with the core modules to form a working system.
- iv. Aspect-Oriented Programs (AOP) helps to overcome the limitation of inefficient way of expressing crosscutting concerns of Object-Oriented Programs (OOP).
- v. This approach implements *reusability*, as aspects can be reused and (vi) By reducing code tangling, it makes it easier to understand what the core functionality of a module is.

### 1.2.1 Aspect Weaver-Composer of the Final System

An “aspect weaver” takes the aspects and the core modules and composes the final system (Aldrich, 2004). With AOP, we start by implementing an OO language (example: Java), and then we deal separately with crosscutting concerns in our code by implementing aspects (Elrad *et al.*, 2001a). Finally, both the code and aspects are combined into a final executable form using an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of methods, modules, or objects, increasing both reusability and maintainability of the code. Figure 1.1 explains the weaving process. The original code need not know about any functionality the aspect has added; it needs only to be recompiled without the aspect to regain the original functionality.

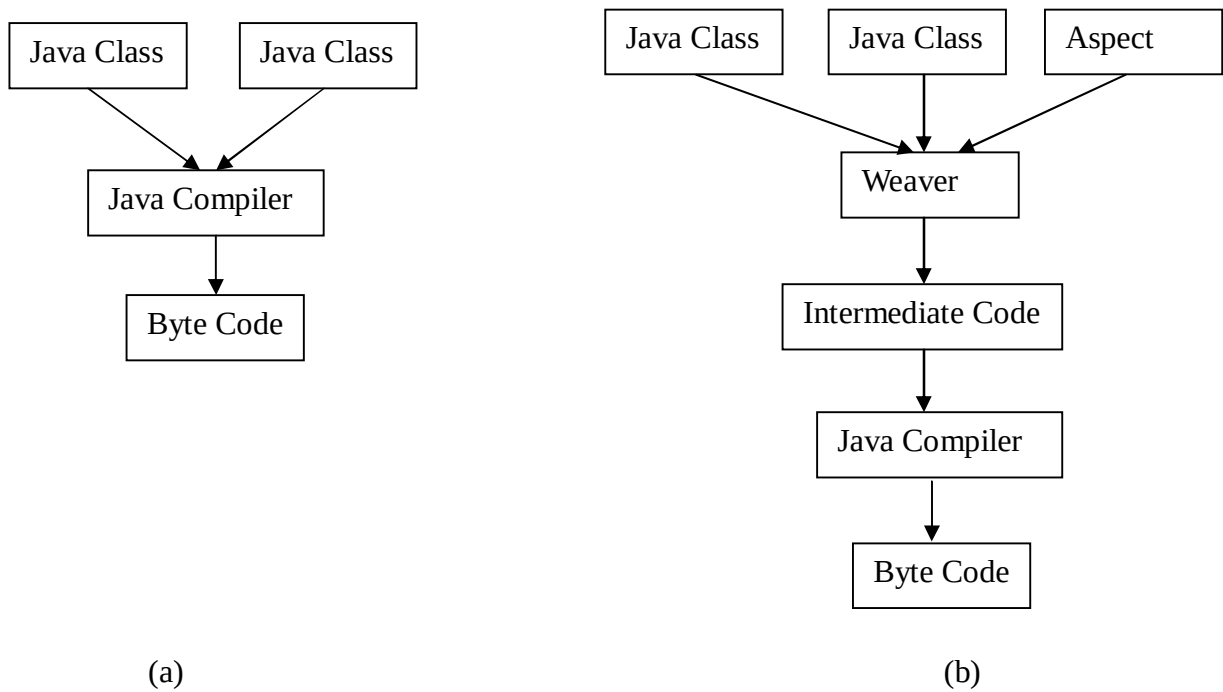


Figure 1.1 Aspect Weaver

In that way, AOP complements object-oriented programming, instead of replacing it, by facilitating another type of modularity that pulls together the widespread implementation of a crosscutting concern into a single unit (aspect).

### 1.2.2 Aspect-Oriented Programming Languages

The aspect-oriented approach to programming not only is applicable for object-oriented systems but also for module-oriented systems. Some of the example programming languages for aspect-oriented systems are AspectC (a C extension), AspectC++ (a C++ extension), AspectXML (a XML extension), AspectL (a LISP extension), AspectJ (a Java extension), CaesarJ and Hyper/J (being used by IBM). Our work is focused on those aspect-oriented programming languages that have new AO features and almost all the features of Java programming language. In this category, we found AspectJ and AspectJ-like (Springs AOP framework, JBoss etc.), CaesarJ and Hyper/J programming languages. Below is a brief introduction to these three languages considered by us.

#### 1.2.2.1 AspectJ

Out of the available AOP languages, AspectJ is the most popular and mostly used in research areas. AspectJ (Gradecki and Lesiecki 2003) is a simple general-purpose extension to Java that provides, through the definition of new constructors, support for modular implementation of crosscutting concerns. It enables plug-and-play implementations of crosscutting concerns. AspectJ has been successfully used for cleanly modularize implementations of crosscutting concerns such as synchronization, consistency checking, protocol management and others.

AspectJ, an aspect-oriented extension to the Java programming language, supports the definition of the following aspects: *join-points*, *pointcuts*, *advice* and *introduction* (Kiczales et al., 2001b) as explained below.

**Join Points:** Join points represent well-defined points in a program's execution. Typical join points in AspectJ include method calls, access to class members, and the

execution of exception handler blocks. Join points may contain other join points. For example, one method call may result in several other method calls before it returns.

**Pointcuts:** A pointcut is a language construct that picks out a set of join points based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards.

**Advice:** An advice is code that executes before, after, or around a join point. One can define advice relative to a pointcut, saying something like "run this code before every method call I want to log."

**Introduction:** An introduction allows aspects to modify the static structure of a program. Using introduction, aspects can add new methods and variables to a class, and declare that a class implements an interface, or convert checked to unchecked exceptions.

#### 1.2.2.2 CaesarJ

CaesarJ (Aracic *et al.*, 2006) is an aspect-oriented language with support for reusability. It does not have aspects as a separate language abstraction. It combines the aspect-oriented constructs, pointcut and advice with advanced object-oriented modularization mechanisms. It supports additional concepts such as *virtual classes*, *mixin composition*, *aspectual polymorphism*, and *bindings*. With *virtual classes*, we can group sets of collaborating classes into a new unit and subclasses of such a unit can refine inherited inner classes. CaesarJ classes can be composed with the operator `&`. The composition operator realizes a variant of multiple inheritance that linearizes the super classes, thereby avoiding ambiguities with respect to method dispatch and with respect to sharing or duplicating inherited state. The order of the *mixin* operands of the operator `&` is important in determining the order of the mixins in the linearized chain. Wrappers and dynamic wrapper section allow defining functionality outside a base class, while retaining subtype polymorphism. Wrapper classes define functionality that is polymorphic with respect to both base object types and aspect types. A family class that implements a component facet by adapting external classes is called *binding*. Concrete

family classes are produced by combining bindings with the families implementing the component functionality to be integrated.

### **1.2.2.3 Hyper/J**

Hyper/J (Pekilis *et al.*, 2002) developed by IBM, is also becoming popular as one of the AOP languages. While using Hyper/J, a developer provides three inputs: a *hyperspace* file that describes the Java class files that can be manipulated by Hyper/J, a *concern mapping* file that describes which pieces of the Java source map to each dimension of concern, and a *hypermodule* file that describes which dimensions of concern should be integrated (i.e., which *hyperslices*) and how that integration should proceed. An individual *aspect* may be viewed as a *hyperslice*, and the set of aspects together with the core classes as a *hypermodule*. There is no central composition rule in AspectJ and CaesarJ. Instead, each aspect contains its part of the rule, specifying how that aspect is to be woven into the base classes.

## **1.3 LITERATURE REVIEW**

This literature review has been divided into four sections 1.3.1 to 1.3.4. Section 1.3.1 deals with research in the field of performance measuring metrics such as cohesion, coupling, complexity, and research in the field of fuzzy logic and implementation of fuzzy logic in automated assessment of complexity of aspect-oriented systems (AOS); section 1.3.2 deals with research in the field of predicting the maintainability of software systems; section 1.3.3 deals with review literature in the field of reusability in AOS; and section 1.3.4 presents a review of quality models for OOS and AOS.

### **1.3.1 Software Metrics**

Software metrics are used as indicators of quality and other characteristics of software systems. Metrics provide a quantitative basis for the development and validation of software development process models. Metrics can be used to improve software productivity and quality. These can also serve as measures of software products for the purpose of comparison, cost estimation, fault prediction, forecasting, guiding decisions

throughout the life cycle, and determining whether software quality improvement initiatives are financially worthwhile (Sedish *et al.*, 2001).

Metrics for aspect-oriented system may be proposed/ defined to measure internal characteristics such as size, cohesion, coupling and complexity, which can be evaluated directly from the application source code. These may also be proposed to determine external quality characteristics such as maintainability, understandability, reusability, readability and modularity, which are indirect measurable characteristics and may be accessed through metrics for internal characteristics.

### 1.3.1.1 Metrics for Internal Characteristics for Legacy Systems

Modules of a high quality software design should obey the principle of high cohesion. Cohesion is defined as a measure of the degree to which the elements of a module belong together (Stevens *et al.*, 1974).

Eder *et al.* (1994) proposed a framework aimed at providing comprehensive, quantitative criteria for cohesion in object-oriented systems. They adapted existing frameworks for cohesion in procedural and object-based paradigms. They divided cohesion in three categories: method, class and inheritance cohesion. Chidamber and Kemerer (1991, 1994) defined a Lack of Cohesion in Methods (LCOM) metric, which is an inverse of cohesion measure. A high value of LCOM indicates low cohesion and vice versa. LCOM is defined as the number of disjoint sets formed by the intersection of the  $n$  sets. Hitz and Montazeri (1995) proposed cohesion measures based on Chidamber and Kemerer as follows:

Let  $X$  denote a class,  $I_X$  the set of its instance variables, and  $M_X$  the set of its methods. Consider a simple, undirected graph  $G_X (V, E)$  with  $V = M_X$  and  $E = \{(m, n) \in V \times V \mid \exists i \in I_X : (m \text{ accesses } i) \wedge (n \text{ accesses } i)\}$ . LCOM is then defined as the number of connected components of  $G_X$ .

Henderson- Seller (1996) defined a cohesion measure having the following properties: (i) the measure yields 0, if each method of the class references every attribute

of the class, (ii) The measure yields 1, if every method of the class references only a single attribute, and (iii) a value between 0 and 1 are to be interpreted as a percentage of the perfect value. Lee *et al.* (1995) proposed a set of cohesion measures based on information flow through method invocations within a class. For a method *m* implemented in class *C*, the cohesion of *m* is the number of invocations to other methods implemented in class *C*, weighted by the number of parameters of the invoked methods.

Coupling is another important internal characteristic, which is defined as the measure of the strength of association established by a connection from one module to another (Stevens et al., 1974).

Eder *et al.* (1994) defined coupling on the basis of three types of relationships: interaction relationship between methods, component relationship, and inheritance relationship between classes. From these relationships, three type of coupling are derived: interaction coupling, component coupling and inheritance coupling. Hitz and Montazeri (1995) divided coupling into two broad categories, class level coupling (CLC) and object level coupling (OLC). CLC represents coupling resulting from state dependencies between two classes in a system during development lifecycle. OLC represents the coupling resulting from state dependencies between two objects during the run time of the system. Briand *et al.* (1997) defined three different types of coupling that cause different types of interaction between modules/ classes. These couplings are as follows: coupling due to class attribute interaction, coupling due to class method interaction, and coupling due to method-method interaction.

Software complexity is one of the important internal characteristics and can be defined as difficulty to analyze source code, modify and maintain software modules. It is one of the branches of software metrics that is focused on direct measurement of software attributes. A complexity measure can be used to predict critical information about reliability, maintainability and reusability of software systems from automatic analysis of source code. Researchers have proposed many metrics for measuring various aspects of complexity, such as size, control flow, data structures, inter-module structure, and number

of variable definition/ usage associations. A brief description of some of the complexity metrics for legacy system and aspect-oriented systems are as follows:

McCabe (1976) proposed the software metric Cyclomatic Complexity (CC), which directly measures the number of linearly independent paths through a program's source code. It is computed using the program's control flow graph. Kafura and Henry (1981) proposed the complexity metrics based on the number of local information- flows entering (fan-in) and exiting (fan-out) in each procedure. Halstead complexity metrics (Halstead, 1977) are computed statically, without program execution. These measures are based on four scalar numbers: the number of direct operators, the number of direct operands, the total number of operators, and the total number of operands, counted directly from the source code. Weyuker (1988) defined some properties for evaluating software complexity measures. These properties are used to evaluate the robustness of the measure and, in turn, lead to the definition of good notions of software complexity. These properties were defined when procedural programming approach was in use. With the help of these properties, researchers can define better proposals for complexity measurement for other programming approaches.

#### **1.3.1.2 Metrics for Internal Characteristics for Aspect-Oriented Systems**

There are many assessment framework and measurement approaches for cohesion measurement in OO Systems, which are due to high maturity level of OO paradigms. AOP is a very new and emerging paradigm. In literature, there exist few assessment framework/measurement approach/metrics (Sant'Anna *et al.*, 2003; Zhao *et al.*, 2004; Ceccato and Tonella, 2004; Gelinas *et al.*, 2006; Anna *et al.*, 2007) to measure cohesion in AO systems and most of the measures are for AspectJ programming language. Since then, many other languages, such as CaesarJ, Hyper/J have also been designed and used in various applications.

Sant'Anna *et al.* (2003) proposed a new metric 'Lack of Cohesion in Operations' (LCOO), which measures the amount of method/advice pairs that do not access to the same instance variables. It is an extension of the well-known Lack of Cohesion in

Methods (LCOM) metric developed by Chidamber and Kemerer (1994). This metric measures the lack of cohesion of a component (class and aspect). A high LCOO value, according to Sant'Anna *et al.* (2003), indicates disparateness in the functionality provided by the aspect. The definition of LCOO metric is almost operational. It suffers from several problems as stated, among others, by Henderson-Sellers (1996). Again, definition of this metric is based on AspectJ-like programming languages.

Sant'Anna *et al.* (2003) also defined a new metrics suite for concern-driven architecture. The proposed metric suite is for measuring cohesion, coupling and complexity of AO and non-AO systems. The proposed metric Lack of Concern-based Cohesion (LCC) for measuring cohesion in concern-driven architecture counts the number of concerns addressed by the assessed component. This is very useful framework for evaluating internal software quality attributes such as cohesion, coupling, complexity and modularity at architecture level. It is not specified that how to quantify aspect cohesion for empirical evaluation. It is also not clear whether inheritance has been accounted or not.

Zhao and Xu (2004) defined cohesion measures for aspect-oriented systems, which are based on a dependency model for aspect-oriented software that consists of a group of dependency graphs. Zhao and Xu (2004) defined cohesion as the degree of relatedness between attributes and modules (method/advice). They presented two ways for measuring aspect cohesion based on inter-modules and module-attribute dependencies. The first way suggests that each measurement works as a field. The second way suggested for cohesion measurement, each facet can be integrated as a whole with parameters. In general, this approach suggests a complex way to measure aspect cohesion that may be problematic to use in a real development context and particularly in the case of scalable aspect-oriented software.

Gelinas *et al.* (2006) proposed metric ACoh to measure aspect cohesion. This cohesion measurement is based on dependency analysis. They defined this metric using two aspect cohesion criteria, *Modules-Data Connection Criterion* and *Modules-Modules Connection Criterion*. Two modules can be connected in many ways: by sharing

attributes or by sharing modules or both. They defined a new metric for aspect cohesion measures:  $ACoh(Aspect_i) = NC(Aspect_i) / NM(Aspect_i) \in [0, 1]$ , where  $NC(Aspect_i)$  is the total number of modules pairs in an aspect and  $NM(Aspect_i)$  is the maximum number of connections between aspect modules.

Kumar *et al.* (2008a) proposed a cohesion metric for generic/ unified aspect-oriented systems. Here, generic means applicable to most of the AOP languages. Their framework is defined for Java, AspectJ and CaesarJ languages. In order to represent elements of different programming languages as common and unambiguous names, they defined new terminology for AOS. They identified six different types of connections that cause cohesion. Using these connections and framework criteria, they derived cohesion metric from Gelinas *et al.* (2006) and defined Unified Aspect Cohesion as  $UACoh(c_i) = ANC(c_i) / MNC(c_i) \in [0, 1]$ , where  $ANC$  is actual number of connections and  $MNC$  is maximum number of possible connections. Further, they extended their work in Kumar *et al.* (2010) and correlate cohesion metric values with changeability metric for unified aspect-oriented system and authors came to a conclusion that cohesion metric cannot be used as indicator for assessing changeability of the AOS.

Zhao (2004) proposed a coupling measure suite that implements the coupling framework for AspectJ-like programming languages. His coupling measures are defined on the number of dependencies between aspect and some classes, i.e. *attribute-class*, *module-class*, *module-method* and *aspect-inheritance* dependencies. He also discussed the mathematical properties of these coupling measures, in which he showed that these measures satisfy properties that a good coupling measure should have. The coupling measures proposed by Zhao (2004) focused only on dependencies between aspect and classes, and did not consider dependencies between aspects or between classes into account.

Ceccato and Tonella (2004) proposed coupling metrics for AOP languages. They represented *classes* and *aspects* as *module* and term *operation* subsumes class *methods* and aspect *advices/introductions*. They defined six different coupling metrics as:

- i. Coupling on Advice Execution (CAE) counts aspects containing advices possibly triggered by the execution of operations in a given module.
- ii. Coupling on Intercepted Module (CIM) counts modules or interfaces explicitly named in the pointcuts belonging to a given aspect.
- iii. Coupling on Method Calls (CMC) counts modules or interfaces declaring methods that are possibly called by a given module.
- iv. Coupling on Field Access (CFA) counts modules or interfaces declaring fields that are accessed by a given module.
- v. Response for a Module (RFM) counts operations potentially executed in response to a message received by a given module, and
- vi. Crosscutting Degree of an Aspect (CDA) counts modules affected by the pointcuts and by the introductions in a given aspect.

Bortsch and Harison (2006) suggested a coupling framework for AspectJ, which is an extension of a coupling framework for OO systems (Briand *et al.*, 1999). The extended framework contains a specific definition of the different coupling mechanisms found in AspectJ and an additional criterion instantiation. These criteria are *types of connection*, *locus of impact*, *granularity*, *stability of server*, *direct or indirect connections*, *inheritance* and *instantiation*. The authors validated existing coupling metrics theoretically for AOP language using these framework criteria. They concluded that the definition of coupling measures is a difficult task that needs a great deal of rigor if the measures are to be theoretically valid and well defined.

Bartolomei *et al.* (2006) proposed a unified coupling framework for AOP. In order to consistently and unambiguously define coupling measurements, they defined new terminologies for representing members of AO systems. In their framework, they considered two AOP languages, AspectJ and CaesarJ. They have extended their framework from Briand's frameworks (Briand *et al.*, 1999; Arisholm *et al.*, 2004) and tried to maximize the use of the terminology and coupling criteria originally defined in

Briand's frameworks. They have shown how the framework can be instantiated for Java, AspectJ and CaesarJ. Also, they demonstrated the applicability of the framework by using it on an existing coupling metric. In their study, they have not proposed any new coupling metric.

Kumar *et al.* (2009) extended their framework for coupling measures from Bartolomei *et al.* (2006), Briand *et al.* (1999) and Arisholm *et al.* (2004) frameworks and have proposed new coupling metrics for generic AO systems. They also included Hyper/J besides Java, AspectJ and CaesarJ in their framework. They identified the connections that cause coupling in aspect-oriented systems and defined six different coupling metrics as:

- i. Coupling on Attribute Type (CoAT)
- ii. Coupling on Parameter Type (CoPT)
- iii. Coupling on Attribute Reference (CoAR)
- iv. Coupling on Operation Invocation (CoOI)
- v. Coupling on Inheritance (CoI) and
- vi. Coupling on High Level Association (CoHA).

Researchers have also suggested many approaches to assess complexity of AOS. Dospisil (2003) measured code complexity in projects designed in AspectJ. He has defined and used entropy metrics for ordering of symbols to estimate the complexity of AspectJ based programs. The entropy metrics are useful in ranking different modules and symbols with regard to their complexity. He introduced weighted entropy values to accommodate the subjective perspective of an observer. His approach provides a multi valued space more suitable for prediction models. This framework is applicable only for AspectJ-like languages.

Sicilia *et al.* (2006) talked about the main design and implementation issues of aspect-oriented design (AOD)-based extensions on OJB database libraries using fuzzy

logic. They specified that fuzziness can be considered as separate crosscutting concern in existing software, and in consequence, AOD techniques provide a convenient framework to implement fuzzy extensions to existing libraries.

Pataki *et al.* (2006) described a multi-paradigm metrics which is extended for aspect-oriented programs. The metric can measure complexity of Module-Oriented (MO), OO and AO parts of programs implemented in AspectJ. This extended metric revealed that an aspect-oriented approach does not necessarily reduce the complexity on its own but the gain highly depends on the actual problems.

Xia *et al.* (2008) described a new way of assigning complexity weight values to function point metric. They discussed the concepts of calibrating Function Points, whose aims are (i) to estimate a more accurate software size that fits for specific software application, (ii) to reflect software industry trend, and to improve software projects' cost estimation. In their paper, they proposed, a Function Point calibration model called Neuro-Fuzzy Function Point Calibration Model (NFFPCM) that integrates the learning ability from neural network and the ability to capture human knowledge from fuzzy logic. The empirical validation using International Software Benchmarking Standards Group (ISBSG) data repository release 8 shows a 22% accuracy improvement of mean magnitude relative error (MMRE) in software effort estimation after calibration. These weight values are assigned to functions (operations) only, and not to other members of the class and this framework is basically for object-oriented systems.

Kumar *et al.* (2008b) proposed complexity metrics for generic/ unified aspect-oriented systems that are applicable to Java, AspectJ, CaesarJ and Hyper/J programming languages. They divided complexity of AOS into two categories as code complexity and interaction complexity. The code complexity of an AO component (class/aspect) is due to the complexity of the attributes, operations (advice/ methods), and nested components present in the component. Interaction complexity is due to the operation invocations, attribute references, and executing statements that cause interaction between AO components such as exception occurrence, array creation and applying typecasting. In order to automate complexity evaluation, they extended this work and applied fuzzy logic

approach in Kumar *et al.* (2010a) to measure code complexity of generic aspect-oriented systems.

### **1.3.2 Software Maintainability**

Software maintenance is the modification of the software product/project after delivery. The purpose of maintenance is to correct faults, to improve performance or other attributes, or to adapt product/project to a modified environment. Maintainability is an important pre-requisite for reusable aspects and aspect-oriented systems because crosscutting concerns are very difficult to change, but in aspect-oriented system, these concerns can be changed easily. Therefore, maintainability of a software system should be considered as one of the most important quality factors. Researchers have proposed many methodologies to evaluate changeability and maintainability of legacy systems as well as of aspect-oriented systems. Some of them are as follows:

#### **1.3.2.1 Maintainability of Legacy Systems**

Li and Offutt (1996) proposed algorithms for calculating the complete impact of changes made in a given class. They explored the effects of encapsulation, inheritance, and polymorphism. Chaumon *et al.* (2002) proposed a change impact model for changeability assessment in object-oriented software systems, which was applied to programs developed in C++. In their work, for each of the possible change identified in C++, the impact was calculated so that necessary actions may be taken to ensure a successful system compilation after change implementation. Hsia *et al.* (2006) studied the effect of architecture on maintainability. They measured maintainability and its relationship to architecture, especially broadness of the architecture trees. As a result, they found that maintainability is better for systems with broader trees.

#### **1.3.2.2 Maintainability of Aspect-Oriented Systems**

Zhao (2002) presented an approach to support change impact analysis of aspect-oriented software based on *program slicing technique*. He performed changes in source code and assessed change impact analysis to other modules of AO software. All

definitions in his approach to measure change impact are in terms of AspectJ. There are some unified/generic frameworks for AO systems. Li *et al.* (2006) explored the benefits of using AOP in component-based systems, especially in COTS-based development. As per their study, use of AOP does not ensure that less ‘lines of code’ need to be modified when adding or replacing COTS component. It will depend on the type of system (homogenous or heterogeneous). When cross-cutting concerns are heterogeneous, application of AOP in component-based system may not bring benefits.

Kumar *et al.* (2007) proposed a methodology to measure maintainability of AspectJ based aspect-oriented systems. They defined possible syntactical changes in AspectJ. Using these possible changes, they performed experiment on projects designed in Java and AspectJ and proved that AspectJ projects will absorb changes more easily compared to Java projects. Further, they extended this work for generic aspect-oriented systems (Grover *et al.*, 2008) and proposed a changeability metric for generic aspect-oriented systems. Changeability is mentioned as sub-characteristic of maintainability in ISO 9126 (ISO, 2001). They performed experiment on two different projects designed in AspectJ and CaesarJ and evaluated average change impact by making random changes at code level. In their study, high change impact means low changeability. They validated their approach by one of the existing approach. Using their proposed metric for changeability, maintainability of AOS can be predicted.

### **1.3.3 Software Reuse and Reusability**

Software reuse is the process of building new software from existing software, rather than starting from scratch. One of the advantages claimed of object-oriented technology is that it encourages better design (encapsulation) that can be reused in a more reliable way, as there is exact knowledge of which operations access which data. Inheritance allows a developer to reuse a class whose behavior provides some of the behavior required for the new class so that only the different parts need to be designed. Reusability is the degree to which a module can be used again to add new functionalities with slight or no modification and reduces software development cost by enabling less coding and more integration.

### 1.3.3.1 Reusability of Legacy Systems

Price *et al.* (1997) presented a framework for reusability measurements, which facilitates large-scale object-oriented design reuse. Their work contributes:

- i. an effective and subjective method to distinguish between what is expected to be reused and what is not
- ii. an effective and subjective method to group related components into reusable portions, and
- iii. specific metrics based on the above concepts, which can work at any level of design detail.

This approach relies heavily on subjective decisions and it requires the software designers to have a good understanding about the application domain and the type of systems that they expect to build in future.

Barnard *et al.* (1998) proposed reusability metric for object-oriented software. Their metric is based on empirical evidence taken from well used and accepted programming libraries in which the classes are assumed to be highly reusable. One of the findings in their experiments was that a class can be regarded as a black box; its contents are irrelevant to its reusability. So long as the class is correct and robust, the re-user is only interested in its documentation and interface.

Dandashi *et al.* (2002) proposed an assessment of reusability of C++ object-oriented code by measures of indirect quality attributes such as adaptability, completeness, maintainability, and understandability. They measured these indirect quality attributes by performing correlation analysis between these indirect quality attributes and direct measurable metrics for internal characteristics such as McCabe's Complexity, Halstead's Volume, the number of Physical Source Statements (PSS), the Depth of nested statements, Weighted Methods per Class (WMC), Depth of Inheritance tree, Number of Children (NOC), Response for a class (RFC), Coupling Between Objects (CBO). Their reusability model is helpful in assessing and comparing reusability level of projects designed in C++.

Aggarwal *et al.* (2005c) proposed a set of two object oriented code metrics, which are evaluated on the basis of Weyuker's properties. Their experimental proposal suggests

that developers should proactively use a template mechanism and their metrics suite can aid project managers to analyze how much reusability is incorporated in the coding process. They have also shown that more amounts of testing is required to test functions/classes using templates, simultaneously increasing reuse of software components. Another benefit of using these metrics is reduction in lines of code via using more generic programming in the form of function and class templates.

### **1.3.3.2 Reusability of Aspect-Oriented Systems**

As per available literature, there is no complete reusability assessment framework/model that has considered all features of aspect-oriented technology. Most of the research is comparative study such as object-oriented reusability with and without aspect implementation, reusability of connector with and without aspect implementation, reusability of CBSD with and without aspect implementation. Sant'Anna *et al.* (2003) have proposed a framework for the assessment of reusability and maintainability of AO system, which is incomplete. The following is brief description of some AO reusability measures:

Sant'Anna *et al.* (2003) suggested a framework for the reuse and maintenance of aspect-oriented software. They proposed a model, which specifies the relationship of reusability and maintainability with metrics for internal characteristics such as cohesion, coupling, size and crosscutting concerns. Using this proposed model, reusability and maintainability can be evaluated by these internal characteristics metrics.

Cunha *et al.* (2006) presented a collection of well-known high level concurrency patterns and mechanisms coded in AspectJ. With the help of some examples, they discussed benefits in terms of reusability, modularity, understandability and unplug-ability, while applying AO approach in Java programs. From their study, they concluded that the above mentioned quality characteristics will improve, while applying AO approach.

Connectors are becoming popular and are more significant for software architecture. Zhang *et al.* (2008) incorporated AO technique in the reusable design and implementations of connectors. They explored the affect of crosscutting concerns in reusability of connectors. By introducing AO technique in these connectors, reusability at the architectural level has been improved.

ParaAJ is a new programming language that is an extension of AspectJ (Aljasser *et al.*, 2009). This language has the ability to parameterize aspects. The authors claimed that using parameters is a step further towards modularizing aspects and enhancement in reusability.  $\beta$ -version of this language has been released, hence, it is neither tested and nor validated.

#### **1.3.4 Software Quality Models**

There are many software quality models that suggest ways to together different quality characteristics. Each model helps in understanding how several quality factors contribute to the overall quality. In order to evaluate the overall quality of the software product, many researchers and practitioners have proposed quality models to relate user's external views to the developer's internal views for the software products. A brief description of some of the popular quality models is given below.

##### **1.3.4.1 Quality Models for legacy Systems**

Boehm *et al.* (1976) represented their quality model as a hierarchical tree and broke quality characteristics into sub-characteristics. They also included the hardware related characteristics in their model. Their model is helpful in evaluating the overall quality of a software system due to hierarchical structure of this model. But there are no guidelines to get individual values of quality characteristics and sub-characteristics.

McCall *et al.* (1977) classified software quality factors into three broad categories:

- i. Product Operation Factors
- ii. Product Revision Factors, and

### iii. Product Transition Factors.

Product Operation Factors are correctness, reliability, efficiency, integrity, and usability, which contribute to product operation. Product Revision Factors are maintainability, flexibility, and testability, which contribute to product revision. And Product Transition Factors are portability, reusability, and interoperability, which contribute to product transition.

Dromey (1995) proposed a quality model that establishes the link between tangible product characteristics and less tangible attributes. This model helps in predicting defects and also indicates the properties that will need to be violated to create defects. The model supports making of quality into software, definition of language-specific coding standards, systematic classification of quality defects, and the development of automated code auditors for detecting software quality defects. Reusability is also considered as one of the quality characteristics in this model.

Deutsch *et al.* (1998) classified software quality into two categories:

- i. software procedure quality, and
- ii. software product quality.

Software procedure quality consists of software engineering related elements such as technology, tools, personnel, organization, and equipment. Software product quality consists of characteristics such as document clarity, integrity, design trace-ability, organization, program reliability, and test integrity.

Ward *et al.* (1999) suggested that software quality measures may include one or more of the followings:

- i. User-based software quality refers to the degree of satisfaction of user expectations
- ii. Product delivery-based software quality refers to the degree of system effectiveness and program maintainability
- iii. Manufacturing-based software quality emphasizing quality control and

management, and

- iv. Organizational control-based software quality refers to project costs, production time, resource control and risk management.

The ISO/IEC -9126 (2001) series standards have introduced a hierarchical model with six major quality characteristics. These quality characteristics are divided into 21 sub-characteristics, which contribute to internal qualities. Though the ISO/IEC 9126 quality model is fairly elaborative, but it does not cover some important quality characteristics that contribute to the quality of the AOP.

Sharma *et al.* (2008) proposed a quality model for Component-Based Software Development (CBSD) which is an extension of ISO/IEC 9126. In their model, they included *reusability*, *complexity*, *track-ability*, and *flexibility* as new sub-characteristics in the six characteristics of ISO/IEC 9126. In order to evaluate total quality of a component, they used Analytic Hierarchy Process (AHP).

Chang *et al.* (2008) provided guidelines to evaluate software quality by integrating fuzzy theory and AHP. They applied this quality assessment approach on ISO/IEC 9126 quality model. Instead of taking the mean of collected sample data, they applied fuzzy theory to get relative weights of characteristics and sub-characteristics. They have not proposed any new quality model, but provided guidelines to evaluate the software quality.

#### **1.3.4.2 Quality Models for Aspect-Oriented Systems**

Kumar *et al.* (2009b) proposed an Aspect-Oriented Software Quality Model (AOSQUAMO), which is extension of the ISO/IEC 9126 (2001) software quality model. They have added four new sub-characteristics, *modularity*, *code-reusability*, *complexity*, and *reusability* under the main characteristics maintainability, efficiency, usability, and functionality respectively. They redefined existing characteristics of ISO/IEC 9126 in terms of AOS and highlighted the importance of the new added characteristics in AOS. In order to evaluate the quality of proposed model as a single parameter, they applied AHP.

Using their quality measurement approach, two or more projects of AOS could be compared in terms of quality.

## **1.4 CONCLUSION**

In order to assess various quality characteristics such as maintainability, reusability, modularity, complexity, and overall quality of aspect-oriented systems, researchers and practitioners have proposed different methodologies. From review of the literature, it is evident that there exist no precise criteria to measure these quality characteristics. Most of the work is either proposed theoretically without any quantitative evaluation, or without valid methodology of validation. In order to obtain overall quality of AOS as single parameter, relationships among these characteristics have not been explored.

The proposed study aims to assess quality characteristics of aspect-oriented systems. The work proposes metrics for internal characteristics such as cohesion, coupling, complexity and size. It proposes a reusability assessment model by using these internal characteristics metrics for AOS. In order to predict maintainability, a changeability metric has been proposed and validated. It also proposes the evaluation of the quality of AOS as a single parameter. These estimates will provide guidelines to develop product with less effort and will lead to develop good quality product. The study uses Analytic Hierarchy Process and Fuzzy Logic based approaches to assess these measures.





# CHAPTER 2

## METRICS SUITE FOR GENERIC ASPECT-ORIENTED SYSTEMS\*

---

### 2.1 INTRODUCTION

Now-a-days, our society is becoming dependent on software. That is why demand for quality software is increasing day by day. In the literature of software quality models, many researchers and practitioners have proposed their quality models such as McCall *et al.* (1977), Boehm *et al.* (1976), FURPS (Grady, 1992), ISO/IEC 9126 (ISO 2001), Dromey (1995) and Chulani *et al.* (1999). These models are intended to evaluate external software qualities such as *maintainability, usability, efficiency, functionality, reliability, portability, and reusability*. These external software quality characteristics can be measured with the help of software metrics. Metrics are designed on the basis of design structure of programming languages such as Module-Oriented Programming (MOP), Object-Oriented Programming (OOP) and Aspect-Oriented Programming (AOP) (Kumar *et al.*, 2006). Design of these metrics depends on internal quality characteristics such as *encapsulation, cohesion, coupling* and *complexity* of the software system. In order to

**\*The major findings of the chapter has been published in: (i) IEEE proceedings of ASWEC'08, pp: 57-65. (ii) IEEE proceedings of CSSE'08, pp: 98-103, and (iii) ACM SIGSOFT SEN, 34(3), pp: 1-6, May-2009.**

determine internal characteristics of software systems, researchers and practitioners have proposed their metrics and assessment frameworks, for quality design principles like cohesion (Briand *et al.*, 1998; Zhao *et al.*, 2004; Gelinas *et al.*, 2006; Kumar *et al.*, 2008a; Kumar *et al.*, 2010b;), coupling (Briand *et al.*, 1999; Arisholm *et al.*, 2004; Zhao, 2004; Bartolomei *et al.*, 2006; Bartsch *et al.*, 2006; Kumar *et al.*, 2009a), and complexity (Harrison, 1992; Davis *et al.*, 1988; Weyuker, 1988; Bansiya *et al.*, 1999). For any software system, high quality means, high cohesion, low coupling and low complexity.

Aspect-oriented programming (AOP) languages aim to improve the ability of designers to modularize concerns that cannot be modularized using traditional module-oriented (MO) or object-oriented (OO) paradigms. Such concerns are scattered in multiple modules (classes) and are known as *crosscutting concerns*. Examples of crosscutting concerns include *logging*, *tracing*, *caching*, *resource pooling* etc. The ability to modularize such concerns is expected to improve comprehensibility, parallel development, reuse and ease of change (Kumar *et al.*, 2006b; Kumar *et al.*, 2007), reducing development costs, increasing dependability and adaptability. Since AO is a new abstraction, the definition of cohesion, coupling, complexity and size metrics is required to redefine in the context of AOP.

Out of all available AOP languages, the most popular is AspectJ (Kiczales *et al.*, 2001a). AspectJ is an extension of Java with several complementary mechanism, namely *join points (JPs)*, *Pointcut Descriptors (PCDs)*, *advice*, *introduction* and *aspect*. JPs represent well-defined points in a program's execution. Typical *join points* in AspectJ include method calls, access to class members, and the execution of exception handler blocks. A PCD is a language construct that picks a set of *join points* based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards. *Advice* is a code that executes *before*, *after*, or *around* a join point. Advice is defined with respect to a *pointcut*, saying something like "run this code before every method call I want to log". *Introduction* allows aspects to modify the static structure of a program. Using introduction, aspects can add new methods and variables to a class. It can also declare that a class implements an interface, or convert checked to unchecked

exceptions. Advice, pointcuts, ordinary data members and methods are grouped into class-like modules called *aspects*. Aspects are intended to support the modular representation of crosscutting concerns (Garcia *et al.*, 2004), although they admit other uses. Some existing AOP languages and frameworks provide a very similar composition model to the AspectJ such as Springs AOP framework (Johnson, 2003) and JBoss AOP (JBoss, 2004). Most of the research work done for measuring internal characteristics of AOS is for AspectJ. Characteristics of other aspect-oriented programming languages namely AspectC, AspectC++, AspectXML, CaesarJ and Hyper/J are fairly unexplored.

Our work is focused on those aspect-oriented programming languages which have new AO features and most of the features of Java programming language. In this category, we found AspectJ-like (AspectJ, Springs AOP framework, JBoss etc.) , CaesarJ and Hyper/J programming languages. CaesarJ (Aracic *et al.*, 2006) does not have aspects as a separate language abstraction. It supports additional concepts such as *virtual classes*, *mixin composition*, *aspectual polymorphism*, and *bindings*. Hyper/J (Pekilis *et al.*, 2002) is also becoming popular as one of the AOP languages developed by IBM. While using Hyper/J, a developer provides three inputs: a *hyperspace* file that describes the Java class files that can be manipulated by Hyper/J, a *concern mapping* file that describes which pieces of the Java source map to each dimension of concern, and a *hypermodule* file that describes which dimensions of concern should be integrated (i.e., which *hyperslices*) and how that integration should proceed. Individual *aspect* may be viewed as *hyperslice*, and the set of aspects together with the core classes as *hypermodule*. There is no central composition rule in AspectJ and CaesarJ. Instead, each aspect contains its part of the rule specifying how that aspect is to be woven into the base classes.

In this chapter, we have proposed a metric suite for AOS, which is applicable to those AOP languages which have features of Java programming language and AO technology. In order to present standardized and unambiguous representation of components/members of different AOP languages, we have defined new terminologies. For definition of metrics for internal characteristics, cohesion, coupling, and complexity;

unified frameworks have been defined. Usefulness of these proposed metrics have been shown in rest of the chapters.

## 2.2 TERMINOLOGY AND FORMALISM

In order to define metrics for cohesion, coupling, complexity and size of generic AOS, it is necessary to establish a proper standardized, unambiguous and operational terminology and formalism. Briand *et al.* (1998) and Briand *et al.* (1999) introduced a terminology for object oriented (OO) systems based on set and graph theory. Bartolomei *et al.* (2006) defined terminology and formalism for AO systems which include Java, AspectJ-like languages and CaesarJ family languages as part of unified framework. Bartolomei's framework is an extension of Briand *et al.* (1999)'s framework and terminology in this framework has not properly been defined. In this section we introduce the terminology and formalism foundation for structural elements in AO systems. The presented terminology is for Java, AspectJ-like, CaesarJ and Hyper/J programming languages. These terminologies have been derived from Briand *et al.* (1998), Briand *et al.* (1999) and Bartolomei *et al.* (2006).

### 2.2.1 Structure

Structure of an Aspect-Oriented (AO) system is a set of fundamental elements which defines its key abstractions for module specifications. Figure 2.1 (Bartolomei *et al.*, 2006) presents a detailed view of the structural abstractions defined for a system. Each type of abstraction is alternatively called an element.

An AO system  $S$  consists of a set of components, denoted by  $C(S)$ . A component  $c$  consists of a set of attributes,  $Att(c)$ , a set of operations,  $Op(c)$  and a set of nested components,  $Nested(c)$ . In other words, a component is composed of sub-components in addition to its internal attributes and operations. The set of members of a component  $c$  is defined by  $M(c) = Att(c) \cup Op(c) \cup Nested(c)$ . For generality purposes, a component is a unified Figure 2.1, structure of an Aspect-Oriented System abstraction consists of both aspectual and non-aspectual modules, for example: (i) a component represents either a class or an interface in Java programs, (ii) a component is a class, or an interface, or an aspect in AspectJ programs, (iii) a component is a cclass as defined in CaesarJ, or a

conventional class, or an interface in CaesarJ programs, and (iv) a component is a hyperslice as defined in Hyper/J, or a conventional class, or an interface in Hyper/J programs. In Hyper/J, hypermodule is a set of components  $C(S)$ . An operation  $o$  consists of, a set of code,  $Code(o)$ , a set of parameters,  $Par(o)$ , and a return type,  $Rt(o)$ . An attribute consists of a set of code,  $Code(a)$ . Each piece of code provides a set of join points,  $JP(cod)$ .

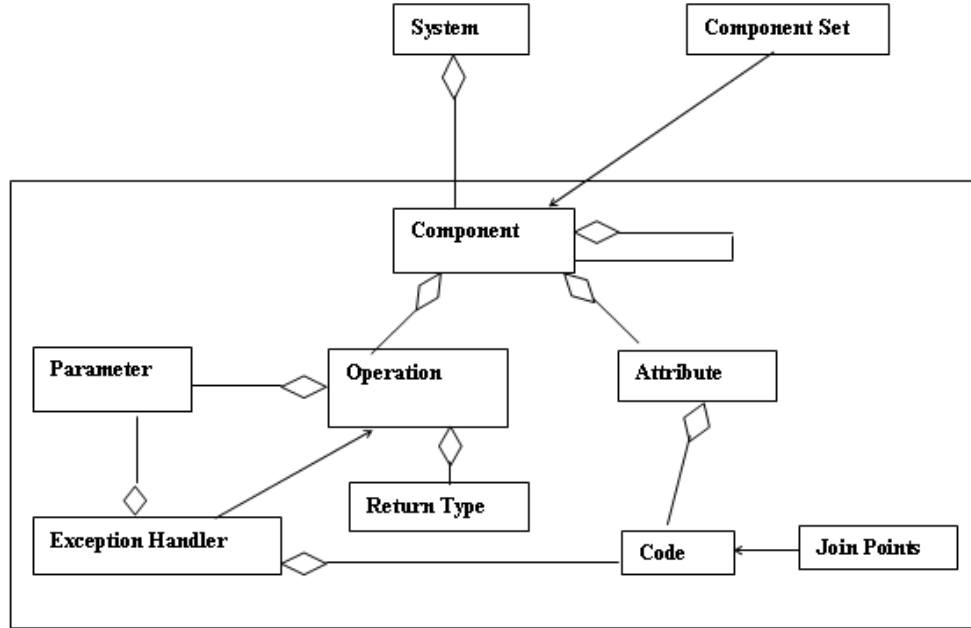


**Figure 2.1 Components of an Aspect-Oriented System**

## 2.2.2 System

**Definition 1:** An AO system  $S$  consists of a set of components denoted by  $C(S)$ .

## 2.2.3 Components

**Definition 2:** A component  $c$  consists of attributes,  $Att(c)$ , a set of operations,  $Op(c)$ , and a set of nested components,  $Nested(c)$ . The set of members of a component  $c$  is defined as:

$$M(c) = Att(c) \cup Op(c) \cup Nested(c).$$

This abstract structure can be instantiated for different languages. In the present work, we have considered Java (*J*), AspectJ (*AJ*), CaesarJ (*CJ*) and Hyper/J (*HJ*) programming languages. Abstractions that map to all languages are marked with an *All*. Aspects and Inter-type declarations are only valid for AspectJ-like languages, CClasses are defined only in CaesarJ and Hyperslices are defined only in Hyper/J. So, component is Class (*All*), Interface (*All*), Aspect (*AJ*), CClass (*CJ*) and Hyperslice (*HJ*). There may be inheritance relationship between components, as such for each component  $c \in C$ , let

- $Parents(c) \subset C$  be the set of parent components of component  $c$ .
- $Children(c) \subset C$  be the set of children components of component  $c$ .
- $Ancestors(c) \subset C$  be the set of ancestor components of component  $c$ .
- $Descendents(c) \subset C$  be the set of descendent components of component  $c$ .

Because of the inheritance relationships between components, the following member sets are defined for a component  $c$ :

- $M_{NEW}(c)$  - Members newly implemented in the component (not overridden).
- $M_{VIR}(c)$  - Members declared as virtual in the component.
- $M_{OVR}(c)$  - Members inherited and overridden.
- $M_{INH}(c)$  - Members inherited and not-overridden.

## 2.2.4 Operations

**Definition 3:** An *operation*  $o$  consists of a set of code,  $Code(o)$ , a set of parameters,  $Par(o)$ , and a return type,  $Rt(o)$ . The set of members of an operation  $o$  is defined as:

$$M(o) = Code(o) \cup Par(o) \cup Rt(o).$$

For this abstract structure, an operation is Method (*All*), Constructor (*All*), Static Initializer (*All*), Intertype Method (*AJ*), Intertype Constructor (*AJ*), Intertype Attribute (*AJ*), Pointcut (*AJ*, *CJ*, *HJ*), Advice (*AJ*, *CJ*), Declare Statements (*AJ*, *CJ*, *HJ*), Wrapper Constructor(*CJ*), and Composition rules(*HJ*).

The operation  $o$  can be further divided into following categories:

**Definition 4:** *Declared and Implemented operations*

For each component  $c \in C$ , let,  $Op(c)$  be the set of operations of component  $c$ .

- $Op_D(c) \subseteq Op(c)$  be the set of operations *declared* in  $c$ , i.e., operations that  $c$  inherits but does not override or virtual operation of  $c$ .
- $Op_I(c) \subseteq Op(c)$  be the set of operations *implemented* in  $c$ , i.e., operations that  $c$  inherits but overrides or non-virtual operation of  $c$ .

where  $Op(c) = Op_D(c) \cup Op_I(c)$  and  $Op_D(c) \cap Op_I(c) = \phi$ .

**Definition 5:** *New, virtual, overriding, and inherited operations*

For each component  $c \in C$ , let,  $Op(c)$  be the set of operations of component  $c$ .

- $Op_{NEW}(c) \subseteq Op(c)$  be the set of *newly* implemented operations in the component  $c$  (not overridden).
- $Op_{VIR}(c) \subseteq Op(c)$  be the set of operations declared as *virtual* in the component  $c$ .
- $Op_{OVR}(c) \subseteq Op(c)$  be the set of operations inherited and *overridden* in the component  $c$ .
- $Op_{INH}(c) \subseteq Op(c)$  be the set of operations *inherited* and not-overridden in the component  $c$ .

**Definition 6:** *Public and non-public operations*

For each component  $c \in C$ , let,  $Op(c)$  be the set of operations of component  $c$ .

- $Op_{PUB}(c) \subseteq Op(c)$  be the set of public operations of  $c$ .
- $Op_{NPUB}(c) \subseteq Op(c)$  be the set of non-public operations of  $c$ .

where  $Op(c) = Op_{PUB}(c) \cup Op_{NPUB}(c)$  and  $Op_{PUB}(c) \cap Op_{NPUB}(c) = \phi$ .

A public operation can be accessed by other operation of the system, while access of non-public operation varies from programming language to programming language. We are taking no assumption for the access of non-public method in our study.

**Definition 7:** *Set of all operations in the system*

$Op(C)$  is the *set of all operations* in the system and is represented as  $Op(C) = \cup_{c \in C} Op(c)$ .

### 2.2.5 Operation Invocations

In AO system, operation invocation can be defined in two categories: (i) *Implicit Invocations* are caused by a join point being reached or by an exception handler catching an exception, and (ii) *Explicit Invocations* are caused by code calling the operation. Operation invocation can be either static or dynamic; it is necessary to distinguish between these. Following are the definitions for the types of invocations.

**Definition 8:**  $IOI_{ST}(o)$  - *implicit operation invocations* that can be completely determined *statically*.

**Definition 9:**  $IOI_{DYN}(o)$ - *implicit operation invocations* that can only be determined *dynamically*.

**Definition 10:**  $EOI_{ST}(o, o')$ - *explicit operation invocations* that can be fully determined *statically*.

**Definition 11:**  $EOI_{ID}(o)$ - *explicit operation invocations*, accounting for *implementing descendants* of the operations component.

### 2.2.6 Indirect Operation Invocations

**Definition 12:** *Indirectly invoked operations.*

For an operation  $o \in Op(C)$ ,  $EOI_{ST}(o, o')$  and  $EOI_{ID}(o)$  are sets of operations explicitly directly invoked by  $o$ . There is need to account those operations also which can be indirectly invoked by  $o$ . Operation  $o$  indirectly invoked  $o'$ , if there are operations

$o_1, o_2, \dots, o_n$  such that  $o$  directly invokes  $o_1$ ,  $o_1$  directly invokes  $o_2$ , etc., and  $o_n$  directly invokes  $o'$ . This can be defined as:

$$EOI_{ST}(o, o') = \{o' \mid o' \in Op(C) \wedge \exists n \geq 1 \exists o_1, o_2, \dots, o_n \in Op(C): o_1 = o \wedge o_n = o' \wedge \forall i, 1 < i \leq n: o_i \in EOI_{ST}(o_{i-1}, o')\}$$

$$EOI_{ID}(o) = \{o' \mid o' \in Op(C) \wedge \exists n \geq 1 \exists o_1, o_2, \dots, o_n \in Op(C): o_1 = o \wedge o_n = o' \wedge \forall i, 1 < i \leq n: o_i \in EOI_{ID}(o_{i-1})\}$$

### 2.2.7 Attributes

Component have either inherited attributes or newly defined attributes. Attributes can be defined as:

**Definition 13:** *Declared and implemented attributes*

For each component  $c \in C$ , let  $Att(c)$  be the set of attributes of component  $c$ .

- $Att_D(c)$ - is a set of attributes declared in component  $c$  (i.e., inherited attributes)
- $Att_I(c)$ - is a set of attributes implemented in component  $c$  (i.e., new added attributes)

where  $Att(c) = Att_D(c) \cup Att_I(c)$

**Definition 14:**  $Att(C)$ - is the set of all attributes in the system and is represented as  $Att(C) = \cup_{c \in C} Att(c)$ .

In language specification, attribute is either a Field (*All*) or Intertype Field (*AJ*).

### 2.2.8 Attribute References

Attribute references may also be influenced by polymorphism. Hence, a complementary list of sets of attribute references  $AR(c)$  can be defined for a component  $c$  as:

**Definition 15:**

- $AR_{ST}(c)$ – is the set of attribute references, accounting only for statically determined attributes.
- $AR_{ID}(c)$ – is the set of attribute references, accounting for implementing descendants of the attribute's component.
- $AR_{FI}(c)$ – is the set of attribute references, accounting only for the first statically determined implementation, looking for the attribute's component and its ancestors.
- $AR_{ST}(c, c')$ – is the define sets with a meta variable representing the target component  $c'$ .

### 2.2.9 Types

Attribute and operation have some type and return type respectively.

**Definition 16:** Built-in types, user-defined types and component types

- $BT(S)$ – is the set of built-In types of the system  $S$ .
- $UDT(S)$ – is the set of user defined types of the system  $S$ .
- $CT(S)$ – is the set of component types of the system  $S$ .
- $T(S)$ – is the set all available types, and  $T(S) = BT(S) \cup UDT(S) \cup CT(S)$ .

## 2.3 RELATED WORK

In literature, there are many assessment frameworks and measurement approaches for measuring internal characteristics such as cohesion, coupling and complexity of OO Systems. This is because of high maturity level of OO paradigm. AOP is a new and emerging paradigm, for which in literature, there exist few assessment frameworks/measurement approaches/metrics to measure cohesion (Zhao *et al.*, 2004; Sant'Anna *et al.*, 2003; Gelinas *et al.*, 2006; Sant'Anna *et al.*, 2007), coupling (Zhao, 2004; Ceccato *et al.*, 2004; Bartsch *et al.*, 2006; Bartolomei *et al.*, 2006), and complexity (Dospisil, 2003; Pataki *et al.*, 2006). Most of these measures are for AspectJ

programming language. Bartolomei *et al.* (2006) proposed unified coupling measures for AO systems, in which Java, AspectJ and CaesarJ have been considered in their framework. In their work, metrics have not been defined. Detailed description of the measures for cohesion, coupling and complexity are as follows:

First approach towards measuring aspect cohesion was proposed by Zhao *et al.* (2004). Their approach is based on a dependency model for aspect-oriented software that consists of a group of dependency graphs. They defined cohesion as the degree of relatedness between attributes and modules (method/advice). They, in fact, presented two ways for measuring aspect cohesion based on inter-attributes ( $\gamma_a$ ), inter-modules ( $\gamma_m$ ) and module-attribute ( $\gamma_{ma}$ ) dependencies. The first way suggests that each measurement ( $\gamma_a, \gamma_m, \gamma_{ma}$ ) works as a field. Therefore, aspect cohesion for a given aspect A is defined as a 3-tuples  $\Gamma(A) = (\gamma_a, \gamma_m, \gamma_{ma})$ . They also suggested another way of cohesion measurement where each facet could be integrated as a whole with  $\beta$  parameters. Aspect cohesion is then expressed as:

$$\Gamma(A) = \begin{cases} 0, & n = 0 \\ \beta * \gamma_m, & k = 0 \text{ and } n \neq 0 \\ \beta_1 * \gamma_a + \beta_2 * \gamma_m + \beta_3 * \gamma_{ma}, & \text{otherwise,} \end{cases}$$

where k is the number of attributes and n is the number of modules in aspect A.  $\beta \in (0,1)$ ,  $\beta_1, \beta_2, \beta_3 > 0$ , and  $\beta_1 + \beta_2 + \beta_3 = 1$ .

Users determine the selection of weights  $\beta_1, \beta_2$  and  $\beta_3$ , which is arbitrary. However, some relationship definitions and their consideration in aspect cohesion measurement are difficult to capture. In general, this approach suggests a complex way to measure aspect cohesion that may be problematic to use in a real development context. Generating such dependency graphs is a time consuming process while parameter's weightage could be misleading. Secondly, using this approach, cohesion measurement is only for the family of AspectJ programming language.

Sant'Anna *et al.* (2003) proposed a metric 'Lack of Cohesion in Operations' (LCOO), which measures the amount of method/advice pairs that do not access to the

same instance variables. It is an extension of the well-known ‘Lack of Cohesion in Methods’ (LCOM) metric, developed by Chidamber and Kemerer (1994). This metric measures the lack of cohesion of a component. They considered a component  $C1$  with operations  $O_1, \dots, O_n$ . Let  $\{I_j\}$  = set of instance variables used by operation  $O_j, j=1, 2, \dots, n$ ,

$$|P| = \{(I_i, I_j) \mid I_i \cap I_j = \phi\}, \text{ and}$$

$$|Q| = \{(I_i, I_j) \mid I_i \cap I_j \neq \phi\}$$

If all  $n$  sets  $\{I_1\}, \dots, \{I_n\}$  are  $\phi$  then let  $P=Q$ . They defined LCOO as

$$\text{LCOO} = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise.} \end{cases}$$

According to Sant'Anna *et al.* (2003), a high LCOO value indicates disparateness in the functionality provided by the aspect. In this measurement, it is not stated whether inherited operations and attributes have been included or not, and we have to assume that sets  $\{I_j\}$  only include attributes of component  $C1$ . It suffers from several problems as stated, among others, by Henderson-Sellers (1996). Definition of this metric is also based on AspectJ-like programming languages.

Gelinas *et al.* (2006) proposed a metric ACoh to measure aspect cohesion. This cohesion measurement is based on dependency analysis. They defined this metric using two aspect cohesion criteria, *Modules-Data Connection Criterion* and *Modules-Modules Connection Criterion*.

#### ***Modules-Data Connection Criterion***

Let  $UA_{M_i}$  be the set of attributes used directly or indirectly by the module  $M_i$ . An attribute is used directly by a module  $M_i$ , if the attribute appears in its body. An attribute is indirectly used by a module  $M_i$ , if it is used directly by another module invoked directly or indirectly by  $M_i$ . There are  $m$  sets  $UA_{M_1}, UA_{M_2}, \dots, UA_{M_m}$ . Two modules  $M_i$  and  $M_j$  of an aspect are related by the *UA<sub>M</sub> relationship* if  $UA_{M_i} \cap UA_{M_j} \neq \phi$ . It means that there is at least one attribute shared (directly or indirectly) by the two modules.

#### ***Modules-Modules Connection Criterion***

Let  $UM_{M_i}$  be the set of modules used directly or indirectly by the module  $M_i$ . A module  $M_j$  is used directly by a module  $M_i$  if  $M_j$  appears in the body of  $M_i$ . A module  $M_j$  is indirectly used by a module  $M_i$ , if it is used directly by another module or indirectly by  $M_i$ . There are  $m$  sets  $UM_{M_1}, UM_{M_2}, \dots, UM_{M_m}$ . Two modules  $M_i$  and  $M_j$  of an aspect are related by the *UM<sub>M</sub> relation* if  $UM_{M_i} \cap UM_{M_j} \neq \phi$ . It means that there is at least one module jointly used (directly or indirectly) by the two modules. They also consider that  $M_i$  and  $M_j$  are directly related if  $M_j \in UM_{M_i}$  or  $M_i \in UM_{M_j}$ .

### ***Aspect Cohesion Measurement***

Two modules  $M_i$  and  $M_j$  can be connected in many ways: by sharing attributes or by sharing modules or both. Let  $NM(\text{Aspect}_i)$  be the total number of modules pairs in an aspect.  $NM$  is the maximum number of connections between aspect modules. Thus in an aspect having  $N$  modules,  $NM(\text{Aspect}_i) = N*(N-1)/2$ ,  $N > 1$ .  $NC(\text{Aspect}_i)$  is the number of connection between modules in undirected graph  $G_D$ . They defined a new metric for aspect cohesion measures:

$$ACoh(\text{Aspect}_i) = NC(\text{Aspect}_i) / NM(\text{Aspect}_i) \in [0, 1].$$

This cohesion metric is based on AspectJ-like programming languages. It is also not clear, whether they considered inherited member or not? If considered, what was the effect on cohesion values?

Sant'Anna *et al.* (2007) defined a metrics suite for concern-driven architecture. The proposed metrics suite is to measure cohesion, coupling and complexity of AO and non-AO systems. In this metric suite, 'Lack of Concern-based Cohesion' (LCC) is for measuring cohesion in concern-driven architecture, which counts the number of concerns addressed by the assessed component. This is very useful framework for evaluating internal software quality attributes such as cohesion, coupling, complexity and modularity at architecture level. In this framework, it is not specified, how to quantify aspect cohesion, coupling, complexity and modularity for empirical evaluation. It is also not clear, whether inheritance has been accounted for?

In order to measure coupling in AOS, Zhao (2004) proposed first coupling measurement framework that implements the coupling measures for AspectJ-like programming languages. His coupling measures are defined on the number of dependencies between aspect and classes such as *attribute-class*, *module-class*, *module-method* and *aspect-inheritance*. He also discussed the mathematical properties of these coupling measures, in which he showed that these measures satisfy properties that a good coupling measure should have. The coupling measures proposed by Zhao (2004) focused only on dependencies between aspect and classes, and did not take dependencies between aspects or between classes into account. This framework is useful only for AspectJ family languages.

Ceccato *et al.* (2004) proposed coupling metrics for AspectJ-like AOP languages. They represented *classes* and *aspects* as *module* and term *operation* subsumes class *methods* and aspect *advices/introductions*. They defined six different coupling metrics as:

- i. Coupling on Advice Execution (CAE) counts aspects containing advices possibly triggered by the execution of operations in a given module.
- ii. Coupling on Intercepted Module (CIM) counts modules or interfaces explicitly named in the pointcuts belonging to a given aspect.
- iii. Coupling on Method Calls (CMC) counts modules or interfaces declaring methods that are possibly called by a given module.
- iv. Coupling on Field Access (CFA) counts modules or interfaces declaring fields that are accessed by a given module.
- v. Response for a Module (RFM) counts operations potentially executed in response to a message received by a given module.
- vi. Crosscutting Degree of an Aspect (CDA) counts modules affected by the pointcuts and by the introductions in a given aspect.

These coupling metrics have been defined only for AspectJ and therefore needs to be redefined for other languages like CaesarJ and Hyper/J. Moreover, these metrics have not been validated theoretically or empirically.

Bartsch *et al.* (2006) suggested a coupling framework for AspectJ, which is an extension of a coupling framework for OO systems, defined by Briand *et al.* (1999). The extended framework contains a specific definition of the different coupling mechanisms found in AspectJ and an additional criterion: *instantiation*. These criteria are *types of connection, locus of impact, granularity, stability of server, direct or indirect connections, inheritance* and *instantiation*. Authors validated existing coupling metrics theoretically for AOP language using these framework criteria. They concluded that the definition of coupling measures is a difficult task that needs a great deal of rigor if the measures are to be theoretically valid and well defined. This can only be the first step to ensure that empirical validation operates on measures that are designed for their measurement purpose.

Bartolomei *et al.* (2006) proposed a unified coupling framework for AOP. In order to consistently and unambiguously define coupling measurements, they defined new terminologies for representing components of AO systems. In their framework, they considered two AOP languages, AspectJ and CaesarJ. They have extended their framework from Briand's frameworks (Briand *et al.*, 1999; Arisholm *et al.*, 2004) and tried to maximize the use of the terminology and coupling criteria originally defined in Briand's frameworks.

They have shown how the framework can be instantiated for Java, AspectJ and CaesarJ. Also they demonstrated the applicability of the framework by using it on an existing coupling metric. In their work, they have not proposed coupling metrics. Our coupling framework is derived from (Briand *et al.*, 1999; Arisholm *et al.*, 2004; Bartolomei *et al.*, 2006) and has proposed new coupling metrics for generic AO systems.

In order to measure complexity of AOS, Dospisil (2003) proposed a code complexity measurement approach for projects designed in AspectJ. They defined and used entropy metrics for ordering of symbols to estimate the complexity of AspectJ based programs. The entropy metrics are useful in ranking different modules and symbols with regard to their complexity. They introduced weighted entropy values to accommodate the subjective perspective of an observer. Their approach provides multi-valued space, more

suitable for prediction models. This framework is applicable only for AspectJ-like languages.

Pataki *et al.* (2006) described a multi-paradigm metric which is extended for aspect-oriented programs. The metric can measure complexity of MO, OO and AO parts of programs implemented in AspectJ. This extended metric revealed that aspect-oriented approach does not necessarily reduce the complexity on its own- the gain highly depend on the actual problems.

## **2.4 UNIFIED FRAMEWORKS**

### **2.4.1 Unified Framework for Cohesion Measures**

In order to define a unified cohesion measurement framework, frameworks for OO system (Briand *et al.*, 1998), AO systems (Zhao *et al.*, 2004; Sant'Anna *et al.*, 2003) and specific for AspectJ (Gelinas *et al.*, 2006) have been analyzed. We have also identified the influences of CaesarJ and Hyper/J and their specific features. From these analyses, a list of criteria emerged that must be considered, while proposing a new cohesion measurement framework. The proposed framework consists of five criteria; each criterion determines one basic feature of the resulting measure. In order to select the criteria, we need to describe it and decide, what decisions have to be made, what are the available options, how the criterion reflected by cohesion measures, and how the framework can be used to derive cohesion measures. The proposed criteria are as follows:

#### **2.4.1.1 Type of Cohesion Connections**

*What constitutes cohesion?*

The criterion for connections defines mechanism that causes cohesion in component. A connection within a component is a link between elements of the component (operation, attribute, inter-type attribute, or data declarations). In this framework, we take into account connections related to OO cohesion (Briand *et al.*, 1998), AspectJ specific connections (Gelinas *et al.*, 2006), and generic AO connections

(Zhao *et al.*, 2004). We also studied possible connections for cohesion caused by CaesarJ and Hyper/J type constructs.

**Definition 17:** *Type of connections* in a component  $c$

Let, an operation  $o, o' \in Op(c)$  and an attribute  $a \in Att(c)$ ,

- $CON\#1$ : is the connection where,  $o$  references  $a$ .
- $CON\#2$ : is the connection where,  $o$  invokes  $o'$ .
- $CON\#3$ : is the connection where,  $o$  and  $o'$  directly reference  $a$  of  $c$  in common (“similar operations”).
- $CON\#4$ : is the connection where,  $o$  and  $o'$  directly or indirectly reference  $a$  of  $c$  in common (“connected operations”).
- $CON\#5$ : is the connection where, data-data interacts (data declaration in  $c$ )
- $CON\#6$ : is the connection where, data-operation interacts

#### 2.4.1.2 Domain of Measure

*Which element to account for, which to restrict?*

By domain of measure means finding out which elements (attribute, operation, etc.) will participate in determining cohesion of the component. Most of the reviewed measures are defined at the component level. However, finer and coarser domains are also conceivable. We have also considered new elements, *cclass*, *wrapper constructor*, etc. of CaesarJ family, and *hyperslice*, *composition rules*, etc. of Hyper/J. This analyzes how closely related the attribute or operation is to other elements of its component or in other words, it is the degree to which the attribute or operation contributes to the cohesion of its component. From this measure, we can analyze, which element of the component should remain in the same component and which should be moved to new or another component.

| <b>Domain of Measures</b> | <b>Terminology Representation</b>                                                                                                                                                                                                                                                                                                                                      |
|---------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Components                | Class ( <i>All</i> ), Interface ( <i>All</i> ), Aspect ( <i>AJ</i> ), CClass ( <i>CJ</i> ) and Hyperslice( <i>HJ</i> )                                                                                                                                                                                                                                                 |
| Operations                | Method ( <i>All</i> ), Constructor ( <i>All</i> ), Static Initializer ( <i>All</i> ), Intertype Method ( <i>AJ</i> ), Intertype Constructor ( <i>AJ</i> ), Intertype Attribute ( <i>AJ</i> ), Pointcut ( <i>AJ, CJ, HJ</i> ), Advice ( <i>AJ, CJ</i> ), Declare Statements ( <i>AJ, CJ, HJ</i> ), Wrapper Constructor( <i>CJ</i> ) and Composition Rules( <i>HJ</i> ). |
| Attributes                | $Att_D(c)$ and $Att_I(c)$                                                                                                                                                                                                                                                                                                                                              |

**Table 2.1 Domain of Measures for Cohesion.**

It is also required to select the types of elements that should be accounted for. For example, Gelinas *et al.* (2006) count only attributes (data members) and module (methods and advice) in their framework. They measured cohesion on the basis of two connection criteria, modules-data connection criterion and modules-modules connection criterion. Zhao *et al.* (2004) measured aspect cohesion in which, they accounted attribute, intertype attribute, intertype methods, and advice as members responsible for cohesion of the component. In present proposed cohesion measurement framework, we accounted attributes and operations defined in Section 2.2, which includes elements of class (*All*), aspect (*AJ*), cclass (*CJ*) and hyperslice (*HJ*). Table 2.1 shows domain of measures with terminology representation.

#### **2.4.1.3 Direct or Indirect Connections**

*Count direct connections only or also indirect connections?*

In this section, our focus is whether to count direct connections only or also indirect connections. Direct connection definitely makes a component cohesive (*CON#1 to CON#6*). But for indirect connections, cohesion measures depend on the type of connections. For example, consider an operation  $o_1$ , which is “similar” to another operation  $o_2$  (*CON#3*), and operation  $o_2$  which is “similar” to operation  $o_3$  (*CON#3*).

Here operations  $o_1$  and  $o_2$  are directly connected. Similarly operations  $o_2$  and  $o_3$  are also directly connected. Now, we can say that operations  $o_1$  and  $o_3$  are indirectly connected, which is a transitive relationship connection.

#### 2.4.1.4 Inheritance

*Whether to account or not the members of  $Parents(c)$ , and  $Ancestors(c)$  of component  $c$ ?*

There are two different aspects to be considered with respect to inheritance.

- How do we assign attributes and operations to components?
- Whether to consider static or polymorphic invocation of operation?

These could be described as:

*How do we assign attributes and operations to components?*

For the analysis of cohesion measures, there are two elements of a component  $c$ , attributes  $Att(c)$  and operations  $Op(c)$ . For the measures of cohesion, either we can account non-inherited elements only or also account inherited elements.

- Exclude inherited attributes and operations from the analysis.*

If we exclude inherited attributes and operations, cohesion analysis will be limited to the same component only and will represent a single semantic concept.

- Include inherited attributes and operations in the analysis.*

If we include inherited attributes and operations, cohesion measures accounts attributes inherited, attributes newly added and operations  $Op_{NEW}(c)$ ,  $Op_{VIR}(c)$ ,  $Op_{OVR}(c)$ , and  $Op_{INH}(c)$  in the component  $c$ . We will analyze, whether component  $c$  as a whole, still represents a single semantic concept.

*Whether to consider static or polymorphic invocation of operation?*

Polymorphism will be relevant, only if the type of connection involves operation invocation. If  $c$  is a component, and  $Ancestors(c)$  is/are the ancestor(s) of  $c$ , then we have following two options:

- Do not account for polymorphism.*

Let  $o$  be an operation of component  $c$ , which is having connection with operations  $o'$ .  $o'$  can possibly be invoked statically in the implementation of  $o$ .

ii. *Account for polymorphism.*

Let  $o$  be an operation of the component  $c$ , which is having connections with operations  $o'$ .  $o'$  can possibly be in the implementation of  $o$  through polymorphism and dynamic binding.

#### 2.4.1.5 Access Operations and Constructors

*How to account for access operations and constructors for measuring cohesion?*

Access operations and constructors may artificially increase or decrease cohesion measure values. In the definition of cohesion measures and measurement framework, we have to decide how to account access operations and constructors.

*Access operations*

Following options can be used to account for access operations:

i. *Treat access operations as regular operations*

Do nothing; i.e., consider access operations as regular operations, which may cause artificially increase or decrease in cohesion values.

ii. *Consider the invocation of an access operation as a reference to the attribute*

If an operation is not referring an attribute directly, an access operation could be used to refer the attribute indirectly. These access methods are not accounted for by *CON#1* & *CON#3*, which causes artificially decreased number of references to attributes. If we account the invocation of access operation as reference to the attributes, it will be solution to this problem. Implementation of this problem is difficult because to recognize access operations automatically are not always possible.

iii. *Exclude the access operations from the analysis*

If pair of operations are using common attributes (*CON#3* & *CON#4*), then access operations may cause problems for cohesion measures. Access operations usually access only one attribute, many pairs of operations that do not refer a common attribute can be found using access operation. This problem causes artificially decrease in cohesion. Solution to this problem is to exclude access operations from the analysis.

### *Constructors*

Although we have considered constructors (*All*), Wrapper Constructors (*CJ*) and composition rules (*HJ*) as part of operation in Section 2.4.1.2, even then it is important to discuss effects of constructor, wrapper constructors and composition rules separately. Either we can include constructors in cohesion analysis or we can exclude constructors from cohesion analysis.

*i. Include constructors in the analysis*

Do nothing; *i.e.* count constructors and wrapper constructors as operation like method and advice.

*ii. Exclude constructors from the analysis*

Constructors, wrapper constructors and composition rules cause problems for measures that count pairs of operations which use common attributes (*CON#3* & *CON#4*). Most of the time, constructors typically reference all the attributes. It causes artificially increase in cohesion of the component because it generates many pairs of operations that use a common attribute. The solution to this problem is to exclude constructors from the analysis.

## **2.4.2 Unified Framework for Coupling Measures**

The objective of the proposed coupling measurement framework is to support the analysis and comparison of the existing coupling measures with respect to a particular measurement goal. This framework also provides guidelines to support the definition of new coupling metrics with respect to a particular measurement goal when there are no existing measures available. In order to define a generic coupling framework, we have analyzed coupling frameworks for object-oriented (OO) systems (Briand *et al.*, 1999; Arisholm *et al.*, 2004) and AO systems (Zhao, 2004; Bartsch *et al.*, 2006; Bartolomei *et al.*, 2006). In addition to this, we have also analyzed specific features, which causes coupling, of CaesarJ and Hyper/J languages. From these analyses, a list of criteria emerged that must be considered when defining new coupling metrics. We have derived these criteria, originally defined in Briand *et al.* (1999) and Arisholm *et al.* (2004) for object-oriented systems.

In order to select criterion, we need to describe it and decide, what decisions have to be made, what are the available options, how is the criterion reflected by the coupling measures. For each criterion, we can choose one or more of the available options, which will be influenced by the stated measurement goal. Following are the criteria of the coupling framework:

#### 2.4.2.1 Type of Coupling Connections

*What constitutes coupling?*

| CON# | Client Item                                 | Server Item                                    | Description                                                                                                                                                                           |
|------|---------------------------------------------|------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7.   | Attribute <i>a</i> of a component <i>c1</i> | Component <i>c2</i> ,<br><i>c1</i> ≠ <i>c2</i> | Component <i>c2</i> is the type of <i>a</i>                                                                                                                                           |
| 8.   | Operation <i>o</i> of a component <i>c1</i> | Component <i>c2</i> ,<br><i>c1</i> ≠ <i>c2</i> | Component <i>c2</i> is the type of parameter/return type of <i>o</i> .                                                                                                                |
| 9.   | Operation <i>o</i> of a component <i>c1</i> | Component <i>c2</i> ,<br><i>c1</i> ≠ <i>c2</i> | Component <i>c2</i> is the type of a local variable of <i>o</i> .                                                                                                                     |
| 10.  | Operation <i>o</i> of a component <i>c1</i> | Component <i>c2</i> ,<br><i>c1</i> ≠ <i>c2</i> | Component <i>c2</i> is the type of a parameter/return type of an operation explicitly invoked by <i>o</i> that can be fully statically determined.                                    |
| 11.  | Operation <i>o</i> of a component <i>c1</i> | Component <i>c2</i> ,<br><i>c1</i> ≠ <i>c2</i> | Component <i>c2</i> is the type of a parameter/return type of an operation explicitly invoked by <i>o</i> that can account for implementing descendants of the operation's component. |
| 12.  | Operation <i>o</i> of a component <i>c1</i> | Component <i>c2</i> ,<br><i>c1</i> ≠ <i>c2</i> | Component <i>c2</i> is the type of a parameter/return type of an operation implicitly invoked by <i>o</i> that can be fully statically determined.                                    |
| 13.  | Operation <i>o</i> of a component           | Component <i>c2</i> ,<br><i>c1</i> ≠ <i>c2</i> | Component <i>c2</i> is the type of a parameter/return type of an operation implicitly invoked by <i>o</i> that can be only                                                            |

|     |                                   |                                                   |                                                                                                                                    |
|-----|-----------------------------------|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
|     | $c1$                              |                                                   | dynamically determined.                                                                                                            |
| 14. | Operation $o$ of a component $c1$ | Operation $o'$ of a component $c2$ , $c1 \neq c2$ | $o$ invokes $o'$ explicitly that can be fully statically determined.                                                               |
| 15. | Operation $o$ of a component $c1$ | Operation $o'$ of a component $c2$ , $c1 \neq c2$ | $o$ invokes $o'$ explicitly, accounting for implementing descendants of the operation's component.                                 |
| 16. | Operation $o$ of a component $c1$ | Operation $o'$ of a component $c2$ , $c1 \neq c2$ | $o$ invokes $o'$ implicitly that can be fully statically determined.                                                               |
| 17. | Operation $o$ of a component $c1$ | Operation $o'$ of a component $c2$ , $c1 \neq c2$ | $o$ invokes $o'$ implicitly that can be only dynamically determined.                                                               |
| 18. | Operation $o$ of a component $c1$ | Attribute $a$ of a component $c2$ , $c1 \neq c2$  | $o$ references $a$ , which is statically determined attribute.                                                                     |
| 19. | Operation $o$ of a component $c1$ | Attribute $a$ of a component $c2$ , $c1 \neq c2$  | $o$ references $a$ , which is implementing descendents of the attributes component.                                                |
| 20. | Operation $o$ of a component $c1$ | Attribute $a$ of a component $c2$ , $c1 \neq c2$  | $o$ references $a$ , which is first statically determined implementation, looking for the attribute's component and its ancestors. |
| 21. | Operation $o$ of a component $c1$ | Attribute $a$ of a component $c2$ , $c1 \neq c2$  | $o$ references $a$ , which defines with a meta variable representing the target component.                                         |
| 22. | Component $c1$                    | Component $c2$ , $c1 \neq c2$                     | $c2$ is ancestor of $c1$ .                                                                                                         |
| 23. | Component $c1$                    | Component $c2$ , $c1 \neq c2$                     | High-level relationships between components, such as "uses", "consists-of".                                                        |

**Table 2.2 Types of Coupling Connections**

This criterion defines the types of connections between elements that are relevant to the coupling measure. Summary of the possible type of connections between client and server “item” (attributes, operations or components), is given in Table 2.2. Symbol “CON#” indicates connection number which is used as reference in remaining Sections and Chapters. CON#1 to CON#6 have been used for cohesion in Section 2.4.1.1 and CON#7 to CON#23 are used for coupling, which are given in Table 2.2. The items are listed in the columns “Client Item” and “Server Item”. Column “Description” explains the connection type.

#### **2.4.2.2 Locus of Impact**

*Import or export coupling?*

It is important to decide whether to count import coupling or export coupling or both.

- Import coupling counts the number of connections of a component *c1* with other components, which are used in *c1* or members of other components are used in *c1*. For example, operations of other components are invoked (implicit or explicit) in *c1*.
- Export coupling counts number of connections of a component *c1* with other components, which are using *c1* or which are using members of *c1*. For example, an operation of a component *c1* is invoked (implicit or explicit) in other components.

#### **2.4.2.3 Granularity of The Measure**

*What should be the domain of the measure and how to count coupling connections?*

*Domain of Measure*

Domain of measure means finding out which elements (component, attribute, operation, etc ) would participate in determining coupling of the AO system. If one domain is selected then other domain will be excluded from counting. For example, if selected domain is component then connection between elements of the same component

with elements of other components will not be counted as coupling connection. Table 2.3 shows domain of measures with terminology representation.

| Domain of Measures | Terminology Representation                                                                                                                                                                                                                                                                                                                                             |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| System             | C(S)                                                                                                                                                                                                                                                                                                                                                                   |
| Components         | Class ( <i>All</i> ), Interface ( <i>All</i> ), Aspect ( <i>AJ</i> ), CClass ( <i>CJ</i> ) and Hyperslice( <i>HJ</i> )                                                                                                                                                                                                                                                 |
| Operations         | Method ( <i>All</i> ), Constructor ( <i>All</i> ), Static Initializer ( <i>All</i> ), Intertype Method ( <i>AJ</i> ), Intertype Constructor ( <i>AJ</i> ), Intertype Attribute ( <i>AJ</i> ), Pointcut ( <i>AJ, CJ, HJ</i> ), Advice ( <i>AJ, CJ</i> ), Declare Statements ( <i>AJ, CJ, HJ</i> ), Wrapper Constructor( <i>CJ</i> ) and Composition Rules( <i>HJ</i> ). |
| Attributes         | Att <sub>D</sub> (c) and Att <sub>I</sub> (c)                                                                                                                                                                                                                                                                                                                          |
| Exception Handler  | Exc(c)                                                                                                                                                                                                                                                                                                                                                                 |

**Table 2.3 Domain of Measures for Coupling.**

#### *Counting Coupling Connections*

Following are the ways to count coupling connections listed in Table 2.2:

- i. Count individual connections, which are fully statically determined.
- ii. Count individual connections, which are only dynamically determined (ancestor elements, descendent elements, matching pointcuts, etc).
- iii. Count the number of distinct items at the other end of the connections, which are fully statically determined.
- iv. Count the number of distinct items at the other end of the connections which are only dynamically determined (ancestor elements, descendent elements, matching pointcuts, etc).

- v. Add up the number of connections counted as in (i) and in (ii) for each operation or attribute of the component.
- vi. Add up the number of connections counted as in (iii) and in (iv) for each operations or attributes of the components.
- vii. For a component  $c$ , count the number of other components to which there is at least one connection exists.

#### **2.4.2.4 Direct or Indirect Connections**

We have to decide whether to count direct connections only or indirect connections also. For example, operation  $o1$  of component  $c1$  invokes operation  $o2$  of component  $c2$ . Operation  $o2$ , in turn, invokes operation  $o3$  of component  $c3$ , where  $c1 \neq c2$  and  $c2 \neq c3$ . We can say  $o1$  indirectly invokes  $o3$ . This is a type of indirect connection.

#### **2.4.2.5 Stability of Server**

This criterion is same as in case of object-oriented systems. There is no distinction between object-oriented and aspect-oriented stability of server. It tells whether modification in server component will influence coupled components or not.

#### **2.4.2.6 Inheritance**

*Whether to account inheritance or not?*

In order to make decision whether to account inheritance or not, there are three main considerations: (i) does the use of members of ancestor class constitute coupling or not? (ii) does coupling measure consider both statically and polymorphically invoked operations? (iii) do inherited members belong to the inheriting component or not?

#### **2.4.2.7 Aggregation**

Aggregation specifies that what will be considered and how these connections will be counted. In order to take decision to aggregate connections, it must be decided: (i) which members of the connections are aggregated? (ii) whether to count import or export or both types of coupling connections? (iii) whether to count all type of connections listed

in Table 2.2 or distinct end points connections? (iv) what will be the level of counting indirect connections?

#### **2.4.2.8 Dynamic Issues**

These are the issues which cause coupling but are determined at run time or in other words these type of coupling connections cannot be determined statically. In order to deal with dynamic issues it must be decided: (i) how to determine the possible components at the end points of connections to members, when polymorphism is involved? (ii) what are the members of the components that influence the coupling measurement at run time? (iii) whether to consider it or not, connections involving join points matched by dynamic pointcuts?

#### **2.4.2.9 Language Specific Considerations**

Although, in the proposed generic coupling measurement framework for AOS, we have considered most of the coupling connections, there may be some connections, which might be in a particular AO language. Whenever a new language is considered as part of this generic coupling measurement framework, there may be some specific features in that particular language, which can cause coupling and have not been included in listed connections of Table 2.2.

In order to define metrics for complexity of generic AOS, complexity is divided in two categories: *code complexity* and *interaction complexity*. Interaction complexity is due to interaction between components of AOS, which is coupling. So, unified framework for complexity measures will be same as defined unified framework for coupling measures.

## **2.5 AOS METRICS SUITE**

In literature, metrics for internal characteristics such as cohesion, coupling, complexity and size of AOP have either been used, as originally defined for OOP with no or minor modifications. Some of the metrics for these characteristics have been defined for AOS, which are specific to AspectJ and there is no empirical or quantitative validation of these metrics. In present work, the proposed metrics are for generic aspect-oriented systems, which are applicable to Java, AspectJ-like, family of CaesarJ and Hyper/J.

Empirical and quantitative validation of these proposed metrics is presented in the Chapters to follow.

### 2.5.1 Cohesion Metric

In order to define cohesion metric for generic AOS, terminology and formalism defined in Section 2.2 and unified cohesion measurement framework, defined in Section 2.4.1, have been used. This metric is an extension of existing metric defined by Gelinas *et al.* (2006). First, we go through the whole process of analyzing and selecting the criteria, and then we derive a formal definition.

#### 2.5.1.1 Selection Criteria

For each criterion of the framework, one or more of the available options is chosen, depending on the decisions on the objective of the measurement. The types of connections are targets of explicit invocation and targets of join point implicit invocations. Domain of measures will be components, operations, and attributes. Account both direct and indirect connections because both involves in the cohesion of the component. Include inherited attributes and operations in the analysis, and also account for polymorphism. Treat access operations depending on conditions specified in Section 2.4.1.5. Exclude constructors from the analysis.

#### 2.5.1.2 Formal Definition of Cohesion Metric

Formal definition of unified aspect cohesion  $UACoh(c_i)$  metric for a component  $c_i$  of the system  $S$  :

$$UACoh(c_i) = ANC(c_i) / MNC(c_i) \in [0, 1]$$

where,  $ANC(c_i)$  is the actual number of connections ( $CON\#1$  to  $CON\#6$ ) between members of  $c_i$ .  $MNC(c_i)$  is the maximum number of possible connections of interest ( $CON\#1$  to  $CON\#6$ ) between the members of  $c_i$ .

## 2.5.2 Coupling Metrics

In order to define coupling metrics for generic AOS, terminology and formalism [Section 2.2] and unified coupling measurement framework [Section 2.4.2] have been used. These measures are defined on counting coupling connections only. We have divided coupling measures in six categories. These are briefly discussed as:

### *i. Coupling on Attribute Type (CoAT)*

Coupling on attribute type is the total number of connection counts of type CON#7.

### *ii. Coupling on Parameter Type (CoPT)*

Coupling on parameter type measure is the total number of connection counts of two different categories: (i) when a component type is parameter of an operation, and (ii) when a component is a local variable of an operation.

- First type of coupling measure is the total number of connection counts of type CON#8, CON#10, CON#11, CON#12 and CON#13.
- Second type of coupling measure is the total number of connection counts of type CON#9.

### *iii. Coupling on Attribute Reference (CoAR)*

Coupling on attribute reference measure is the total number of connection counts of type CON#18, CON#19, CON#20 and CON#21.

### *iv. Coupling on Operation Invocation (CoOI)*

Coupling on operation invocation measure is the total number of connection counts of type CON#14, CON#15, CON#16 and CON#17.

### *v. Coupling on Inheritance (CoI)*

Coupling on inheritance measure is the total number of connection counts of type CON#22.

### *vi. Coupling on High-level Association (CoHA)*

Coupling on high-level association measure is the total number of connection counts of type *CON#23*.

### 2.5.3 Complexity Metrics

Complexity of AO system ( $CMPX_{AOS}$ ) depends on two main factors: (i) code complexity of Component Set ( $C(S)$ ), and (ii) complexity of interactions between the components ( $IC(S)$ ) of Component Set. We have excluded interaction between members of the same component from the analysis of the framework. Code complexity of the component set ( $CMPX_{C(S)}$ ) is the sum of the code complexity of the individual component ( $M(c)$ ). Code complexity of a component can be evaluated as sum of the complexity of the attributes with their corresponding weight values, complexity of the operations with their corresponding weight values and complexity of the nested components with their corresponding weight values. Complexity of nested component can be evaluated recursively. Weight values corresponding to attributes depend on their types  $BT(S)$ ,  $UDT(S)$  and  $CT(S)$ . Weight values corresponding to operations depend on types of operations, simple code *Code* ( $o$ ), number of parameters, parameter types *Par* ( $o$ ) and return types *Rt* ( $o$ ).

Complexity of Interactions between components, ( $CMPX_{IC(S)}$ ) is due to three main factors: (i) complexity of operation invocation ( $CMPX_{OI(c)}$ ) (ii) complexity of attribute references ( $CMPX_{AR(c)}$ ) and (iii) complexity of statements ( $CMPX_{STMT(c)}$ ). Complexity of operation invocations is sum of the complexities of  $IOI_{ST}(o(c))$  with their corresponding weight values,  $IOI_{DYN}(o(c))$  with their corresponding weight values,  $EOI_{ST}(o(c), o'(c))$  with their corresponding weight values and  $EOI_{ID}(o(c))$  with their corresponding weight values. Weight values depend on the type of operation invocations ( $IOI_{ST}(o(c))$ ,  $IOI_{DYN}(o(c))$ ,  $EOI_{ST}(o(c), o'(c))$  and  $EOI_{ID}(o(c))$ ). Complexity of attribute references is sum of the complexities of  $AR_{ST}(c)$  with their corresponding weight values,  $AR_{ID}(c)$  with their corresponding weight values,  $AR_{FI}(c)$  with their corresponding weight values and  $AR_{ST}(c, c')$  with their corresponding weight values. Weight values depend on type of attribute references  $AR_{ST}(c)$ ,  $AR_{ID}(c)$ ,  $AR_{FI}(c)$ , and  $AR_{ST}(c, c')$ . Complexity of statements is sum of the complexities of exceptions ( $Exc(c)$ ) with their corresponding

weight values, array creations (ArrC(c)) with their corresponding weight values and type castings (TypeC(c)) with their corresponding weight values. Weight values in statements depend on the type of statements Exc(c), ArrC(c), and TypeC(c).

### 2.5.3.1 Selection Criteria

For each criterion of the framework, one or more of the available options is chosen depending on the decisions on the objective of complexity measurement. Domain of measure will be system, components, operations and attributes. Since we are measuring complexity of whole system, we need to account members of Parents(c) and Ancestors(c) because inherited members are also part of the component  $c$ . Access operations and constructors also play important role in component; we need to account access operations and constructors as regular operations. All three types of interactions: *operation invocations*, *attribute references* and *statements* cause interaction complexity; in this framework, we are accounting all three interaction types.

### 2.5.3.2 Formal Definition of Complexity Metrics

***Complexity of AO system:***

$$CMPX_{AOS} = CMPX_{C(s)} + CMPX_{IC(s)}$$

where,  $CMPX_{AOS}$  is complexity of AO system,  $CMPX_{C(s)}$  is the code complexity of Component Set and  $CMPX_{IC(s)}$  is interaction complexity of the Component Set.

***Code Complexity of Component Set:***

$$CMPX_{C(s)} = \sum_{x=1}^X CMPX_{M(c_x)}$$

where,  $x$  is the total number of components in the AO system (Component Set) and  $CMPX_{M(c_x)}$  is the code complexity of component  $c_x$ .

***Code Complexity of a Component:***

$$CMPX_{M(c)} = \alpha \times CMPX_{Att(c)} + \beta \times CMPX_{Op(c)} + \gamma \times CMPX_{Nested(c)}$$

where,  $\alpha$ ,  $\beta$  and  $\gamma$  are the coefficients for  $CMPX_{Att(c)}$ ,  $CMPX_{Op(c)}$  and  $CMPX_{Nested(c)}$  respectively and depend on the nature of components.  $CMPX_{Att(c)}$  is the complexity of attributes in component  $c$ ,  $CMPX_{Op(c)}$  is the complexity of operations in component  $c$  and  $CMPX_{Nested(c)}$  is the complexity of nested components in component  $c$ .

***Complexity of Attributes:***

$$CMPX_{Att(c)} = \sum_{l=1}^L w_l \times Att_l(c)$$

where,  $L$  is the total number of attributes in component  $c$  and  $w_l$  is the corresponding weight value of attribute  $Att_l(c)$ .

***Complexity of Operations:***

$$CMPX_{Op(c)} = \sum_{m=1}^M w_m \times Op_m(c)$$

where,  $M$  is the total number of operations in component  $c$  and  $w_m$  is the corresponding weight value of operation  $Op_m(c)$ .

***Complexity of Nested Components:***

$$CMPX_{Nested(c)} = \sum_{n=1}^N w_n \times Nested_n(c)$$

where,  $N$  is the total number of nested components in component  $c$  and  $w_n$  is the corresponding weight value of nested component  $Nested_n(c)$ .

Now, we can represent ***Code Complexity of a Component*** as:

$$CMPX_{M(c)} = \alpha \times \sum_{l=1}^L w_l \times Att_l(c) + \beta \times \sum_{m=1}^M w_m \times Op_m(c) + \gamma \times \sum_{n=1}^N w_n \times Nested_n(c)$$

and ***Code Complexity of Component Set*** as:

$$CMPX_{C(S)} = \sum_{x=1}^X \left( \alpha \times \sum_{l=1}^L w_l \times Att_l(c_x) + \beta \times \sum_{m=1}^M w_m \times Op_m(c_x) + \gamma \times \sum_{n=1}^N w_n \times Nested_n(c_x) \right)$$

### **Complexity of Interactions among Components:**

$$CMPX_{IC(S)} = CMPX_{OI(S)} + CMPX_{AR(S)} + CPMX_{STMT(S)}$$

where,  $CMPX_{IC(S)}$  is the complexity of interactions between the components of system  $S$ ,  $CMPX_{OI(S)}$  is the complexity of operation invocations in components of system  $S$ ,  $CMPX_{AR(S)}$  is the complexity of attribute references in components of system  $S$  and  $CPMX_{STMT(S)}$  is the complexity of statements in components of system  $S$ .

### **Complexity of Operation Invocations:**

$$CMPX_{OI(S)} = \sum_{x=1}^X \left( \sum_{d=1}^D w_d \times IOI_{STd(o)}(c_x) + \sum_{e=1}^E w_e \times IOI_{DYN e(o)}(c_x) + \sum_{f=1}^F w_f \times EOI_{STf(o, o')}(c_x) + \sum_{g=1}^G w_g \times EOI_{IDg(o)}(c_x) \right)$$

where  $X$  is the total number of components in system  $S$ ,  $D$  is the total number of implicit operation invocations that can be fully statically determined in component  $c_x$ ,  $E$  is the total number of implicit operation invocations that can be only dynamically determined in component  $c_x$ ,  $F$  is the total number of explicit operation invocations that can be fully statically determined in component  $c_x$  and  $G$  is the total number of explicit operation invocations, accounting for implementing descendants of the operations component in component  $c_x$ .  $w_d$  is the corresponding weight value of  $IOI_{STd(o)}(c_x)$ ,  $w_e$  is the corresponding weight value of  $IOI_{DYN e(o)}(c_x)$ ,  $w_f$  is the corresponding weight value of  $EOI_{STf(o, o')}(c_x)$  and  $w_g$  is the corresponding weight value of  $EOI_{IDg(o)}(c_x)$ .

### **Complexity of Attribute References:**

$$CMPX_{AR(S)} = \sum_{x=1}^X \left( \sum_{h=1}^H w_h \times AR_{STh}(c_x) + \sum_{i=1}^I w_i \times AR_{IDi}(c_x) + \sum_{j=1}^J w_j \times AR_{FIj}(c_x) + \sum_{k=1}^K w_k \times AR_{STk}(c_x, c_x') \right)$$

where  $X$  is the total number of components in system  $S$ ,  $H$  is the total number of attribute references, accounting only for statically determined attributes in component  $c_x$ ,  $I$  is the total number of attribute references, accounting for implementing descendants of the attribute's component in component  $c_x$ ,  $J$  is the total number of attribute references, accounting only for the first statically determined implementation, looking for the attribute's component and its ancestors in component  $c_x$  and  $K$  is the total number of define sets with a meta variable representing the target component in component  $c_x$ .  $w_h$  is the corresponding weight value of  $AR_{Stk}(c_x)$ ,  $w_i$  is the corresponding weight value of  $AR_{ID}(c_x)$ ,  $w_j$  is the corresponding weight value of  $AR_{FI}(c_x)$  and  $w_k$  is the corresponding weight value of  $AR_{Stk}(c_x, c_{x'})$ .

### **Complexity of Statements:**

$$CMPX_{STMT(S)} = \sum_{x=1}^X \left( \sum_{p=1}^P w_p \times Exc_p(c_x) + \sum_{q=1}^Q w_q \times ArrC_q(c_x) + \sum_{r=1}^R w_r \times TypeC_r(c_x) \right)$$

where  $X$  is the total number of components in system  $S$ ,  $P$  is the total number of exception occurred in component  $c_x$ ,  $Q$  is the total number of array creations in component  $c_x$ , and  $R$  is the total number of type casting in component  $c_x$ .  $w_p$  is the corresponding weight value of  $Exc_p(c_x)$ ,  $w_q$  is the corresponding weight value of  $ArrC_q(c_x)$ , and  $w_r$  is the corresponding weight value of  $TypeC_r(c_x)$ .

It is important to note that we can add above complexity metrics only when all metrics values are in same unit/scale. For example, if one metrics is in a/b format and ranges between 0 to 1, then others must be in same unit to add values. If scales are different then there is a need to normalize the units so that all metrics are of same scale.

### **2.5.4 Size Metrics**

In order to determine physical size of system's design and code, metrics suite of present study contains following size metrics.

**Number of system components (NOSC):** It counts the number of system components, *i.e.* the number of classes and aspects in the system. It counts only the name of the components and not instances.

**Lines of Code (LOC):** It counts the number of code lines. Comment lines, blank lines and documentation lines are not counted as part of LOC. It is one of the traditional metric for measuring size of the project.

**Number of Attributes (NOA):** It counts the number of attributes of each component. As per our generic framework, inherited attributes have also been included in counting.

**Number of Operations (NOO):** This metric counts the number of operations of each component. It also includes inherited operations in our framework.

## 2.6 CONCLUSION

In this Chapter, a framework for generic aspect-oriented systems, for measuring internal characteristics such as cohesion, coupling, complexity and size, have been defined. The present study is limited to those AOP languages, which have most of the features of Java programming language and features of AO technology. As part of generic aspect-oriented systems, three families of AOP languages: AspectJ-like, family of CaesarJ and Hyper/J, have been considered, selection of which is based on the scope of the study. In order to define new metrics for internal characteristics of generic AOS, new terminology and formalism have been defined to establish standardized, unambiguous and operational terminology. A metrics suite, for *cohesion*, *coupling*, *complexity* and *size* of generic aspect-oriented systems, has been defined. These metrics are applicable to Java, AspectJ-like, family of CaesarJ and Hyper/J. While defining these metrics, terminology and unified framework defined in the present Chapter have been used. These proposed metrics have large impact on external characteristics such as *understandability*, *usability*, *modularity*, *maintainability*, *adaptability* and *reusability*. So, use of these metrics for internal characteristics can be deployed to evaluate external characteristics of

generic AO systems. Empirical and quantitative validations of these proposed metrics have been presented in Chapters 3, 4 and 5.

Our framework is useful in describing existing metrics and measures for internal characteristics of AO technology. In future work, other AOP languages such AspectC, AspectC++ and AspectXML can also be included as part of generic AO framework. Results indicates that to make the framework generic after adding new AOP languages, major extensions and changes will not be required.



# CHAPTER 3

## ASSESSMENT OF COMPLEXITY OF GENERIC ASPECT-ORIENTED SYSTEMS USING FUZZY LOGIC APPROACH\*

---

### 3.1 INTRODUCTION

Researchers and practitioners have proposed complexity measurement frameworks and metrics for AOP (Sicilia *et al.*, 2006, Dospisil, 2003, Pataki *et al.*, 2006). But most of them are for AspectJ and complexity is defined in context of AspectJ. Despite a growing body of work dedicated to measure complexity in AO systems, there is no tool which could automate the assessment of complexity of generic AO systems. In this Chapter, a tool using fuzzy logic (Zadeh, 1994) for complexity measurement of generic AOS has been proposed. The input variables in fuzzy inference engine are the complexity metrics for generic AO systems defined in Chapter 2. Complexity metrics have been divided into two categories: *code complexity* and *interaction complexity*. The scope of the proposed fuzzy logic tool for measuring complexity is limited to *code complexity* only. There are three different input variables, which contribute in code complexity of component in AO systems. Ninety six fuzzy rules have been defined as rule base (knowledge base) to measure complexity of generic AO systems. The proposed framework has specifically targeted at the composition models supported by Java, AspectJ, CaesarJ and Hyper/J. This will help in: (i) defining new complexity metrics which, in turn, permits the analysis and comparison of Java, AspectJ, CaesarJ and Hyper/J implementations, (ii) integrating different existing measures, which examines the

\*The major findings of the chapter have been published in *Journal of Object Technology (JOT)*, 9(3), pp: 43-57, May-June 2010.

same concepts in different ways, and (iii) automate complexity measurement of majority of the AOP languages.

## 3.2 COMPLEXITY CONCEPTS

IEEE defines software complexity as “the degree to which a system or component has a design or implementation that is difficult to understand and verify” (IEEE, 1990).

Basili (1980) defines complexity as a measure of the resources expended by a system, while interacting with a piece of software to perform a given task. If the interacting system is a program, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software.

There is no consensus on how to define software complexity, and the term complexity measure is a misnomer. The true meaning of the term software complexity is the difficulty to maintain, change and understand software (Zuse, 1991).

These definitions associate software complexity with the difficulty of performing a task on the software. An implicit assumption is that software complexity correlates well with reusability, maintainability or overall development efforts.

### 3.2.1 Complexity Metrics for Module-Oriented and Object-Oriented Systems

Complexity is a major driver of the cost, reliability, and functionality of software systems. Several metrics have been proposed for measuring various aspects of complexity such as size, control flow, data structures, and inter-module structure. The most widely used complexity metric is Cyclomatic Complexity proposed by McCabe (1976). This metric is based on the program graph and is defined as:

$$V(G) = e - n + 2p$$

where  $e$  is the number of edges,  $n$  is the number of nodes in the graph and  $p$  is the number of connected components. Author proposed that  $V(G)$  can be used as a measure of procedure complexity of the program. Halstead (1977) complexity measurement was developed to measure a program module's complexity directly from source code (operators and operands in the module), with emphasis on computational complexity.

These measures are based on four scalar numbers, derived directly from a program's source code. These numbers are:

$n_1$ : the number of distinct operators,  $n_2$ : the number of distinct operands,

$N_1$ : the total number of operators and  $N_2$ : the total number of operands.

From these numbers, five measures are derived as shown in Table 3.1. Among the earliest software metrics, these measures served as strong indicators of code complexity. Because these are applied to code, these are the most often used as maintenance metrics (Edmond, 2007).

Kafura and Henry (1981) also proposed the complexity metrics based on the number of local information-flows entering (*fan-in*) and exiting (*fan-out*) in each procedure. This metric is given as:

$$\text{Complexity} = (\text{Proc. Length}) * (\text{fan-in} * \text{fan-out})^2$$

| Measure            | Symbol | Formula                   |
|--------------------|--------|---------------------------|
| Program length     | $N$    | $N = N_1 + N_2$           |
| Program vocabulary | $n$    | $n = n_1 + n_2$           |
| Volume             | $V$    | $V = N * (\log_2 n)$      |
| Difficulty         | $D$    | $D = (n_1/2) * (N_2/n_2)$ |
| Effort             | $E$    | $E = D * V$               |

**Table 3.1: Halstead Complexity Measures**

But in aspect-oriented context, these metrics may not be used to measure the complexity of entire AO component as these metrics measure the complexity for procedures only and do not consider the other features of the AO components like classes/aspects, attributes and operations, which may contribute a good amount of complexity to the AO component (Kumar *et al.*, 2008b).

Li (1983) proposed other metrics for complexity, based on size measures such as number of methods and attributes. In an object-oriented environment, Chidamber and Kemerer (1994) proposed a set of quality measures for class complexity. They proposed Depth of Inheritance Tree (*DIT*), Number of Children (*NOC*), and Weighted Methods Complexity (*WMC*) Coupling between Objects (*CBO*), Response for a Class (*RFC*) and Lack of Cohesion in Methods (*LCOM*). Out of these metrics *DIT*, *NOC*, *CBO* and *RFC* evaluate the external complexity of the relation between classes and do not depend upon the code complexity of the methods. *WMC* measures the complexity of the methods and is defined as:

$$WMC(C) = \sum_{i=1}^n C_i$$

where  $C_i$  is the static complexity of the corresponding method  $M_i$ . This metric evaluates the complexity of methods for a class.

### 3.2.2 Fuzzy Logic and Complexity Metrics for Aspect-Oriented Systems

The complexity metrics proposed by Dospisil (2003) and Pataki *et al.* (2006) are for projects designed in AspectJ. The details of which have already been discussed in Chapter 2, Section 2.3. Sicilia and García-Barriocana (2006) suggested about the main design and implementation issues, aspect-oriented design (AOD)-based extensions on OJB database libraries using fuzzy logic. They specified that fuzziness can be considered as separate crosscutting concern in existing software, and in consequence, AOD techniques provide a convenient framework to implement fuzzy extensions to existing libraries.

Xia *et al.* (2008) described a new way of assigning complexity weight values to function point metric. They discussed the concepts of calibrating Function Points, whose aims are to estimate a more accurate software size that fits for specific software application, to reflect software industry trend, and to improve the cost estimation of software projects. In this Chapter, a Function Point calibration model called Neuro-Fuzzy Function Point Calibration Model (NFFPCM) that integrates the learning ability from

neural network and the ability to capture human knowledge from fuzzy logic is proposed. The empirical validation using International Software Benchmarking Standards Group (ISBSG) data repository release 8 shows a 22% accuracy improvement of mean magnitude relative error (MMRE) in software effort estimation after calibration. This weight values assignment is for functions (operations) only and not for other members of the class, this framework is for object-oriented systems only.

### 3.3 PROPOSED COMPLEXITY MODEL

In order to define generic complexity model which accounts Java, AspectJ, CaesarJ and Hyper/J as part of generic AO framework, first step will be to define (i) AO terminology and formalism for unambiguous and standardized representation and (ii) generic AO framework which can specify different aspects like inheritance, domain of measure, interaction type and so on. Since this work is an extension of the work specified in Section 2.5.2 of Chapter 2, in which we have already defined both (i) and (ii).

We have divided complexity of generic AO system in two categories: (i) Code Complexity and (ii) Interaction Complexity. The scope of proposed fuzzy logic tool is to determine code complexity only and Interaction complexity can be determine in similar way. The metrics for code complexity are defined in Section 2.5.3.2 of Chapter 2.

It is important to note that we can add above complexity metrics only when all metrics values are in same unit/scale. For example, if one metrics is in a/b format and ranges between 0 to 1, then others must be in same unit to add values. If scales are different then there is a need to normalize units so that all metrics are of same scale. It is very difficult to assign numeric values to weight values  $w_l$ ,  $w_m$  and  $w_n$ . For example,  $w_m$  is numeric value of complexity weight value of an operation; operation complexity depends on return type, number of parameters and their parameter type; return type and parameter type may be built in types, user defined types and component types. Due to different number of parameters, different parameter types and different return types, there will be hundreds of combinations of complexity weight values of operations. Similarly we can assign different numeric values to  $w_l$  and  $w_n$ , which may be hundreds in numbers. Complexity of attributes, complexity of operations and complexity of nested components

are different in nature and have different type of complexity value.  $CMPX_{Att(c)}$  is due to data,  $CMPX_{Op(c)}$  is due to data as well as business logic (set of instructions) implemented for the data and  $CMPX_{Nested(c)}$  is contribution of both  $CMPX_{Att(c)}$  and  $CMPX_{Op(c)}$ . So, these complexities cannot be added to get  $CMPX_{M(c)}$  of a component. Solution of this problem is proposed in Sections 3.4.1 and 3.4.2.

### 3.4 A FUZZY-LOGIC APPROACH TO COMPLEXITY METRICS AND MODEL

The solution that is suggested here to overcome previously mentioned problems is to use fuzzy logic linguistic variables for the complexity metrics and model. Fuzzy logic is a mathematical tool for dealing with uncertainties and also it provides a technique to deal with imprecision and information granularity. Fuzzy logic is seen as a means of approximate reasoning .Our fuzzy model for integrating AO component complexity  $CMPX_{M(c)}$  accounts the effect of complexity of attributes  $CMPX_{Att(c)}$ , complexity of operations  $CMPX_{Op(c)}$  and complexity of nested components  $CMPX_{Nested(c)}$ . A block diagram for the fuzzy model is shown in Figure 3.1.

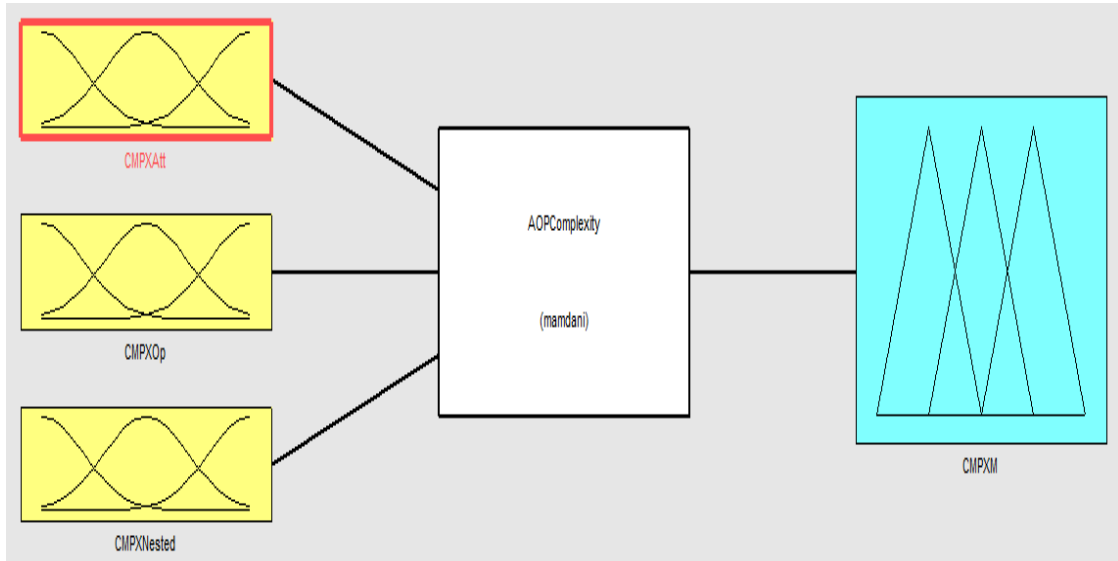


**Figure 3.1 Fuzzy Model for Complexity Measurement of a Component.**

The fuzzy model consists of four modules. The fuzzification module is the first stage in working of any fuzzy model, which transforms crisp input(s) into fuzzy values. In the second stage, these values are processed in the fuzzy domain by interface engine based on production rules (knowledge base) supplied by the domain expert(s). During second stage, the fuzzy operators are applied. In third stage implication process is applied and then all outputs are aggregated. In fourth and final stage, the processed output is transformed from fuzzy domain to crisp domain by defuzzification module.

### 3.4.1 Membership Functions For Input Parameters

In this Chapter, complexity of component  $CMPX_{M(c)}$  have been taken in the scale of 0 to 1 and member functions as NIL, Very Low (VL), Low (L), Medium (M), High (H) and Very High (VH). Because nested component is also a component, member function of nested component will be same as of a component *i.e.* NIL, VL, L, M, H and VH. Complexity of nested component  $CMPX_{Nested(c)}$  can be evaluated recursively with terminal condition as the component is without nested component *i.e.*  $CMPX_{Nested(c)}$  as NIL for that component. For simplification,  $CMPX_{Att(c)}$  and  $CMPX_{Op(c)}$  values have also been taken in the range of 0 to 1. For  $CMPX_{Att(c)}$  and  $CMPX_{Op(c)}$ , member functions have been considered as NIL, L, M and H. Now the question is, how to decide whether  $CMPX_{Att(c)}$  is NIL, L, M or H? Value of  $CMPX_{Att(c)}$  depends on number of attributes and their data types. In the Section 2.2 of Chapter 2, terminology and formulism, there are three broad categories of types as built in types (BT), user defined types (UDT) and component types (CT). If complexity weight value assigned to individual attribute is taken in the range of 0 to 1, then we can assign weight values to the three types as given in Table 3.2.

| Attribute Types | Complexity weight value |
|-----------------|-------------------------|
| BT              | 0.33                    |
| UDT             | 0.66                    |
| CT              | 1.00                    |

**Table 3.2: Complexity Weight Values to Individual Attribute**

In order to evaluate complexity of all the attributes of a component, *i.e.*  $CMPX_{Att(c)}$ , we can apply formula given in the Section 2.5.3.2 of Chapter 2. For example, in a component, there are 3 attributes of type BT, 4 attributes of type UDT and 2 attributes of type CT, then  $CMPX_{Att(c)}$  will be 5.63 ( $3*0.33+4*0.66+2*1.00$ ). For a component without attribute,  $CMPX_{Att(c)}$  will be NIL. In order to decide whether this value '5.63' of complexity of attributes falls in category of L, M or H, we performed experiment on benchmark projects (CaesarJ, 2004; Laddad 2003) and found that the maximum value of  $CMPX_{Att(c)}$  was 10.61 ( $9*0.33+4*0.66+5*1$ ), and in many other components this value was reaching 10. In other projects, this value may vary but if there is major difference, then proper decomposition of components at design time have not been done. For simplification, this maximum value of  $CMPX_{Att(c)}$  has been normalized in the scale of 0 to 1. Range of member functions NIL, L, M and H of variable  $CMPX_{Att(c)}$  are 0.00-0.00, 0.00-0.36, 0.33-0.68 and 0.65-1.00 respectively. There is overlapping in member function values for better results in fuzzy system. This is also shown in Figure 3.2.

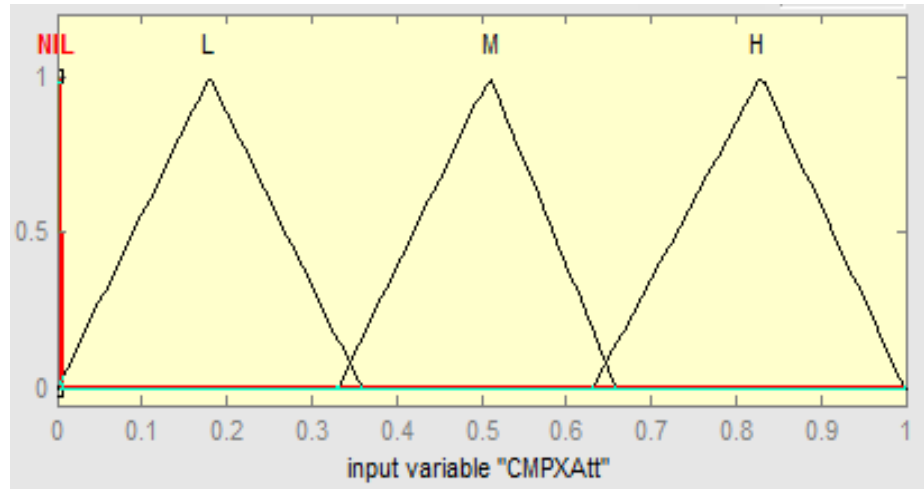


**Figure 3.2 Member Functions of Input Variable CMPXAtt**

Complexity of an operation depends on its return type, number of input parameters and parameter types because business logic written in an operation depends on these. To evaluate complexity of parameters, we can use same methodology as applied

on attributes. We can define whether an operation is simple type, medium type, complex type or very complex type; by input parameter complexity and return type. By the experience and expertise opinion of the domain, we have defined the different types of operations, which are listed in Table 3.3.

| <b>Return Type</b> | <b>Input Parameters Complexity</b> | <b>Operation Type</b> |
|--------------------|------------------------------------|-----------------------|
| Void               | NIL                                | Simple                |
| Void               | L                                  | Simple                |
| Void               | M                                  | Simple                |
| Void               | H                                  | Medium                |
| BT                 | NIL                                | Simple                |
| BT                 | L                                  | Simple                |
| BT                 | M                                  | Medium                |
| BT                 | H                                  | Complex               |
| UDT                | NIL                                | Simple                |
| UDT                | L                                  | Medium                |
| UDT                | M                                  | Medium                |
| UDT                | H                                  | Complex               |
| CT                 | NIL                                | Medium                |
| CT                 | L                                  | Medium                |
| CT                 | M                                  | Complex               |
| CT                 | H                                  | Very Complex          |

**Table 3.3: Type of Operations**

After categorizing operation types, we can assign complexity weight values to individual operations in the scale of 0 to 1 similar as in case of attributes, these are given in Table 3.4.

| Operation Types   | Complexity Weight Value to Individual Operation |
|-------------------|-------------------------------------------------|
| Simple Type       | 0.25                                            |
| Medium Type       | 0.50                                            |
| Complex Type      | 0.75                                            |
| Very Complex Type | 1.00                                            |

**Table 3.4: Complexity Weight Values of Operations**

We can now measure  $CMPX_{Op(c)}$  using formula given in Section 2.5.3.2 of Chapter 2 and by using weight values from Table 3.4. For example, a component is having 3 simple type operations, 2 medium type operations, 3 complex type operations and 1 very complex type operations, then  $CMPX_{Op(c)}$  for this component will be 5.0 ( $3*0.25+2*0.5+3*0.75+1*1.0$ ). Here, the same question arises as in case of  $CMPX_{Att(c)}$ , whether this value ‘5.0’ of  $CMPX_{Op(c)}$  is L, M or H? For the answer of this, we again performed experiment on same set of projects (CaesarJ, 2004; Laddad 2003) and found that maximum value of  $CMPX_{Op(c)}$  was 6.5 ( $6*0.25+4*0.50+2*0.75+1*1.00$ ). We also found that for many components,  $CMPX_{Op(c)}$  value is in between 5 and 6. This maximum value of  $CMPX_{Op(c)}$  may vary from project to project, but if there is major variation, then it may be because of improper decomposition of components. Range of member functions NIL, L, M and H of  $CMPX_{Op(c)}$  variable have been considered as 0.0-0.0, 0.0-0.36, 0.33-0.69 and 0.66-1.00 respectively, which are also shown in Figure 3.3.

Range of member functions NIL, VL, L, M, H and VH of input variable  $CMPX_{Nested(c)}$  and output variable  $CMPX_{M(C)}$  have been considered as 0.0-0.0, 0.0-0.23, 0.2-0.43, 0.4-0.63, 0.6-0.83 and 0.8-1.00 respectively, and for  $CMPX_{Nested(c)}$ , it is shown in Figure 3.4, which will be similar for  $CMPX_{M(C)}$ .

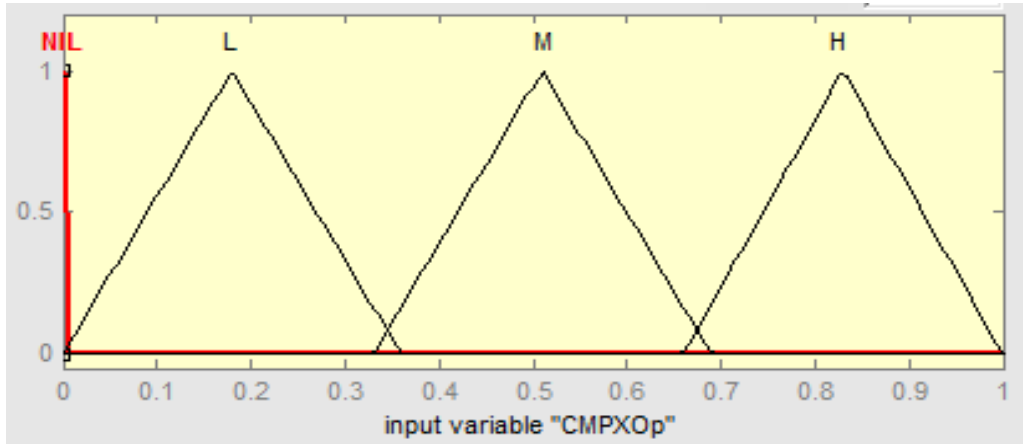


Figure 3.3 Member Functions of Input Variable CMPXOp

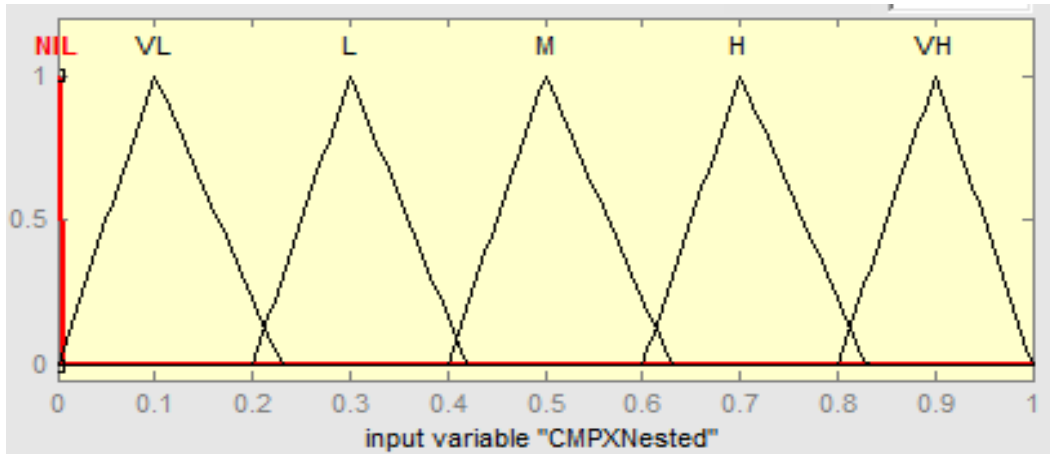


Figure 3.4 Member Functions of Input Variable CMPXNested

### 3.4.2 Fuzzy Rules for the Proposed Model

In order to measure complexity of a component  $CMPX_{M(c)}$ , which is the main objective of our model, there are three members  $CMPX_{Att(c)}$ ,  $CMPX_{Op(c)}$  and  $CMPX_{Nested(c)}$  contributing in the complexity of any component. Attributes, operations and nested components are different in nature and have different type of contribution in the complexity of a component, so we cannot simply add these values to get complexity of a component. As a solution of this problem, we have used fuzzy logic and have designed 96 fuzzy rules (4 member functions of  $CMPX_{Att(c)}$  \* 4 member functions of  $CMPX_{Op(c)}$  \* 6

member functions of  $CMPX_{Nested(c)}$ ). Here, mamdani method for defining fuzzy rules is used, which is used for nonlinear equations. These rules are designed on the basis of experience and expertise knowledge of the field that's why these are also known as knowledge base. For sample, some of the rules are listed in Table 3.5. First column labeled Rule# represent rule number, second column is for input linguistic variables,  $CMPX_{Att(c)}$ ,  $CMPX_{Op(c)}$  and  $CMPX_{Nested(c)}$  and third column is for output linguistic variable  $CMPX_{M(c)}$ .

| Rule# | Input Variables |                |                    | Output Variable |
|-------|-----------------|----------------|--------------------|-----------------|
|       | $CMPX_{Att(c)}$ | $CMPX_{Op(c)}$ | $CMPX_{Nested(c)}$ | $CMPX_{Op(c)}$  |
| 1     | NIL             | NIL            | NIL                | NIL             |
| 12    | NIL             | NIL            | VH                 | VH              |
| 25    | L               | NIL            | NIL                | VL              |
| 37    | L               | M              | NIL                | M               |
| 45    | L               | H              | L                  | H               |
| 57    | M               | L              | L                  | M               |
| 65    | M               | M              | H                  | VH              |
| 93    | H               | H              | NIL                | H               |

**Table 3.5: Some Sample Rules of the Complexity Fuzzy Model**

As an example, if  $CMPX_{Att(c)}=0.573$  (M),  $CMPX_{Op(c)}=0.596$  (M) and  $CMPX_{Nested(c)}=0.5$  (M) are input values then  $CMPX_{Op(c)}$  value is resulting as 0.692, which is high for output variable  $CMPX_{Op(c)}$ . It is also shown in Figure 3.5 as rule viewer.

Using proposed methodology and model, complexity of all the components of the software system can be measured and we can evaluate average of these complexity values. The average complexity value will be between 0 and 1 and will fall in any of the categories, VL, L, M, H and VH. With the help of this value, we can specify complexity

level of AO system. Three dimensional surface view of this rule base is given in Figure 3.6.

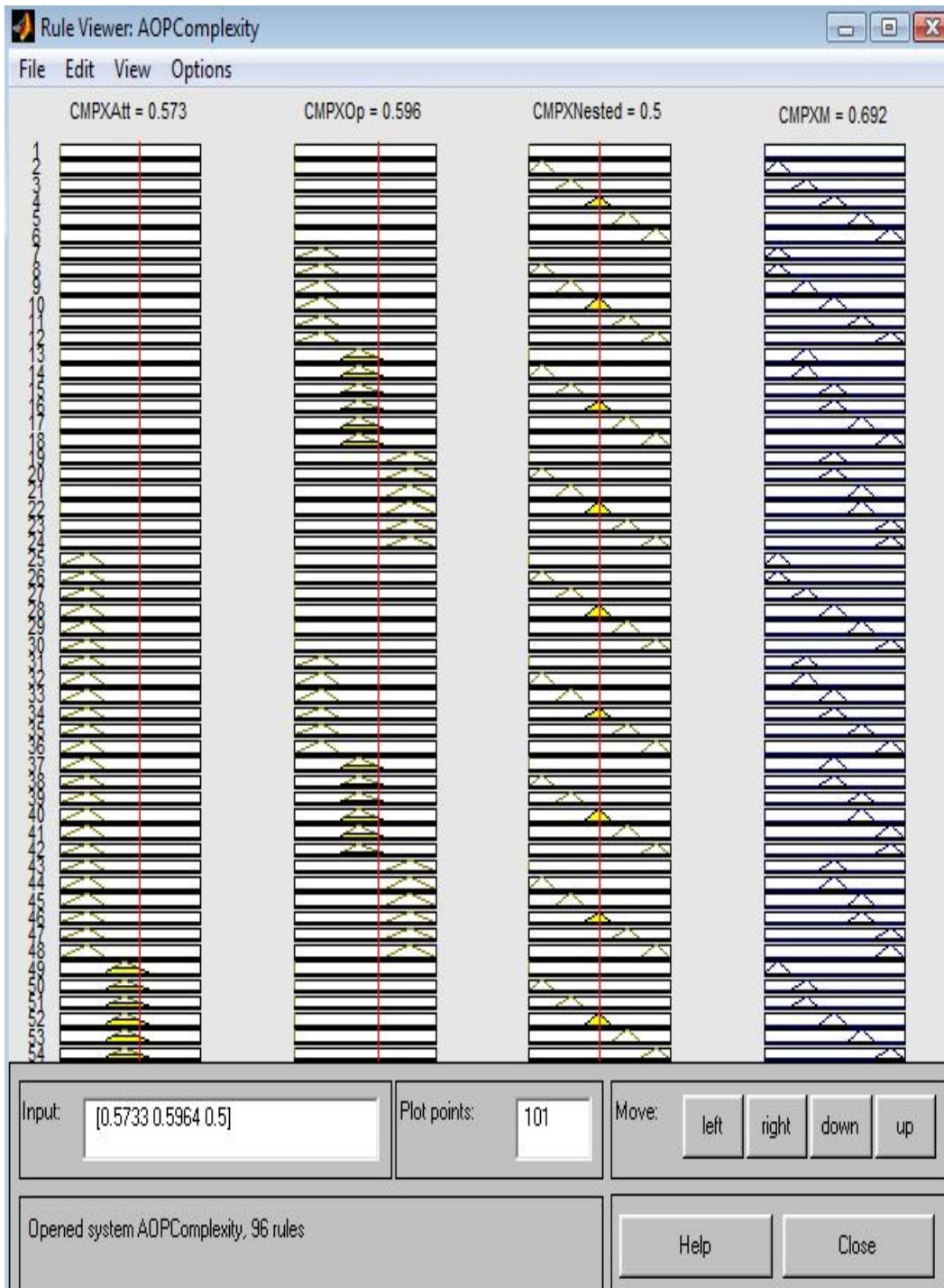


**Figure 3.5: Rule Viewer of AOPComplexity**

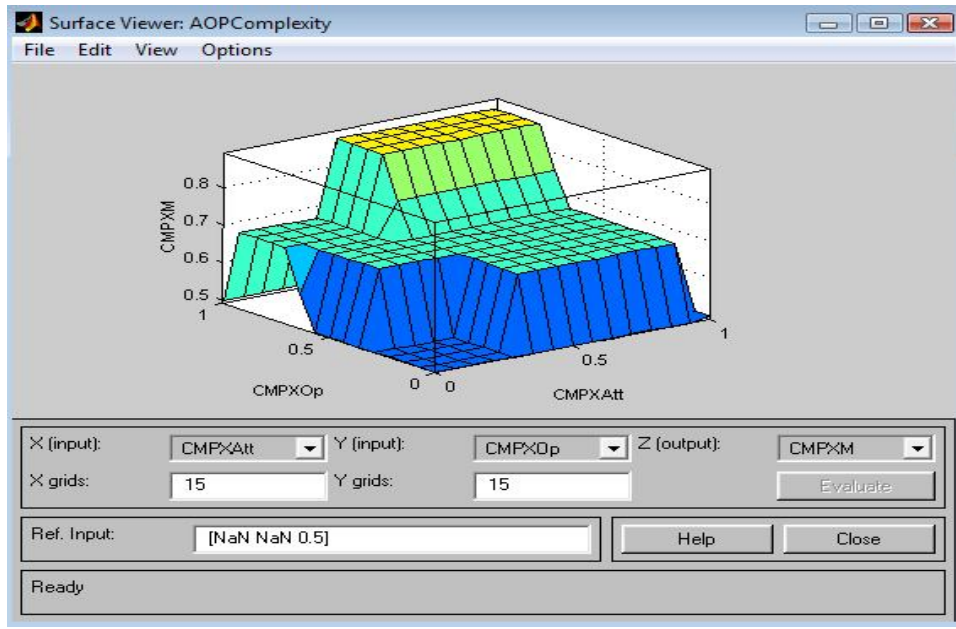


**Figure 3.6 Surface Viewer of AOPComplexity**

### 3.5 CONCLUSION

In this chapter, we have used fuzzy logic for defining software complexity metrics as linguistic variables and for the modeling process. Motivation of applying fuzzy logic is due to, difficulties faced to get total complexity of a component in generic aspect-oriented system because component members, which contribute in the complexity of a component, are different in nature and have different type of complexity value. Using common terminology, formalism and generic/unified framework defined for Java, AspectJ, CaesarJ and Hyper/J, new complexity metrics have been defined. These metrics are defined for measuring code complexity and interaction complexity of AO system. In this Chapter, only code complexity has been evaluated. A fuzzy model has been defined to measure code complexity of a component. Average complexity of all the components available in the AO software system will be indicator of the complexity level of the system. Using this model, complexity of software developed in most of the AO languages including Java (OOP), can be measured, which further may be used as an indicator to external software quality such as *maintainability*, *reusability*, *adaptability* and *understandability*. In future work, by applying similar approach, interaction complexity may also be assessed for AOS.



# CHAPTER 4

## PREDICTING MAINTAINABILITY OF ASPECT-ORIENTED SOFTWARE SYSTEMS\*

---

### 4.1 INTRODUCTION

Quality of software system/product is characterized by certain attributes, which are highlighted by ISO and IEEE standards. An example of such a standard is ISO/IEC 9126 (ISO, 2001), which is a standard that provides a generic definition of software quality, in terms of six main desirable characteristics viz, *functionality*, *maintainability*, *usability*, *efficiency*, *reliability*, and *portability*. Extensive studies have shown that maintenance is one of the growing major cost concerns (Pigoski, 1997). Software maintenance is a very broad activity in software development that includes error corrections, enhancement of capabilities, optimization, deletion of obsolete capabilities and so on. Maintenance includes all changes to the product, once the client has agreed that it satisfied the specified document. Maintenance may be corrective, adaptive or perfective. Corrective maintenance consists of the removal of residual faults. Adaptive maintenance changes functionality to meet another requirement in place of an existing requirement due to changes in environment, while perfective maintenance refers to enhancement and optimization. Corrective maintenance is important and critical because system will remain idle during this time whereas other types of maintenance activities can go in parallel with the software operation.

\*The major findings of the chapter has been published in (i) ACM SIGSOFT SEN, 33(6), pp:1-5, Nov. 2008 and (ii) accepted for publication in International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Vol.20, No.6, Sep. 2010.

In environments, where software changes are frequently required to improve software quality, *changeability* is an important sub-characteristic out of four sub-characteristics viz, *analyzability*, *changeability*, *stability*, and *testability* of maintainability in ISO/IEC 9126 (ISO, 2001). Maintainability of a software system depends on the software design approach (Rombach, 1990). Salient design approaches are: Module-Oriented (MO), Object-Oriented (OO), and AO. MO and OO paradigms have been used quite commonly and well-accepted in industry. Each has its own limitations and range of applicability. One of the major constraints has been the spread of concerns over various modules/classes (cross-cutting concerns). This leads to difficulty in understanding and maintenance of the program.

Most of the research work on changeability assessment has been carried out on MO and OO software (Hung *et al.*, 1994; Loyall and Mathisen, 1993; Zhao. 1999), whereas AO approach has not been studied much. Zhao (2002) did some work in this area based on program slicing technique, which is applicable only for AspectJ. This chapter accounts Java, AspectJ, CaesarJ and Hyper/J programming languages as part of generic AO systems. We have extended our work for changeability assessment in generic AO system and have used terminologies and formulism for generic AO systems defined in Chapter 2. Code level changes have been made in AO system and change-impact of a component to other components due to these changes has been evaluated. Correlation analysis between change impact and dependency among the components has been performed. The correlation factor value indicates that highly dependent AO systems will absorb less change. We extended this work and also checked whether cohesion metric can be used as indicator to predict the level of changeability in AOS system or not?

## **4.2 MAINTAINABILITY**

Maintainability is a prediction of the ease with which a system can evolve from its current state to its future desired state. It is termed as the most difficult and costliest activity due to its inherent involvement in making predictions about the future.

Maintainability is defined as (IEEE, 1998):

"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"

Another definition is (ISO, 2001):

"The capability of the software product to be modified" Modifications may include corrections, improvements or adaption of the software to changes in environment, and in requirements and functional specifications.

#### **4.2.1 Changeability in Traditional Software Systems**

Definition of changeability in ISO/IEC 9126 is as follows:

“Changeability describes capability of any software system to enable a specified modification to be implemented”.

In order to assess changeability of software systems, researchers and practitioner have proposed many changeability models. Some of the popular models are as follows:

Hsia *et al.* (1995) studied the relationship between maintainability and architecture, specially the inheritance hierarchy component, for the object-oriented approach. From their analysis, they found that maintainability of object-oriented systems is dependent on the characteristics of the inheritance. Their study shows that maintainability is facilitated by designs with higher broadness factor because it is focused on issues related already developed things; the smaller the broadness factor is, the more difficult it is to isolate the problem area. Their study is on perfective maintenance only and it was believed that corrective and adaptive maintenance will yield similar results.

Li and Offutt (1996) proposed a changeability assessment technique by measuring ripple effect on other classes due to code level change. Although the effects of changes in object-oriented are restricted, they are also more subtle and more difficult to detect. Their work presents algorithms to analyze the potential impacts of changes to object-oriented software, taking into account encapsulation, inheritance, and polymorphism. Their technique allows software developers to perform "what if" analysis on the effect of proposed changes, and thereby chose the change that has the least influence on the rest of the system. The analysis also adds valuable information (i) to regression testing, by

suggesting what classes and methods need to be retested, and (ii) to project managers, who can use the results for cost estimation and schedule planning.

Ryder and Tip (2001) developed a tool in an IDE to assess changeability in projects designed using Java programming language. Their tool determines the set of potentially affected test drivers. There are three possible scenarios: (i) if the programmer makes an edit that adds functionality to the program and the set `AffectedTests` is empty, then none of the test drivers are affected by the edit. This might occur when new, non-overriding methods are added, requiring new test drivers, (ii) alternatively, the tool may find a nonempty `AffectedTests` set, in this case, the programmer may need to modify an affected test driver, in order to comply with the edited program, and (iii) after all test drivers compile, an affected test driver can produce incorrect results. Two possible strategies can be followed to localize the fault. In first strategy, the tool creates a linear ordering, in the second strategy, binary search is used. At each step they continue with those changes that expose the fault. This tool may be helpful in determining changeability of Java programs.

Chaumon *et al.* (2002) study the changeability of software systems designed using C++ programming languages. In their work, assessment of the impact of changes to a system has been processed. Their model claimed both more complete and systematic than the similar work reported in the literature. For each of the possible changes identified in C++, the impact was calculated to ensure a successful system compilation after change implementation. For impact calculation, the various links between the classes of a system have to be considered. An experiment was carried out on an industrial system, using a prototype of their change impact model. The number of classes and lines impacted by applying the signature change to a method were calculated. Their results showed that the method signature change is a change that the system can absorb quite easily as maximum one class was impacted per method signature change. Another outcome of the experiment led to the establishment of a relation between WMC, a design metric, and the mean change impact. The higher the WMC value was, the higher was the mean change impact.

#### **4.2.2 Changeability in AO Systems**

First approach towards assessment of changeability in aspect-oriented systems was

proposed by Zhao (2002). His approach supports change impact analysis of aspect-oriented software based on *program slicing technique*. He performed changes in source code and assessed change impact analysis to other modules of AO software. All definitions in his approach to measure change impact are in terms of AspectJ. In order to demonstrate the usefulness of his impact analysis approach, he implemented a slicing tool for AspectJ and performed some experiments using the slicing tool to show the effectiveness of their slicing-based change impact analysis approach for supporting aspect-oriented software evaluation.

Li *et al.* (2006) have studied how AOP eases the adding and replacing the components in COTS-based development. When adding or replacing a COTS component, the main benefit of using AOP in a COTS-based system is that fewer classes need to be changed that uses OOP. However, using AOP does not ensure that less Lines Of Code (LOC) need to be modified when adding or replacing COTS components. It depends on whether the cross-cutting concerns in the glue-code are homogeneous or heterogeneous. Applying AOP, when cross-cutting concerns are heterogeneous, may not bring benefits.

Kumar *et al.*, (2007) proposed a changeability measurement technique for projects designed using AspectJ. The change impact has been evaluated for each of the possible code level changes in AspectJ, in which required changes should be made to ensure a successful system compilation after change implementation. Their result shows that a single change at code level will affect the other modules, which on an average, is less than one. This implies that not more than one module is affected with a single change. They also observed that by increasing in WOM metric value, change impact also increases. Correlation factor between WOM and change impact was found to be 0.41, which is weak. It indicates that WOM may be used as an indicator for changeability or maintainability, but not as a strong indicator.

Shen *et al.*, (2008) proposed a fine-grained coupling metrics suite for AO systems. Based on their coupling metrics suite, they measured coupling metrics for software changes during system evolution. In their work, they also presented a correlation model

between coupling metrics and system maintainability. The experiment result suggests that their correlation model provides useful information to evaluate the maintainability of an AO system.

### **4.3 CHANGE IMPACT ANALYSIS**

Change impact analysis is the task through which the programmers can assess the extent of the change, i.e. the software component that will cause impact with a change, or will be impacted by the change. Change impact analysis provides techniques to address the problem by identifying the likely ripple-effect of software changes and using this information, re-engineer the software system design so that software system should work without any problem (Chaumun *et al.*, 2002).

From the viewpoint of separation of *concerns* in software development, change impact analysis can be performed at many levels of software systems during software evolution such as at the specification level, design level, architecture level and code level. Our work is focused on code level change impact for generic/unified AO systems.

#### **4.3.1 Code Level Changes**

Changeability could be evaluated by making changes at code level in AO system and find affects (change impact) from these changes. Following may be the possible code level changes in a generic AO system.

##### ***System Level Changes***

- Add super component
- Delete super component
- Add sub component
- Delete sub component
- Add a component reference
- Delete a component reference
- Add an aggregated component
- Delete an aggregated component

### ***Component Level Changes***

- Add member
- Delete member
- Define / Redefine member
- Change member
  - Change member access scope
  - Change operation signature
  - Change data member
  - Operation Implementation Change
- Change pointcut
  - Add a pointcut
  - Delete a pointcut
  - Pointcut signature change

It is not feasible to perform all possible changes to evaluate average change impact (ACI) in a big AO system. In order to evaluate ACI, some random changes of different patterns have been made in this work. Minimum number of random changes required in a component need to be defined to get approximately same value of ACI, which we were getting by doing all possible changes. To define minimum number of random changes of different patterns, required in a component, we performed an experiment on samples of AspectJ and CaesarJ of different size components. On the basis of these results, criteria have been defined for minimum number of random changes, which is listed in Table 4.1.

| Number of operations in the component | Minimum number of random changes required to evaluate ACI |
|---------------------------------------|-----------------------------------------------------------|
| 1                                     | 4                                                         |
| 2                                     | 7                                                         |
| 3                                     | 12                                                        |
| 4                                     | 18                                                        |

|    |     |
|----|-----|
| 5  | 21  |
| 6  | 23  |
| 7  | 25  |
| 8  | 26  |
| 9  | 28  |
| >9 | >28 |

**Table 4.1 Criteria for Minimum Number of Random Changes**

#### 4.3.2 Change Impact Evaluation

Average change impact can be evaluated as sum of the number of affected operations (of the same component or of another component) due to changes made in the code, and dividing by total number of changes made, which can be formulized as:

$$ACI(c_i) = \frac{\sum_{j=1}^N NAO_j(c_i)}{N}$$

where  $ACI(c_i)$  is average change impact of component  $c_i$ ,  $N$  is the number of changes made which is based on criteria for minimum number of random changes required in component  $c_i$ , and  $NAO_j(c_i)$  is the number of affected operations by  $j^{th}$  change in a component  $c_i$ . For example, consider a change in the scope of an attribute  $a1$  of component  $c1$  from *public* to *protected*, here, components, which are not derived from  $c1$  and referring  $a1$ , will be impacted but components which are derived from  $c1$ , will not be impacted. Similarly, for change in the signature of an operation  $o1$ , components in which  $o1$  is invoked implicitly or explicitly will be impacted and rest of the components will not be impacted.

#### 4.4 CASE STUDY

Proposed change impact assessment technique has been applied on existing AO programs. Using AO terminologies and criteria for minimum number of random changes

required, we have evaluated ACI of all the components. The value ACI of the component is listed against the component name in Table 4.2 and Table 4.3.

It is very difficult to get realistic and very big AO software systems to test proposed approach. So, our proposed changeability assessment technique for generic AO systems has been tested on two different sets, which are of medium size and were available for experiment. One of these tested systems (CaesarJ, 2004) is in CaesarJ, which has 35 components and these are listed in Table 4.2. Another tested AO system (Laddad, 2003) is AspectJ based, which has 14 components in group-I and 10 components in group-II. These components are listed in Table 4.3. These groups have two different versions of similar AspectJ based AO systems.

| <b>Name of Components</b>     | <b>ACI</b> | <b>Number of Dependencies</b> |
|-------------------------------|------------|-------------------------------|
| Company                       | 1.48       | 3                             |
| Database                      | 0.53       | 2                             |
| Department                    | 0.48       | 2                             |
| Employee                      | 1.02       | 3                             |
| CompanyHierarchyBinding       | 1.32       | 4                             |
| CompanyHierarchyDisplay       | 0.32       | 2                             |
| Application                   | 1.28       | 4                             |
| HierarchyDisplayControl       | 2.68       | 5                             |
| HierarchyView                 | 0.24       | 1                             |
| RandomCompanyChange           | 1.32       | 3                             |
| GUIHierarchyBinding           | 0.00       | 0                             |
| GUIHierarchyDisplay           | 0.54       | 2                             |
| AdjustedAngurHierarchyDisplay | 1.03       | 2                             |

|                          |      |   |
|--------------------------|------|---|
| AdjustedHierarchyDisplay | 0.23 | 1 |
| AngularHierarchyDisplay  | 0.31 | 1 |
| HierarchyDisplay         | 0.09 | 0 |
| IHierarchyDisplay        | 0.52 | 1 |
| CompanyRemoting          | 0.04 | 0 |
| CompanyServer            | 1.23 | 2 |
| Connector                | 2.02 | 3 |
| ConnPoint                | 0.34 | 1 |
| FreePoly                 | 1.87 | 3 |
| Line                     | 0.82 | 2 |
| MultiPointShape          | 0.63 | 1 |
| Polygon                  | 0.72 | 2 |
| Polyline                 | 0.82 | 1 |
| Rectangle                | 1.43 | 3 |
| RightAngledConnector     | 0.73 | 2 |
| Shape                    | 0.30 | 1 |
| StraightConnector        | 0.72 | 2 |
| TextShape                | 0.83 | 3 |
| TwoPointShape            | 0.74 | 2 |
| Intersection             | 0.60 | 1 |
| Point                    | 0.00 | 0 |
| TextDraw                 | 0.00 | 0 |

**Table 4.2 CaesarJ Project- Tested for Changeability**

| <b>Name of Components(Group-I)</b>  | <b>ACI</b> | <b>Number of Dependencies</b> |
|-------------------------------------|------------|-------------------------------|
| Account                             | 0.34       | 2                             |
| AccountSimpleImpl                   | 1.02       | 2                             |
| CheckClearanceSystem                | 1.65       | 3                             |
| CheckingAccount                     | 0.32       | 0                             |
| CheckingAccountSimpleImpl           | 0.95       | 2                             |
| Customer                            | 0.38       | 1                             |
| InsufficientBalanceException        | 0          | 0                             |
| LogAccountActivities                | 0.46       | 1                             |
| SavingsAccount                      | 0.23       | 0                             |
| SavingsAccountSimpleImpl            | 1.94       | 2                             |
| Test                                | 3.43       | 5                             |
| IndentedLogging                     | 0          | 0                             |
| AbstractDebitRulesAspect            | 0.23       | 0                             |
| RuleEngineBasedDebitRulesAspect     | 1.24       | 4                             |
| <b>Name of Components(Group-II)</b> | <b>ACI</b> | <b>Number of Dependencies</b> |
| Account                             | 0.26       | 1                             |
| AccountJDBCImpl                     | 1.78       | 3                             |
| DatabaseHelper                      | 0.57       | 1                             |
| InsufficientBalanceException        | 0.97       | 2                             |
| InterAccountTransferSystem          | 0.00       | 0                             |
| PreserveCheckedException            | 0.46       | 1                             |

|                       |      |   |
|-----------------------|------|---|
| Test                  | 0.75 | 2 |
| TransactionLogging    | 1.84 | 3 |
| IndentedLogging       | 0.00 | 0 |
| JDBCTransactionAspect | 1.02 | 2 |

**Table 4.3 AspectJ Project- Tested for Changeability**

#### 4.4.1 Validation

To validate our changeability measurement approach, we have measured dependency (both implicit and explicit) among components. This dependency is measured with the help of adjacency matrix. Number of dependencies for the components has been listed in Table 4.2 and Table 4.3, against the name of the components. We have performed correlation analysis between ACI and number of dependencies. For CaesarJ based system, correlation value is 0.87 and for AspectJ based system, correlation value (combining both the groups) is 0.89. In both the AO systems, correlation factor values are high enough to indicate that in highly dependent AO system, change impact will be high i.e. highly dependent AO system will absorb less changes. In other words, these systems will have low changeability.

## 4.5 COHESION MEASURE AS INDICATOR OF CHANGEABILITY

Further, we extended this work and performed correlation analysis between ACI and cohesion metrics  $UACoh(C_i)$  [Section 2.5.1.2] to check whether  $UACoh(C_i)$  metric can be used as indicator in predicting changeability or not. In order to validate it, we performed an experiment.

In order to get numeric values of  $UACoh(C_i)$  metric, we have selected same set of projects, which we have considered in section 4.4.  $UACoh(C_i)$  metric values for each component have been evaluated using formula given in section 2.5.1.2. These evaluations have been performed manually. Values for  $UACoh(C_i)$ , for each component, are listed in Table 4.4 and Table 4.5. Values for ACI for same set of components, we had already given the evaluated values, for ACI for same set of components, in section 4.4. These values are also listed against the name of the component in Table 4.4 and Table 4.5.

Since both the languages are extensions of Java and both the experimented systems have class as primary concern, we do not find any need to analyze pure Java based system separately.

#### 4.5.1 Results

In Table 4.4 and Table 4.5, *UACoh* values are given against name of the component. *UACoh* values are between 0 and 1 including these two values. Frequency of zero values of *UACoh* is high in AspectJ based project, because more components in this project are either interface or abstract classes.

| <b>Nane of Components</b>       | <b>UACoh</b> | <b>ACI</b> |
|---------------------------------|--------------|------------|
| Company                         | 0.4          | 1.48       |
| Database                        | 0            | 0.53       |
| Department                      | 0.28         | 0.48       |
| Employee                        | 0.44         | 1.02       |
| CompanyHierarchyBinding         | 0.08         | 1.32       |
| CompanyHierarchyDisplay         | 0            | 0.32       |
| Application                     | 0.6          | 1.28       |
| HierarchyDisplayControl         | 0.36         | 2.68       |
| HierarchyView                   | 1            | 0.24       |
| RandomCompanyChange             | 0.71         | 1.32       |
| GUIHierarchyBinding             | 0.03         | 0.00       |
| GUIHierarchyDisplay             | 0            | 0.54       |
| AdjustedAngularHierarchyDisplay | 0            | 1.03       |
| AdjustedHierarchyDisplay        | 0.09         | 0.23       |
| AngularHierarchyDisplay         | 0.07         | 0.31       |
| HierarchyDisplay                | 0.09         | 0.09       |
| IHierarchyDisplay               | 0            | 0.52       |
| Connector                       | 0.04         | 2.02       |
| ConnPoint                       | 0            | 0.34       |

|                      |      |      |
|----------------------|------|------|
| FreePoly             | 0.37 | 1.87 |
| Line                 | 0.47 | 0.82 |
| MultiPointShape      | 0.44 | 0.63 |
| Polygon              | 0.1  | 0.72 |
| Polyline             | 0.39 | 0.82 |
| Rectangle            | 0.23 | 1.43 |
| RightAngledConnector | 0.39 | 0.73 |
| Shape                | 0.11 | 0.30 |
| StraightConnector    | 0.34 | 0.72 |
| TextShape            | 0.14 | 0.83 |
| TwoPointShape        | 0.49 | 0.74 |
| Intersection         | 1    | 0.60 |
| Point                | 0.91 | 0.00 |
| TextDraw             | 0    | 0.00 |

**Table 4.4 CaesarJ Project- Tested for Cohesion and Changeability**

| <b>Name of Components(Group-I)</b> | <b>UACoh</b> | <b>ACI</b> |
|------------------------------------|--------------|------------|
| Account                            | 0            | 0.34       |
| AccountSimpleImpl                  | 0.33         | 1.02       |
| CheckClearanceSystem               | 0            | 1.65       |
| CheckingAccount                    | 0            | 0.32       |
| CheckingAccountSimpleImpl          | 0            | 0.95       |
| Customer                           | 0.3          | 0.38       |
| InsufficientBalanceException       | 0            | 0          |
| LogAccountActivities               | 0.5          | 0.46       |
| SavingsAccount                     | 0            | 0.23       |
| SavingsAccountSimpleImpl           | 0.29         | 1.94       |
| Test                               | 0            | 3.43       |

|                                     |              |            |
|-------------------------------------|--------------|------------|
| IndentedLogging                     | 1            | 0          |
| AbstractDebitRulesAspect            | 0            | 0.23       |
| RuleEngineBasedDebitRulesAspect     | 0.3          | 1.24       |
| <b>Name of Components(Group-II)</b> | <b>UACoh</b> | <b>ACI</b> |
| Account                             | 0            | 0.26       |
| AccountJDBCImpl                     | 0.4          | 1.78       |
| DatabaseHelper                      | 0            | 0.57       |
| InsufficientBalanceException        | 0            | 0.97       |
| InterAccountTransferSystem          | 0            | 0.00       |
| PreserveCheckedException            | 0            | 0.46       |
| Test                                | 0            | 0.75       |
| TransactionLogging                  | 0.2          | 1.84       |
| IndentedLogging                     | 1            | 0.00       |
| JDBCTransactionAspect               | 0.43         | 1.02       |

**Table 4.5 AspectJ Project- Tested for Cohesion and Changeability**

What is the inference of measuring cohesion of generic aspect-oriented system? Answer to this question is that *UACoh* metric could be used as an indicator to measure external software qualities like *maintainability*, *reusability*, *modularity* etc. First step towards finding correlation between *UACoh* and external software qualities, correlation analysis has been performed between *UACoh* and Average Change Impact (*ACI*). Correlation factor between *UACoh* and *ACI* in CaesarJ based project is 0.08 and in AspectJ based project is -0.13. Both values indicate that *UACoh* metric cannot be used as an indicator of changeability assessment. Kabaili *et al.* (2001a, 2001b) performed experiment for class cohesion as predictor of changeability for C++ projects. They found that cohesion metrics defined for object-oriented systems cannot be used as indicator for changeability assessment. Their results and the results obtained by us are similar, which is one of the strong rationales for the acceptance of our work. However, to get more confidence, it still required to be tested on real-life large industrial projects.

## 4.6 CONCLUSION

This chapter proposes an approach to predict maintainability of aspect-oriented systems. A metric for changeability has been proposed using terminology and formalism defined in Chapter 2. Possible type of changes in AOS has been determined. Using these feasible changes, an experiment is performed on projects designed in AspectJ and CaesarJ. Changeability of generic AO system is assessed by making changes at code level. Impact of these changes has been evaluated. High change impact of a component means that it will absorb less change. In other words, high change impact means low changeability. In order to validate proposed changeability measurement technique, dependency of a component on other components has also been measured. Number of dependencies has been measured with the help of adjacency matrix, which is one of the existing techniques. Correlation analysis between average change impact and number of dependencies gives a correlation factor value. Correlation factor values in CaesarJ based system is 0.87 and in AspectJ based system is 0.89. These values are high enough to indicate that in high dependent AO system, change impact will be high. That means, a highly dependent AO system will absorb low changes. It has been seen from the existing literature that highly dependent systems lead to poor maintainability, which is also proved by the present work.

We also evaluated numeric values of cohesion metric (*UACoh*) for these components and performed correlation analysis between *ACI* and *UACoh*, to check whether cohesion metric can be used to predict changeability or not. Our results show that cohesion metric cannot be used to predict changeability/ maintainability of AOS. In future work, we have planned to perform correlation analysis among metrics for internal characteristics such as coupling, complexity and size, and external characteristic such as maintainability, which will help to predict maintainability of AOS by analyzing code.



# CHAPTER 5

## REUSABILITY ASSESSMENT IN ASPECT-ORIENTED SOFTWARE SYSTEMS\*

---

### 5.1 INTRODUCTION

Reusable software is in high demand in software industry because reuse reduces software development time, efforts and cost. There are many approaches, salient being Module-Oriented (MO), Object-Oriented (OO), and Component-Based Software Development (CBSD), which are used to develop reusable software. Using these approaches, different levels of reusability such as function reusability, code design reusability, sub-system reusability, and system reusability can be achieved. MO approach provides functional reusability, OO approach provides functional as well as code design reusability by its inheritance property and CBSD provides system/sub-system level reusability. As components are black boxes, code reusability is not possible in CBSD. Researchers and practitioners have proposed many models/frameworks for assessing reusability at function level (Selby, 1989); at design/code level (Kitchenham and Kari, 1993; Mili *et al.*, 1995; Li, 1998; Devanbu *et al.*, 1996) and at system/sub-system level (Gill, 2003; Dumke and Schmietendorf, 2000; Washizaki *et al.*, 2003; Boxall and Araban, 2004; Rotaru *et al.*, 2005).

In a software system developed using OO technology, where code related to a concern/feature is scattered in multiple classes, *reusability* and *maintainability* are

\*The major finding of the chapter has been communicated to Journal of Software Quality (Springer)

affected. Such concerns are called crosscutting concerns. Solution of this problem has been proposed by developers of AO technology. AO technology is a new programming paradigm that encapsulates concerns that crosscut the modularity of traditional programming mechanisms. In AO technology, code related to crosscutting concerns is modularized in a single module called Aspect. AO technology cannot stand on its own (Kiczales *et al.*, 2001), aspects are integrated with primary concerns (classes in OO technology). So, function level and code level reusability is already part of AO technology. Code, which earlier was scattered in primary concerns, is now modularized in an aspect and later on the same code will be integrated with classes that must support concern policy by a process known as *weaving*. Weaving injects the code of an aspect into well defined locations (*joinpoints*) in the syntactic structure of primary concerns. This way reusability of aspect code is achieved.

Sant'Anna *et al.* (2003) and Reis (2002) have proposed models to assess reusability of AO systems, but these are mainly theoretical attempts. This Chapter proposes a reusability model for AO systems. Reusability depends on some quality characteristics such as *understandability*, *modularity*, *portability* and so on, which cannot be quantified directly but can be indirectly measured using architectural metrics, which are directly measurable either through manual process or using automated tools. Metrics for *cohesion*, *coupling*, *size* and *complexity*, are examples of such direct measurable metrics. These metrics specific to generic AO systems have been defined in Chapter 2.

## 5.2 REUSABILITY METRICS AND MODELS

Reusability can measure the degree of features that are reused in building applications. There are many metrics (Banker *et al.*, 1991, Karunanithi and Bieman, 1993; Devanbu *et al.*, 1995; Frakes and Terry, 1996; Barnard, 1998; Kamiya *et al.*, 1999; Aggarwal *et al.*, 2005c) available for measuring the reusability for object-oriented systems. These metrics focus on the object structure, which reflects on each individual entity such as methods and classes, and on the external attributes that measure the interaction among entities such as coupling and inheritance. But there are some difficulties in applying existing object-oriented metrics into the development of AOS.

Object-oriented metrics cannot be used to measure the AO component's quality. The reason may be that in AO development, crosscutting concerns are modularized in separate module, which has different type of code characteristics than that of class in OO development. Modularization of crosscutting concerns improves maintainability and reusability features.

Sant'Anna *et al.* (2003) proposed a framework for the reuse and maintenance of aspect-oriented software. In their framework, new metrics suite for measuring cohesion, coupling, size and crosscutting concerns related attributes for AspectJ has been proposed. This metrics suite further has been used to evaluate quality characteristics which are not directly measurable and can be used to assess reusability and maintainability. In order to collect proposed metrics values, a tool called Together 6.0 is used. This tool is for collecting metrics automatically for OO systems. Their metrics suite has not included all features of AO technology. For example, metrics defined for coupling are Coupling Between Components (CBC) and Depth of Inheritance Tree (DIT); it is not mentioned whether inheritance has been considered or not, how implicit invocations will be counted, and so on. Reusability of AO system not only depends on *understandability* and *flexibility* (that they have considered), but also depends on some other factors such as *modularity* and *maintainability*.

Cunha *et al.* (2006) presented a collection of well-known high level concurrency patterns and mechanisms, coded in AspectJ. Benefits of use of AspectJ in all the examples of the study have been shown in terms of *modularity*, *reusability*, *understandability* and *unplug-ability*. This is a comparative study between Java and AspectJ implementations. This framework lacks quantitative validation.

Zhang *et al.* (2008) incorporated AO technique in the reusable design and implementation of connectors, which are becoming more and more significant for software architecture. In their study, discussed, how the reusability of connector is effected, which are having crosscutting concerns. Implementation of AO approach has been proposed to improve reusability of connectors at architectural level. Their work is basically comparative study between reusable connector with and without AO technique

implementation, where crosscutting attributes are present. There are no guidelines that how to assess reusability in AO system?

Aljasser *et al.* (2009) proposed a new language called ParaAJ which is an extension of AspectJ (Kiczales *et al.*, 2001) with the ability to parameterize aspects. This language claims that the use of parameters is a step further towards modularizing aspects, and increases the chance of reusing them. Since there is a lack of an explicit interface between classes and aspect, by using parameters, explicit interface can be created between aspect and their target classes, which can be checked at compile time to insure correctness of aspect-class integration. ParaAJ is in  $\beta$ -version that is why it is incomplete and has not been validated.

### 5.3 ASPECT-ORIENTED METRICS SUITE

The proposed metrics suite describes design and code information in terms of internal software characteristics such as *coupling*, *cohesion*, *complexity* and *size*. In order to define suite of metrics, we have used terminologies and formalism for AO systems, defined in Chapter 2. These terminologies have been defined for Java, AspectJ, CaesarJ and Hyper/J programming languages as a generic framework. But scope of this chapter is to determine reusability in AspectJ based projects. Metrics for cohesion, coupling, complexity and size, have already been defined in Chapter 2. In case of complexity metrics, we have used code complexity only and have left interaction complexity, which is also defined in (Kumar *et al.*, 2008b). This is because our proposed model of reusability is for reuse of the code. Size related metrics have been defined using same terminologies.

The proposed metrics are directly measurable, which can be collected either through manual process for small/ medium size projects or using automated tools for big industrial projects. The proposed metrics may be used to measure external software quality characteristics such as *maintainability*, *understandability* and *adaptability* which cannot be measured directly.

## 5.4 PROPOSED REUSABILITY MODEL

In order to assess reusability of any software system, we need to know the quality characteristics on which reusability depends. An empirical study was conducted by the National Institute of Standards and Technology (NIST) (Salamon and Wallace, 1994), which was on providing measurement information for determining the reusability of software. The report prepared by NIST researchers identified ten indirect quality characteristics for software products based on extensive research in the available literature, which also includes OO technology. The characteristics are (i) understandability (ii) modularity (iii) portability (iv) reliability (v) completeness (vi) maintainability (vii) adaptability, flexibility, expandability (viii) efficiency (ix) completeness, and (x) generality. As AO technology is an extension of OO technology and has some new features, some of the above mentioned quality characteristics may require redefinition in context of AO technology. For example, modularity characteristic in OO technology is encapsulation of data and operations in a single module called class, which is also part of AO technology. In addition to this type of modularity, AO technology also provides modularity of crosscutting concern which is implemented in a single module called aspect. Aspect reduces amount of code and improves maintainability and reusability of crosscutting concerns.

All the above quality characteristics may not have importance in assessing reusability of software systems developed in a language like AspectJ. The important characteristics, which contribute in assessment of reusability of AO systems, cannot be measured directly. In order to provide solution to this problem, we proposed a reusability quality model which is given in Figure 5.1. Using this proposed model, external quality characteristics might be measured indirectly using direct measurable internal characteristics (architectural metrics) such as *complexity*, *cohesion*, *coupling*, and *size*. The reusability model establishes the relationship among external quality characteristics, internal characteristics and the metrics for internal characteristics.

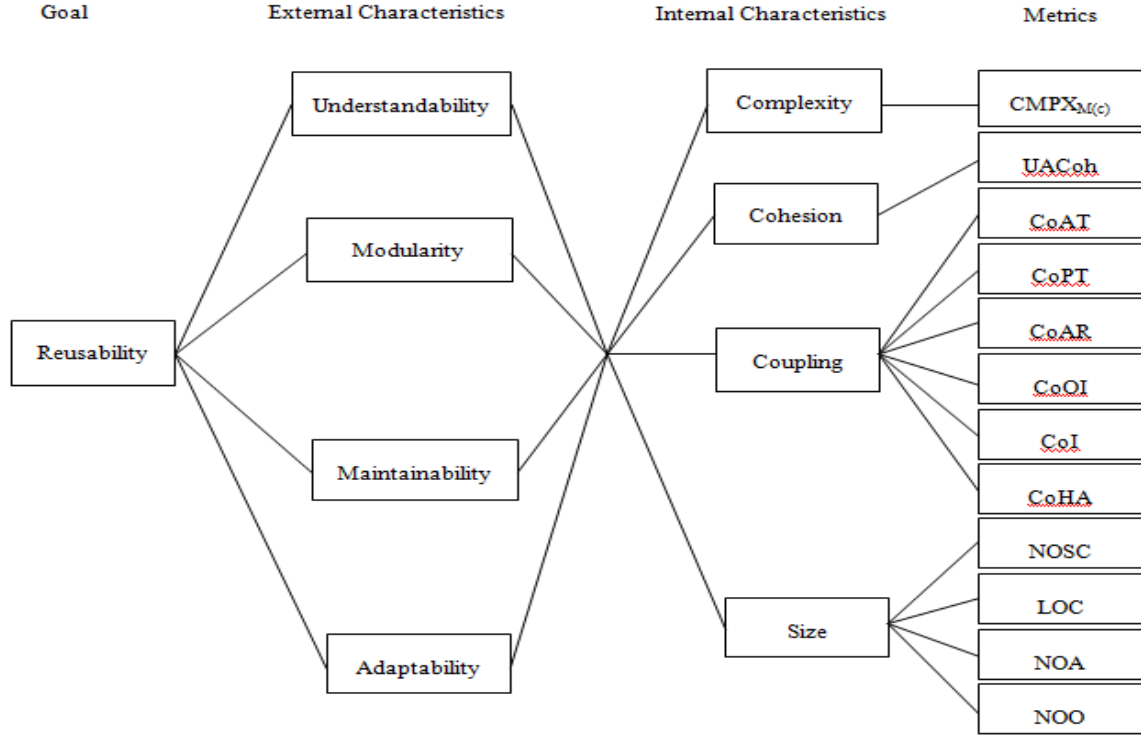


**Figure 5.1 Relationship of Reusability with External and Internal Characteristics**

## 5.5 EMPIRICAL ASSESSMENT

In the proposed reusability model, we are interested in determining reusability of design and code of AO systems. In order to determine external characteristics on which reusability of AO system depends, we conducted a survey. This survey was conducted on 6 different experts. Out of these 6 experts, 2 were Ph.D. students who had good knowledge of AO system and software engineering and 4 were M.Tech. (Computer Science & Engineering) students who are doing their thesis work in the area of Aspect-Oriented Technology. In our survey, we considered all ten characteristics (Salamon and Wallace, 1994) which contribute in assessment of reusability of AO system. All the six experts graded 10 characteristics out of five grade points. It was a class room environment where survey was conducted. From the sample of the survey, we found contribution of some external characteristics in reusability assessment as negligible. Only four external characteristics *understandability*, *modularity*, *maintainability* and

*adaptability* were found as main external characteristics which contribute in assessment of reusability of AO systems, given in Figure 5.1.

In order to quantify understandability, modularity, maintainability and adaptability, we considered medium size projects (Laddad, 2003) developed in AspectJ. Quantification of these external characteristics was done through grading on 5 point scale by the experts, which were member of the survey team. Although, the terminologies used in the present Chapter are same as defined in Chapter 2, but experts were of AspectJ and are not of expert of CaesarJ and Hyper/J, this is the reason of considering the AspectJ based projects only. The mean of graded values by experts for all the components of the project have been taken as values of external characteristics and these values are based on knowledge and experience of the experts.

We can determine metrics values for coupling, cohesion, complexity and size either by automated tool or by manual process. For large industrial projects, automated tool is required and for small/ medium size projects, we can determine the proposed metrics manually. In order to measure  $CMPX_{M(C)}$  metric, we have developed a fuzzy logic tool (Kumar *et al.*, 2010a) which gives this metric value between 0 and 1. Remaining metrics i.e. metrics for coupling, cohesion and size are based on counting which have been determined manually. The experimented projects are medium size projects and are not industrial projects.

After evaluating indirect external quality characteristics (understandability, modularity, maintainability and adaptability) using human survey tool and direct measurable metrics values for internal characteristics, (complexity by using fuzzy logic tool, and remaining by manual process), co-relation between external characteristics and internal characteristics has been evaluated using DataFit tool.

## 5.6 EXPERIMENTAL CASE STUDY

In order to validate the proposed model, we performed experiment using two medium/small size projects developed in AspectJ. First project is having 15 components

and second project is having 10 components. Second project is also having data base handling business logic. We have determined direct measurable metrics for coupling, cohesion, complexity, and size by the approach described in section 5.5. Values of these metrics for all components are listed in Table 5.1 against the name of the component. Indirect measurable external metrics *understandability*, *modularity*, *maintainability* and *adaptability* have been evaluated through survey tool on same set of projects.

Average of samples collected for external characteristics have been listed in Table 5.2. Validity of values for external characteristics is based on experts' knowledge and experience.

Indirect measurable characteristics listed in Table 5.2 for all the components have been plotted and shown in Figure 5.2. The line graphs in Figure 5.2 are consistent with each other, which indicate that every characteristic has same trend as the other characteristics for the same component. In other words, when one external characteristic is increasing for one component, other external characteristics are also increasing for the same component. This result provides strong support for the choice of these characteristics for assessing reusability of a component.

We have also plotted internal characteristics/ metrics for all the components listed in Table 5.1. In order to plot line graphs, we have normalized all the metrics' values in the range 0 to 1. UACoh and  $CMPX_{M(c)}$  are already in the range 0 to 1. After normalizing internal metrics, we have plotted graphs for these metrics values for all the components, which are given in Figure 5.3 and Figure 5.4.

| Component Name | Internal Characteristics |      |      |      |      |     |       |       |     |     |
|----------------|--------------------------|------|------|------|------|-----|-------|-------|-----|-----|
|                | UACoh                    | CoAT | CoPA | CoAR | CoOI | CoI | CMPXM | LOC   | NOA | NOO |
| Account        | 0                        | 0    | 0    | 0    | 0    | 0   | 0     | 0.125 | 0   | 0   |

|                                 |      |      |      |      |      |     |      |      |      |      |
|---------------------------------|------|------|------|------|------|-----|------|------|------|------|
| AccountSimpleImpl               | 0.33 | 0.33 | 0.16 | 0    | 0    | 0.5 | 0.71 | 0.48 | 1    | 0.75 |
| CheckClearanceSystem            | 0    | 0    | 0.33 | 0    | 0.13 | 0   | 0.31 | 0.16 | 0    | 0.25 |
| CheckingAccount                 | 0    | 0    | 0    | 0    | 0    | 0.5 | 0    | 0.05 | 0    | 0    |
| CheckingAccountSimpleImpl       | 0    | 0.33 | 0.33 | 0.29 | 0.04 | 1   | 0.31 | 0.17 | 0    | 0.25 |
| Customer                        | 0.3  | 0    | 0.33 | 0    | 0.08 | 0   | 0.5  | 0.39 | 1    | 0.75 |
| InsufficientBalanceException    | 0    | 0    | 0    | 0    | 0    | 0.5 | 0.11 | 0.09 | 0    | 0.13 |
| LogAccountActivities            | 0.5  | 0    | 0.5  | 0.43 | 0.48 | 0.5 | 0.31 | 0.42 | 0    | 0.13 |
| SavingsAccount                  | 0    | 0    | 0.17 | 0    | 0.04 | 0.5 | 0    | 0.05 | 0    | 0    |
| SavingsAccountSimpleImpl        | 0.29 | 0.33 | 0    | 0    | 0.04 | 0   | 0.31 | 0.17 | 0    | 0.25 |
| Test                            | 0    | 1    | 0    | 1    | 1    | 0   | 0.5  | 0.36 | 1    | 0.13 |
| LogInsufficientBalanceException | 0    | 0    | 0.17 | 0    | 0.26 | 0   | 0.11 | 0.17 | 0    | 0.13 |
| IndentedLogging                 | 1    | 0    | 0    | 0    | 0.30 | 0   | 0.51 | 0.28 | 0.67 | 0.38 |
| AbstractDebitRulesAspect        | 0    | 0    | 0.33 | 0    | 0.08 | 0   | 0.31 | 0.19 | 0    | 0.25 |
| RuleEngineBasedDebitRulesAspect | 0.3  | 0    | 1    | 0.43 | 0.74 | 0.5 | 0.71 | 0.97 | 0.67 | 0.63 |
| Account                         | 0    | 0    | 0    | 0    | 0    | 0   | 0    | 0.12 | 0    | 0    |
| AccountJDBCImpl                 | 0.4  | 0    | 0.5  | 0    | 0.13 | 0.5 | 0.5  | 1    | 0.33 | 1    |
| DatabaseHelper                  | 0    | 0    | 0.67 | 0    | 0.13 | 0   | 0.31 | 0.30 | 0    | 0.25 |
| InsufficientBalanceException    | 0    | 0    | 0.17 | 0    | 0.04 | 0.5 | 0.11 | 0.09 | 0    | 0.13 |
| InterAccountTransferSystem      | 0    | 0    | 0.33 | 0    | 0.09 | 0   | 0.31 | 0.25 | 0    | 0.25 |
| PreserveCheckedException        | 0    | 0    | 0    | 0.14 | 0.22 | 0   | 0.11 | 0.23 | 0    | 0.13 |
| Test                            | 0    | 0    | 0.33 | 0    | 0.17 | 0   | 0.5  | 0.17 | 0.67 | 0    |
| TransactionLogging              | 0.2  | 0    | 0.83 | 0.29 | 0.52 | 0.5 | 0.31 | 0.70 | 0    | 0.5  |
| IndentedLogging                 | 1    | 0    | 0    | 0.14 | 0.35 | 0   | 0.51 | 0.28 | 0.33 | 0.38 |
| DBCTransactionAspect            | 0.43 | 0    | 0.17 | 0.43 | 0.61 | 0   | 0.51 | 0.89 | 0    | 0.63 |

**Table 5.1 Metrics Values for Internal Characteristics**

| Component Name                  | External Characteristics |            |                 |              |
|---------------------------------|--------------------------|------------|-----------------|--------------|
|                                 | Understandability        | Modularity | Maintainability | Adaptability |
| Account                         | 1.17                     | 1          | 1.17            | 1.33         |
| AccountSimpleImpl               | 2.67                     | 2.83       | 2.5             | 2.33         |
| CheckClearanceSystem            | 1.83                     | 2          | 2.17            | 1.67         |
| CheckingAccount                 | 1.17                     | 1          | 1.33            | 1.5          |
| CheckingAccountSimpleImpl       | 2.50                     | 1.83       | 2.17            | 1.33         |
| Customer                        | 4.17                     | 3.67       | 3.83            | 3.5          |
| InsufficientBalanceException    | 1                        | 1          | 1               | 1.17         |
| LogAccountActivities            | 2.17                     | 2.33       | 2               | 2.33         |
| SavingsAccount                  | 1                        | 1.5        | 1               | 1.33         |
| SavingsAccountSimpleImpl        | 1.83                     | 2.33       | 2               | 2.33         |
| Test                            | 2.33                     | 2.17       | 2.33            | 2            |
| LogInsufficientBalanceException | 2                        | 1.83       | 1.5             | 1.17         |
| IndentedLogging                 | 2.17                     | 3.17       | 3               | 2.67         |
| AbstractDebitRulesAspect        | 2.17                     | 3.17       | 3.67            | 2.67         |
| RuleEngineBasedDebitRulesAspect | 3.83                     | 4.83       | 4.83            | 3.83         |
| Account                         | 1.17                     | 1          | 1.17            | 1.33         |
| AccountJDBCImpl                 | 2.83                     | 3.17       | 3.33            | 2.17         |
| DatabaseHelper                  | 1.83                     | 2.17       | 2               | 2.33         |
| InsufficientBalanceException    | 1.67                     | 1.83       | 1.33            | 1.5          |
| InterAccountTransferSystem      | 1.5                      | 1.17       | 1.83            | 1.17         |
| PreserveCheckedException        | 2.5                      | 2.83       | 3.83            | 3.33         |
| Test                            | 1.33                     | 1.67       | 1.17            | 1.33         |
| TransactionLogging              | 3.17                     | 3.67       | 2.83            | 3.67         |

|                      |      |      |      |      |
|----------------------|------|------|------|------|
| IndentedLogging      | 4    | 4.17 | 3.5  | 3.33 |
| DBCTransactionAspect | 3.83 | 4.5  | 3.83 | 4    |

**Table 5.2 Metrics Values for External Characteristics**

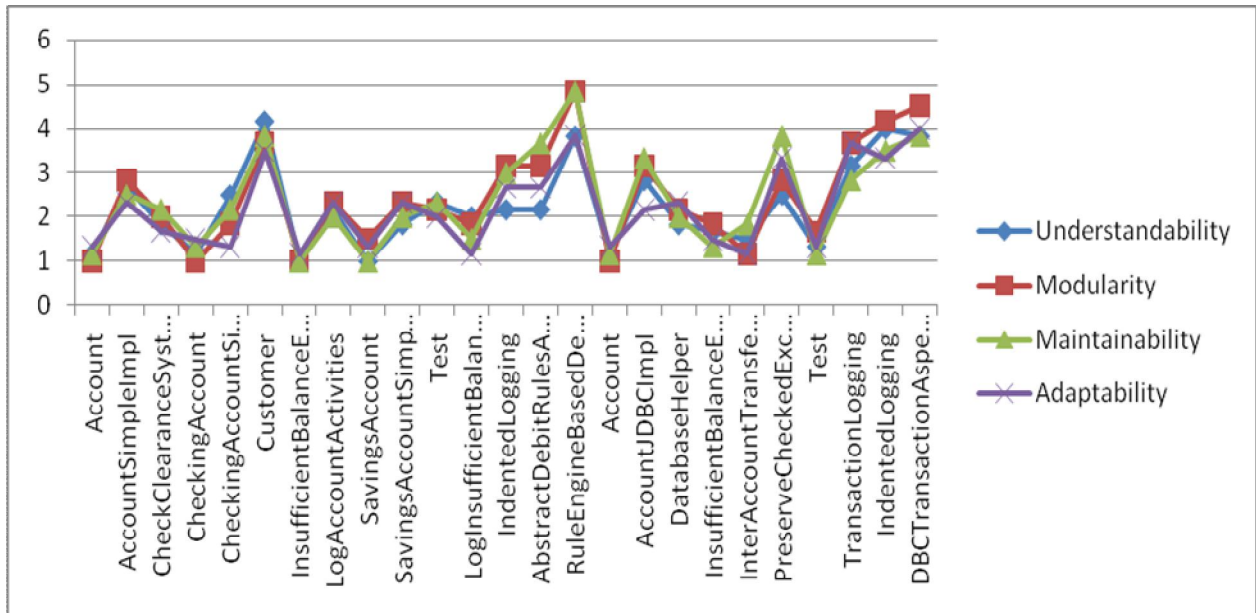


**Figure 5.2 Line Graphs of External Characteristics**

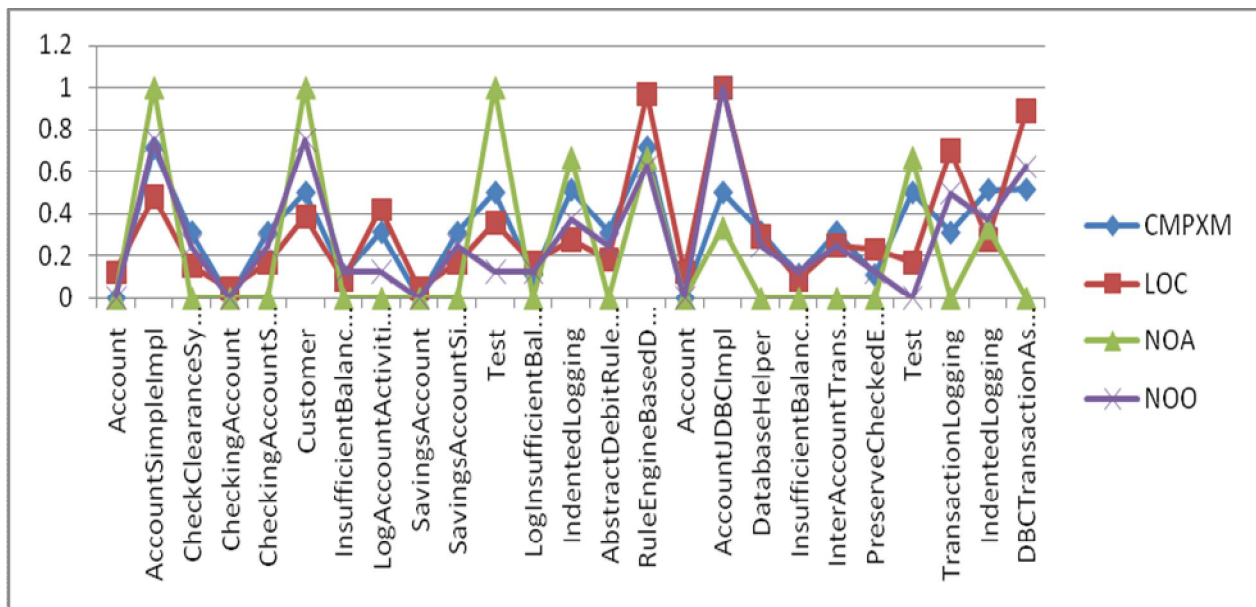


**Figure 5.3 Line Graphs of Internal Characteristics**

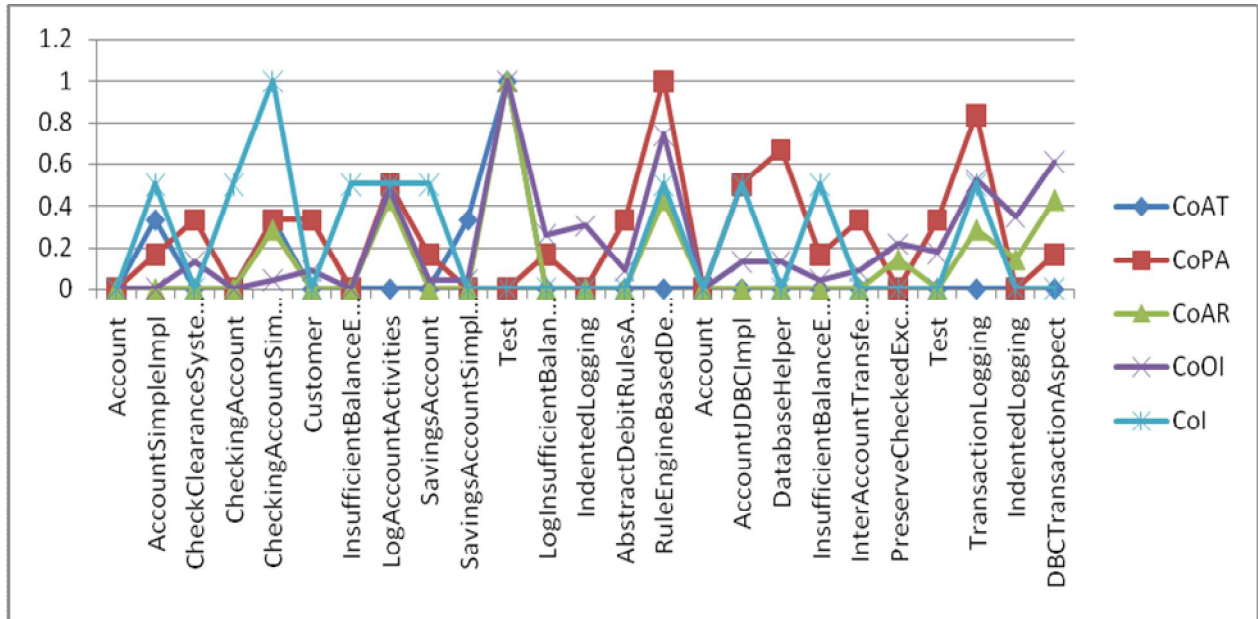


**Figure 5.4 Line Graphs of Internal Characteristics**

We may draw the following inferences from this analysis:

- i. Values for metric CoHA are zero for all the components, that may be because of components of experimental projects are not having high level association. We have not considered this metric in the study.
- ii. Our study is based on assessment of reusability of individual component that is why determination of internal and external metrics will also be at component level. So, metric NoSC, which can be evaluated at system level, is not considered in this study.
- iii. High value of LOC count is usually accompanied by high measures for NOO and  $CMPX_{M(c)}$ .
- iv. CoOI metric values are higher in case of component as Aspect compared to component as class/interface.
- v. There is no consistency in the measures collected for UACoh in every component explored in this study.
- vi. The measures collected for  $CMPX_{M(c)}$ , NOO, and LOC are proportional to the

- levels of external characteristics *understandability*, *modularity*, *maintainability* and *adaptability* for the components. The correlation coefficients of  $CMPX_{M(c)}$ , NOO and LOC with all four external characteristics *understandability*, *modularity*, *maintainability* and *adaptability* shows strong correlation.
- vii. The correlation coefficients for CoAT, CoPT, CoAR and NOA with all four external characteristics *understandability*, *modularity*, *maintainability* and *adaptability* do not show strong relation.
  - viii. The measures collected for CoI are inversely proportional to the levels of external characteristics *understandability*, *modularity*, *maintainability* and *adaptability* for the components.
  - ix. The measures collected for CoOI are proportional to the levels of external characteristics. Correlation coefficient between CoOI and external characteristics shows strong relation in case of components which are aspects but shows weak relation in case of components that are classes.

## 5.7 RESULTS AND DISCUSSIONS

Our experimented results can be used in assessing reusability level of small/medium size AO projects of academic environment. For large industrial projects, there is need to perform similar experiments, which is expected to give similar results. In order to minimize errors in measuring direct measurable internal metrics, there is need to develop automated tool, instead of measuring manually, especially for large industrial projects. Results obtained from the above experiment can be interpreted as follows:

- i. Metrics  $CMPX_{M(c)}$ , NOO, LOC, and CoOI (when component is an aspect) can be used as indicators for assessment of reusability level of the component. For any component, increase in value of  $CMPX_{M(c)}$ , NOO, LOC, and CoOI (when component is an aspect) shows increase level of reusability of the same component.
- ii. Inheritance level in our experiment is 2 and the measure collected for CoI are inversely proportional to the levels of external characteristics *i.e.* inversely

- proportional to the reusability level of AO components. It means, more dependence on other inherited components, more difficult to reuse than stand-alone or self-contained components.
- iii. There is no strong relation between CoAT, CoPT, CoAR, NOA, and CoOI (when component is as class) and external characteristics. It shows that these metrics cannot be used as indicators for assessing reusability.
  - iv. Metric NoSC is measured at system level. We have focused on reusability of individual AO components. So, NoSC cannot be used an indicator of reusability.
  - v. As there is no consistency in the measures collected for UACoh, it is not recommended as being used for assessing reusability.

## 5.8 CONCLUSION

In this chapter, we have proposed a model for reusability assessment of AO components. As per the report provided by NIST (Salamon and Wallace, 1994), which is based on extensive research available in literature, reusability of any software product can be determined by 10 external quality characteristics, (i) understandability (ii) modularity (iii) portability (iv) reliability (v) completeness (vi) maintainability (vii) adaptability, flexibility, expandability (viii) efficiency (ix) completeness, and (x) generality. These external characteristics cannot be measured directly by seeing the code of software, developed using different approaches, like functional programming, OOP and AOP. But internal characteristics like coupling, cohesion, complexity and size can be evaluated directly from the code. For AO technology, we conducted a survey to determine external characteristics, which is based on grade point between 1 to 5. From the survey, we found that there are 4 main indirect measurable external characteristics, which play main role in determining reusability of AO components and other external characteristics play negligible role. These 4 characteristics are *understandability*, *modularity*, *maintainability* and *adaptability*. These external indirect measurable characteristics can also be evaluated by performing correlation analysis between all these 4 characteristics with all direct measurable metrics for coupling, cohesion, complexity and size for AO technology. In other words, reusability of AO components can be

determined by direct measurable metrics for internal characteristics. Metrics for coupling, cohesion, complexity and size can be evaluated manually or through automated tools. For these direct measureable metrics, there is no automated tool for AOS, so we have determined these manually, which is feasible in present study because the experimented projects are small/medium in size. A regards large project, automated tools would be required.

From this study, we found that increase in the value of  $CMPX_{M(c)}$ , NOO, LOC, and CoOI (when component is an aspect) indicates an enhanced level of reusability of AO components. Increase in the value of CoI indicates diminished level of reusability of AO components. Metrics CoAT, CoPT, CoAR, NOA, CoOI, NoSC, and UACoh cannot be used as indicators of reusability of AO components.





# CHAPTER 6

## QUALITY MODEL\*

---

### 6.1 INTRODUCTION

Now-a-days, software systems are more difficult to develop due to their complex structure and dynamic nature. Due to frequent change in business rules and high competition in the market, modifications/changes in software systems are required frequently. In such situation challenges involved in development of software system include maintainability of software system, dynamic and reusable architectural model, devising quality software, accurately controlling overhead cost, complying with the progress schedule, controlling complexity of the software, and most importantly satisfying consumer demand with respect to software quality (Khoshgoftaar *et al.*, 2006). There are many architectural models available to develop a software system. Module-oriented (MO) and object-oriented (OO) are most widely used approaches to develop a software system, which do not account for all the requirements of current and complex software systems. For this reason aspect-oriented software development (AOSD) approach emerged, which provides solution for the system where crosscutting concerns are present (Lieberher *et al.*, 2001).

**\*The major findings of the chapter has been published in ACM SIGSOFT SEN, 34(5), pp: 1-9, Sep. 2009.**

Aspect-oriented technology is a new programming paradigm that encapsulates concerns that crosscut the modularity of traditional programming mechanisms. According to Alexander (2003), among the claimed benefits of this new technology is the reduction in the amount of code written and higher cohesion. In addition to these, aspect-oriented technology has benefit of costs. Quality of AO system can also be measured with the help of internal characteristics like cohesion (Kumar *et al.*, 2008a), coupling (Kumar *et al.*, 2009) and complexity (Kumar *et al.*, 2008b).

In order to measure quality of software systems, researchers and practitioners have proposed metrics and models. Software metric is a qualitative indicator of any attribute of software system and model specifies relationship among these metrics. In the late seventies McCall *et al.* (1977) and in the late eighties Boehm *et al.* (1976) proposed software quality models which formed the basis for project managers to monitor risks and evaluate quality. When object-oriented technology was getting enough maturity level, in early nineties ISO/IEC 9126 (2001) software quality standards were proposed which put emphasis on maintainability. Object-oriented technology's inheritance property emphasized reuse, Dromey (1995) added reusability to the six ISO/IEC 9126 characteristics. IEEE has also taken initiative in publishing the Software Engineering Body Of Knowledge (swebok 2001), which is the first major attempt to collect, classify and standardize the terms models, methods and techniques. There are several other researchers and practitioners (Gill, 2005; Aggarwal *et al.*, 2007; Singh, 2008), who also provided guidelines to measure quality of different types of software systems.

Aspect-oriented approach is a new paradigm and has different type of abstraction. Since AO technology cannot stand on its own (Kiczales, 1998) and aspectual modules are integrated with MO modules or with OO classes, most of the software quality characteristics/sub-characteristics of ISO/IEC 9126 will be applicable to AO quality model. Because of new abstraction type in AO technology, some new software quality characteristics/sub-characteristics are required to be added, which can describe new features of AO technology. We have derived our **Aspect-Oriented Software Quality Model** (AOSQUAMO) from ISO/IEC 9126 in which *modularity*, *code-reducibility*,

*complexity* and *reusability* have been added as new quality sub-characteristics under characteristics, *maintainability*, *efficiency*, *usability*, and *functionality* respectively.

In order to evaluate quality of AOSQUAMO as single unit, we have applied Analytic Hierarchy Process (AHP), which has been extensively adapted in multi-criteria decision making. AHP has successfully been applied to many practical decision making problems (Saaty, 1980).

The chapter is structured as follows: Section 6.2 describes some existing software quality models. Section 6.3 presents new AO software quality model. After defining new quality model, section 6.4 defines analytic hierarchy process and its application to new proposed model. Conclusion is presented in section 6.5.

## **6.2 SOFTWARE QUALITY CHARACTERISTICS AND MODELS**

There are many software quality models that suggest ways to tie together different quality characteristics. Each model helps in understanding how several quality factors contribute to the overall quality. Various researchers and practitioners have proposed their quality models to relate user's external views to the developer's internal views for the software products. Some of the popular quality models are as follows:

### **6.2.1 McCall *et al.* Model**

McCall *et al.* (1977) proposed a software quality factor framework and classified the quality attributes into three broad categories: (i) Product Operation Factors, (ii) Product Revision Factors, and (iii) Product Transition Factors.

- Product Operation Factors: the factors which contribute to product operation are *correctness*, *reliability*, *efficiency*, *integrity*, and *usability*.
- Product Revision Factors: the factors which contribute to product revision are *maintainability*, *flexibility*, and *testability*.
- Product Transition Factors: the factors which contribute to product transition are *portability*, *reusability*, and *interoperability*.

### 6.2.2 Boehm *et al.* Model

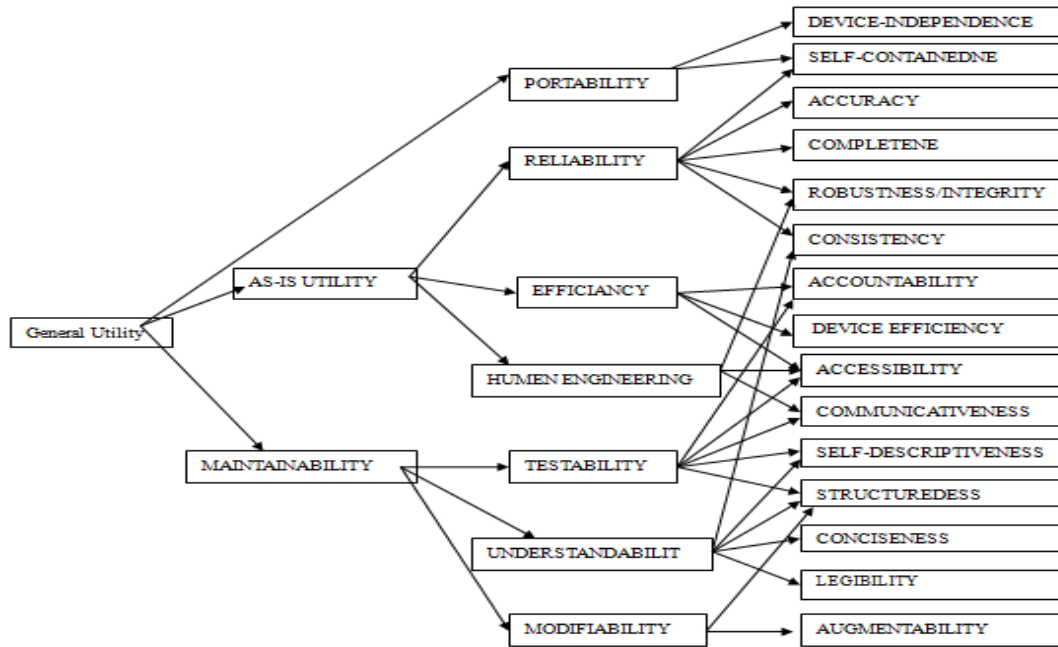


**Figure 6.1 Boehm Quality Model.**

Boehm *et al.* (1976) model is similar to McCall model. Boehm represented their quality model as a hierarchical tree and broke quality characteristics into sub-characteristics, which is given in Figure 6.1. Boehm also included the hardware yield characteristics which were not considered in McCall model. Using this model, quality as a single unit can be evaluated due to hierarchical nature of the model. This model does not provide guidelines to measure listed characteristics.

### 6.2.3 ISO/IEC 9126 Software Quality Model

ISO/IEC 9126 (2001) determines three software quality aspects: (i) process quality, (ii) product quality, and (iii) product in use quality. ISO/IEC 9126 series standards have introduced a hierarchical model with six major quality characteristics. These quality characteristics are divided into 21 sub-characteristics which contribute to internal quality. ISO/IEC 9126-1 is concerned primarily with definition of quality characteristics and sub-characteristics in the final products. ISO/IEC 9126-2 provides external metrics for measuring attributes of six external quality characteristics defined

in ISO/IEC 9126-1. ISO/IEC 9126-2 defines external metrics; ISO/IEC 9126-3 defines internal metrics and ISO/IEC 9126-4 defines quality in use metrics for measurement of quality characteristics or the sub-characteristics. Internal metrics measures software itself, external metrics measures behavior of the computer-based system that include the software, and quality in use metrics measure the effects of using the software in a specific context. These are listed in Table 6.1.

| Quality Type                | Characteristics | Sub-characteristics |
|-----------------------------|-----------------|---------------------|
| Software Product<br>Quality | Functionality   | Suitability         |
|                             |                 | Accuracy            |
|                             |                 | Interoperability    |
|                             |                 | Compliance          |
|                             |                 | Security            |
|                             | Reliability     | Maturity            |
|                             |                 | Fault tolerance     |
|                             |                 | Recoverability      |
|                             | Usability       | Understandability   |
|                             |                 | Learn-ability       |
|                             |                 | Operability         |
|                             | Efficiency      | Time behavior       |
|                             |                 | Resource behavior   |
|                             | Maintainability | Analyzability       |
|                             |                 | Changeability       |
|                             |                 | Stability           |
|                             |                 | Testability         |
|                             | Portability     | Adaptability        |
|                             |                 | Install-ability     |
|                             |                 | Replace-ability     |
|                             |                 | Conformance         |

**Table 6.1 Quality Characteristics of the ISO/IEC 9126.**

ISO/IEC 9126-2:2003 is intended to be used together with ISO/IEC 9126-1. ISO/IEC 9126-4 contains: (i) an explanation that how to apply software quality metrics, (ii) a basic set of metrics for each characteristic, and (iii) an example of how

to apply metrics during the software product life cycle. ISO/IEC 9126 quality model is not covering some important quality characteristics, which contribute to the quality of the AOP.

#### **6.2.4 Dromey's Model**

Dromey (1995) proposed a quality model that provides an explicit process for building quality-carrying properties into software. This model establishes the link between tangible product characteristics and less tangible attributes. This model guides us where to look for defects and also indicates the properties that will need to be violated to create defects. This model addresses product quality by defining all the related sub-characteristics in such a way that they can be merged and amalgamated into higher level characteristics. The model supports make of quality software, definition of language-specific coding standards, systematic classifying quality defects, and the development of automated code auditors for detecting quality defects in software. Dromey added *reusability* as a quality characteristic in his model.

#### **6.2.5 Other Models**

Deutsch and Wills (1998) categorized software quality as software procedure quality and software product quality. As a characteristic of software product development process, software procedure quality consists of software engineering related elements as: *technology, tools, personnel, organization, and equipment*. As a characteristic of software product, software product quality includes *document clarity, integrity, design traceability, organization, program reliability, and test integrity*. In this model, guidance has been provided on what step to follow in order to obtain desired product.

Ward and Venkataraman (1999) suggested that software quality measures may include one or more of the following: (i) User-based: as evaluated by users, software quality refers to the degree of satisfaction of user expectations; (ii) Product delivery-based: evaluated by the product as evaluated by the designer, the degree of system effectiveness and program maintainability; (iii) Manufacturing-based: the development

process, emphasizing quality control and management; and (iv) Organizational control-based: project costs, production time, resource control and risk management.

Sharma *et al.* (2008) proposed a quality model in context of Component-Based Software Development (CBSD) and have derived their model from ISO/IEC 9126. In their model, they included *reusability*, *complexity*, *track-ability*, and *flexibility* new sub-characteristics in the six characteristics of ISO/IEC 9126. In order to evaluate total quality of a component, they used analytic hierarchy process (AHP) and for the weight values of quality characteristics and sub-characteristics, a survey was conducted.

Chang *et al.* (2008) provided guidelines to evaluate software quality by integrating fuzzy theory and AHP. They applied this new concept on ISO/IEC 9126 quality model. Instead of taking mean of collected sample data, they applied fuzzy theory to get relative weights of characteristics and sub-characteristics. They have not proposed any new quality model, but have provided guidelines to evaluate software quality using existing models.

### **6.3 ASPECT-ORIENTED SOFTWARE QUALITY MODEL (AOSQUAMO)**

Most of the software quality models, which are proposed after definition of software quality standards ISO/IEC 9126, have been derived from ISO/IEC 9126. For example, Rawashdeh and Matalkah (2006) have added *compatibility* as a sub-characteristic to functionality, *complexity* as a sub-characteristic to usability and *manageability* as a sub-characteristic to maintainability in their software quality model. They removed sub-characteristics *stability* and *analyzability* from maintainability. They also added a new characteristic in their quality model as *stakeholders*, who are the members of the team responsible for developing, maintaining, integrating and using COTS system. This model focuses in measurement of quality of component-Based System. Bertoa and Vallecillo (2002) proposed a quality model, which defines characteristics and sub-characteristics in Component-Based Systems. In this model, sub-characteristics have been divided into runtime and lifecycle categories based on their

nature. They also added *capability* as a sub-characteristic to functionality, which indicates whether former version of the component is compatible with its current version.

In order to define software quality model, which should cover all the features of AO software system, we have to look at what are the new features and limitations of the new technology. As AO technology is an extension of MO or OO technology. AO programming languages are also extensions of native programming language such as AspectC is an extension of C, AspectC++ is an extension of C++, AspectJ is an extension of Java, CaesarJ is an extension of Java, AspectXML is an extension of XML and so on. Since AO technology cannot exist on its own, it will have all the features of the technology from which it is derived. For example, AspectJ has all the features of Java and additional features which has been added in AspectJ. In other words, if AO technology is derived from OO technology, then it will have all the features of OO technology and additional features added to aspectual code. It indicates that AO quality model (AOSQUAMO) can be extended from any standard quality model which is applicable to MO and/or OO methodologies. Additional quality characteristics/sub-characteristics are required to be added, which can cover new features of crosscutting concerns (aspect(s)) and integration of it with primary concerns (classes). Re-definition of existing characteristics/sub-characteristics, in context of AO technology, is also required.

Our AOSQUAMO is an extension of ISO/IEC 9126 quality model. In this quality model, we have added *modularity* as a sub-characteristic under maintainability, *code-reducibility* as a sub-characteristic under efficiency, *complexity* as a sub-characteristic under usability and *reusability* as a sub-characteristic under functionality. The new AOSQUAMO model is given in Table 6.2. In this table, characteristics and sub-characteristics have also been assigned labels like C2 (characteristic number 2) and SC32 (sub-characteristic number 2 of characteristic number 3), which will be referred in following sections. Definition and justification of importance of the new as well as existing (of ISO/IEC 9126) characteristics and sub-characteristics in context of AO technology are as follows:

| Quality Type             | Characteristics    | Sub-characteristics           |
|--------------------------|--------------------|-------------------------------|
| Software Product Quality | Functionality:C1   | Suitability:SC11              |
|                          |                    | Accuracy:SC12                 |
|                          |                    | Interoperability:SC13         |
|                          |                    | Compliance:SC14               |
|                          |                    | Security:SC15                 |
|                          |                    | <b>Reusability:SC16</b>       |
|                          | Reliability:C2     | Maturity:SC21                 |
|                          |                    | Fault tolerance:SC22          |
|                          |                    | Recoverability:SC23           |
|                          | Usability:C3       | Understandability:SC31        |
|                          |                    | Learn-ability:SC32            |
|                          |                    | Operability:SC33              |
|                          |                    | <b>Complexity:SC34</b>        |
|                          | Efficiency:C4      | Time behavior:SC41            |
|                          |                    | Resource behavior:SC42        |
|                          |                    | <b>Code-reducibility:SC43</b> |
|                          | Maintainability:C5 | Analyzability:SC51            |
|                          |                    | Changeability:SC52            |
|                          |                    | Stability:SC53                |
|                          |                    | Testability:SC54              |
|                          |                    | <b>Modularity:SC55</b>        |
|                          | Portability:C6     | Adaptability:SC61             |
|                          |                    | Install-ability:SC62          |
|                          |                    | Replace-ability:SC63          |
|                          |                    | Conformance:SC64              |

**Table 6.2 AOSQUAMO Model**

### 6.3.1 Definition of Proposed New Characteristics

Following are definitions of new characteristics added to AOSQUAMO model:

#### 6.3.1.1 Modularity

AOP explicitly deals with separation of concerns in software development. In particular, it supports modular programming to implement concerns that crosscut the modularity of traditional programming techniques. For example, code that implements particularly *login* policy is distributed across a set of classes and methods that must enforce the policy. However, in AO methodology, the code implementing the *login* policy is extracted from all the classes and is implemented in one module called aspect. Thus, aspect localizes the code that affects the implementation of multiple classes and methods into one cohesive place (Elrad, 2001). Here modularity of crosscutting concerns will benefit, when some changes or modifications are needed in crosscutting concerns related code. It will reduce ripple effects in such type of changes. That means modularity will benefit in maintenance face of the system. So, we propose to add *modularity* as a sub-characteristic under *maintainability* characteristic.

#### 6.3.1.2 Code-reducibility

As discussed earlier, crosscutting concern related code, which is scattered in multiple classes, is implemented in one module in AO technology. It causes reduction in amount of code. The practical consequence of writing aspects is that less code is written. The code that would otherwise be distributed throughout the collection of primary concerns is now localized, thus reducing redundancies. Reduction in amount of code definitely will enhance the efficiency of the system in terms of time and memory. We propose to add *code-reducibility* as a sub-characteristic under *efficiency* characteristic.

#### 6.3.1.3 Complexity

Complexity of aspect-oriented software systems is divided into two categories: (i) code complexity which is static, and (ii) interaction complexity which is dynamic. Code complexity is due to attributes complexity, operations (methods, advices, introductions

etc) complexity and nested classes/aspects complexity. Interaction complexity is due to interaction between classes, interaction between aspects, interaction between class and aspect which may occur because of implicit/explicit operation invocations, exception occurrence, attribute references etc. Some complexity metrics for AO systems have been defined by Kumar *et al.* (2008b). The complexity characteristic can also be useful in efforts estimation that will be needed in developing and maintaining the software system. Less complex software system will be easy to use for any developer that is why; we propose to add *complexity* as a sub-characteristic under *usability* characteristic.

#### **6.3.1.4 Reusability**

Object-oriented software systems have both types of reusability, functional reusability and code reusability. Functional reusability on OO approach is similar as in case of module-oriented approach. Code reusability in OO approach is due to its inheritance property. In aspect-oriented approach, code related to crosscutting concerns is modularized in an aspect, which earlier was scattered in primary concerns (classes). Later on this code would be integrated with the classes that must support this policy by a process known as *weaving*. Weaving injects the code of an aspect into well-defined locations (*joinpoints*) in to the syntactic structure of a primary concern. This is called reusability of aspect code. Since our model is extended from OO technology, it has three levels of reusability: functional reusability, code reusability due to inheritance property, and aspect code reusability due to weaving process. In our AO quality model, we propose to add *reusability* as a sub-characteristic under *functionality* characteristic.

#### **6.3.2 Definition of Existing Characteristics in Context of AO Technology**

The main quality characteristics, *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability* are available in most of the proposed quality models. However, researchers have different opinion while choosing sub-characteristics of these characteristics. Our focus is on evaluation of software product quality but not on quality in use. So we have not discussed here quality in use characteristics. This section defines

the terms used for various characteristics and sub-characteristics in our proposed quality model AOSQUAMO in context of aspect-oriented software systems.

#### 6.3.2.1 Functionality

Functionality is the essential purpose of any product. It is a set of attributes that bear on the existence of a set of functions and their specified properties. It means that classes and aspects should provide functions and services as per the requirement when used under specific conditions. Existing functionalities of classes can be modified/enhanced easily through inheritance property, and crosscutting functionalities can easily be modified through aspect-oriented approach. It will lower the cost and provide faster development of the product.

- i. **Suitability:** Suitability describes how well the aspect fits into classes. As aspect requires integration with classes, developer should know whether the aspect is suitable to use or increases the overall complexity of the system, which may be due to wrong mining of the aspect.
- ii. **Accuracy:** It evaluates whether the system, after implementing aspect(s), is giving accurate results with correct precision level when used under specific condition.
- iii. **Interoperability:** It refers to the ability of diverse systems and organization to work together (inter-operate). Interoperability of AO systems and OO system is same as aspects are integrated with classes.
- iv. **Compliance:** This sub-characteristic describes whether an AO system is conforming to any international standard or certification etc.
- v. **Security:** Objective of security includes protection of information and data from theft, corruption or protecting unauthorized access to the system. Security property can be considered as crosscutting concern as security policy must be enforced to implement wherever applicable. This security concern related code

could be implemented in an aspect, thus the information leakage could be reduced.

### 6.3.2.2 Reliability

Once the AO software system is in function as specified, after delivery, the reliability characteristic defines the capability of the system to maintain its service provision under defined conditions for specified period of time. It is the probability that an AO system will provide failure within a given period of time.

- i. **Maturity:** This sub-characteristic describes how much work has been done using this technology, in other words, how many versions have been released of this technology? AO technology is not matured enough compared to OO technology. Research work is going on in this area and will take time to reach high maturity level.
- ii. **Fault Tolerance:** It describes whether the AO system can maintain a specified level of performance in case fault occurs. This is one of the important characteristic to understand in context of AO system. Fault in AO system may be at four different stages:
  - a. The fault resides in a portion of the primary concern that is not affected by a woven aspect.
  - b. The fault resides in the code that is specific to the aspect and is isolated from the woven context.
  - c. The fault is an emergent property that results from some interaction between the aspect and the primary concern.
  - d. The fault is an emergent property of a particular combination of aspects woven into the primary concern.
- iii. **Recoverability:** It describes capability of AO software to re-establish its level of performance and recover the data directly affected if failure occurs.

### 6.3.2.3 Usability

Usability only exists with regard to functionality and refers to the ease of use for a given class or aspect. It is the ability to understand, learn, use and execution of AOS, when used under specific conditions. Developer will be able to use aspect and will be able to integrate aspect with classes, only when he/she has knowledge of aspect as well as classes.

- i. **Understandability:** This characteristic describes the capability of software to enable users to understand the appropriateness of software and its use for particular tasks and conditions of use. In AO context, it is developer's capability of understanding primary concerns, aspect code implementation, and how to integrate the aspect code with primary concerns.
- ii. **Learn-ability:** This sub-characteristic describes the capability of AO software to enable users to learn its applications.
- iii. **Operability:** This sub-characteristic describes the capability of AO software to enable users to operate and control it.

### 6.3.2.4 Efficiency

This characteristic is concerned with the system resources used when providing the required functionality. The amount of disk space, memory, network, CPU time etc., provides good indication of this characteristic. Efficiency is improved using AO technology as discussed in definition of code-reducibility in section 6.3.1.2.

- i. **Time behavior:** This sub-characteristic describes the capability of AO software to provide appropriate responses, processing time and throughput rates when performing its function under required conditions.
- ii. **Resource behavior:** This sub-characteristic describes the capability of AO software to use appropriate resources in time when the software implements its function in required conditions.

### 6.3.2.5 Maintainability

This characteristic describes the ability to modify a class or aspect or any feature of the system. For a system having crosscutting concern(s), AO technology has advantage over OO technology in terms of maintaining crosscutting concerns. We have added an important sub-characteristic as *modularity* (modularity of crosscutting concern) in *maintainability* characteristic, which claims that maintenance of such system will take less effort compared to OO system. Other sub-characteristics are as follows.

- i. **Analyzability:** This sub-characteristic describes capability of AO software to be diagnosed for deficiencies, causes of failures, identification of the parts requiring modifications which may be in (a) primary concern, (b) aspect, and (c) viewing process i.e. integration of aspect(s) with primary concerns.
- ii. **Changeability:** This sub-characteristic describes capability of AO software to enable a specified modification to be implemented. The guidelines of measuring changeability in AO software systems have been given by Grover *et al.* (2008) and Kumar *et al.* (2007).
- iii. **Stability:** This sub-characteristic refers ability of AO software to minimize unexpected effects from software modifications.
- iv. **Testability:** This sub-characteristic refers ability of AO software to validate modified software. As described in the definition of fault-tolerance in the section 6.3.2.2, alternatives i-iv are likely to cause a increase in the testing efforts required to achieve a given level of quality.

### 6.3.2.6 Portability

This characteristic refers to that how well the software can adopt to changes in its environment or with its requirement. Object-oriented design and implementation practices can contribute to the extent to which this characteristic is present in a given system. Aspect-oriented design and implementation is same as of object-oriented in terms of portability because environment for aspect and object will be same.

- i. **Adaptability:** It refers to the capability of AO software that can be adapted to different specified platforms. AO software extended from Java is intended to have more adaptability because of platform independence.
- ii. **Install-ability:** It is the capability of AO software to be installed easily on different platforms. AO software after viewing and compilation will behave like OO software. So, *install-ability* has nothings new in AO systems.
- iii. **Replace-ability:** It is the capability of AO software to replace others specified in the environment of that software.
- iv. **Conformance:** It is similar to compliance, which is defined above in the section 6.3.2.1, but here, it is related to portability, e.g. conformance to particular database standard.

## 6.4 EVALUATION OF AOSQUAMO USING ANALYTIC HIERARCHY PROCESS

In order to evaluate the quality of any product developed using AOP, the proposed model AOSQUAMO can be used. Quality as a single unit can be evaluated using Analytic Hierarchy Process (AHP). The AHP process as initially proposed by Saaty (1980), only uses the pair-wise matrix to evaluate ambiguity in multi-criteria decision making problems. In this paper,  $n$  elements,  $C_1, C_2, \dots, C_n$  were considered, which were to be compared. Relative weight of  $C_i$  with respect to  $C_j$  denoted as  $a_{ij}$  and a square matrix  $A=[a_{ij}]$  of order  $n$  as given in equation (6.1).

$$\mathbf{A} = [a_{ij}] \quad \begin{matrix} & C_1 & C_2 & \cdot & C_n \\ \begin{matrix} C_1 \\ C_2 \\ \cdot \\ C_n \end{matrix} & \begin{bmatrix} 1 & a_{12} & \cdot & a_{1n} \\ \frac{1}{a_{12}} & 1 & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & \cdot \\ \frac{1}{a_{1n}} & \frac{1}{a_{2n}} & \cdot & 1 \end{bmatrix} \end{matrix} \quad (6.1)$$

where  $a_{ij} = \frac{1}{a_{ji}}$ , for  $i \neq j$ , and  $a_{ii} = 1$  all  $i$ . Such a matrix is said to be reciprocal matrix.

For matrix involving human judgments, the judgments are inconsistent to a greater or lesser degree, in such a case, find vector  $\omega$  satisfying the equation (6.2).

$$\mathbf{A} \omega = \lambda_{\max} \omega, \quad \text{and } \lambda_{\max} \geq n \quad (6.2)$$

where  $\omega$  is Eigen vector and  $\lambda_{\max}$  represents Eigen values. The difference between  $\lambda_{\max}$  and  $n$ , if any, is an indication of inconsistency of the judgments. Saaty (1980) proposed a consistency index ( $CI$ ) and consistency ratio ( $CR$ ) to verify the consistency of the comparison matrix. These are defined as:

$$CI = \frac{\lambda_{\max} - n}{n - 1} \quad (6.3)$$

$$CR = \frac{CI}{RI} \quad (6.4)$$

where,  $RI$  is the average consistency index over numerous random entries of the same order reciprocal matrices. Saaty suggests that if the  $CR$  exceeds 0.1, the set of judgment may be too inconsistent to be reliable. In such situation, a new comparison matrix is required to prepare until  $CR \leq 0.1$ .

#### 6.4.1 Allocating the Weights to Characteristics and Sub-Characteristics

In order to assign weights to characteristics and sub-characteristics of proposed model, we conducted a survey on 12 experts. Out of these 12 experts, 6 are academicians who have either good knowledge of AO technology or are doing their research in this area. Remaining 6 professionals are working in software industry and are experts of AO technology. Survey form was provided to all the experts. The survey form consist seven tables for filling pair-wise relative weight values of characteristics and sub-characteristics. First table is to fill pair-wise relative weight values of six characteristics C1 to C6. The mean of collected samples of pair-wise relative weights are given in square matrix  $\mathbf{A} = [a_{ij}]$  given in equation (6.5), which is prepared using equation (6.1) to apply AHP process. In the same survey form, second table is for filling the pair-wise relative weights of sub-characteristics SC11 to SC16 of characteristic C1. Similarly in the survey form remaining tables are for filling the pair-wise relative weights of sub-characteristics of C2, C3, C4, C5 and C6.

Next step is to determine Eigen vector and Eigen values to get corresponding weights of C1, C2, C3, C4, C5, and C6 and consistency ratio (CR).

$$\mathbf{A} = [a_{ij}] = \begin{matrix} & \begin{matrix} C_1 & C_2 & C_3 & C_4 & C_5 & C_6 \end{matrix} \\ \begin{matrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{matrix} & \begin{bmatrix} 1 & 0.367 & 0.575 & 0.458 & 0.775 & 3.167 \\ 2.725 & 1 & 0.633 & 0.758 & 0.825 & 2.667 \\ 1.739 & 1.579 & 1 & 1.483 & 0.600 & 2.958 \\ 2.183 & 1.319 & 0.674 & 1 & 0.883 & 3.158 \\ 1.290 & 1.212 & 1.667 & 1.132 & 1 & 3.508 \\ 0.316 & 0.378 & 0.338 & 0.317 & 0.285 & 1 \end{bmatrix} \end{matrix} \quad (6.5)$$

#### 6.4.2 Determining Eigen Vector and Eigen Values

There are many methods for finding the Eigen vector. Multiplying all the entries in each row of the matrix **A** and then taking the  $n^{\text{th}}$  root (in our case  $6^{\text{th}}$  root) of the product helps in getting very good approximation to the correct answer. The values are given in Table 6.3. The  $n^{\text{th}}$  root is summed and this sum is used to normalize the Eigen vector elements to add to 1.000. In the matrix of Table 6.3, the  $6^{\text{th}}$  root for the first row is 0.786 and this is divided by 6.561 to give 0.119 as the first element in the Eigen vector. Thus Eigen vector of the relative importance of C1, C2, C3, C4, C5 and C6 are 0.119, 0.182, 0.212, 0.202, and 0.060, respectively, which are given in Table 6.3. These values are weights of main characteristics i.e. functionality (0.119), reliability (0.182), usability (0.212), efficiency (0.202), maintainability (0.225) and portability (0.060).

As given in equation (6.2),  $\mathbf{A} \cdot \omega = \lambda_{\max} \cdot \omega$ , and  $\lambda_{\max} \geq 6$ , next step is to get product  $\mathbf{A} \cdot \omega$ . Eigen values  $\lambda_{\max}$  can be evaluated by applying  $\lambda_{\max} = (\mathbf{A} \cdot \omega / \omega)$  and these six  $\lambda_{\max}$  values are calculated as 6.428, 6.258, 6.217, 6.103, 6.186 and 6.103. All these values of  $\lambda_{\max}$  are greater than 6, which satisfy the condition of  $\lambda_{\max} \geq n$ . The mean of  $\lambda_{\max}$  values is 6.215.

Now, one can evaluate *CI* and *CR* using equation (6.3) and (6.4).

$$CI = \frac{\lambda_{\max} - n}{n - 1} = \frac{6.215 - 6}{6 - 1} = 0.043$$

The final step is to calculate the consistency ratio (*CR*) for the set of judgments using the *CI* for the corresponding value from samples of matrices of purely random judgments. The calculated value of *CR* comes out less than 0.1 in all the samples of matrices, which indicates that the estimate is acceptable.

|       | C1    | C2    | C3    | C4    | C5    | C6    | $n^{\text{th}} (6^{\text{th}})$ root<br>of product of | Eigen vector<br>( $\omega$ ) | $A.\omega$ | $\lambda_{\text{max}}$<br>= $(A.\omega/\omega)$ |
|-------|-------|-------|-------|-------|-------|-------|-------------------------------------------------------|------------------------------|------------|-------------------------------------------------|
| C1    | 1     | 0.367 | 0.575 | 0.458 | 0.775 | 3.167 | 0.786                                                 | 0.119                        | 0.765      | 6.428                                           |
| C2    | 2.725 | 1     | 0.633 | 0.758 | 0.825 | 2.667 | 1.192                                                 | 0.182                        | 1.139      | 6.258                                           |
| C3    | 1.739 | 1.579 | 1     | 1.483 | 0.600 | 2.958 | 1.390                                                 | 0.212                        | 1.318      | 6.217                                           |
| C4    | 2.183 | 1.319 | 0.674 | 1     | 0.883 | 3.158 | 1.325                                                 | 0.202                        | 1.233      | 6.103                                           |
| C5    | 1.290 | 1.212 | 1.667 | 1.132 | 1     | 3.508 | 1.476                                                 | 0.225                        | 1.392      | 6.186                                           |
| C6    | 0.316 | 0.378 | 0.338 | 0.317 | 0.285 | 1     | 0.392                                                 | 0.060                        | 0.366      | 6.103                                           |
| Total |       |       |       |       |       |       | 6.561                                                 | 1.000                        |            | Mean<br>=6.215                                  |

**Table 6.3 Eigen Vector and Eigen Value for Main Characteristics**

Similarly, we have applied AHP process on pair-wise relative weights of sub-characteristics of characteristics C1 to C6 one by one and Eigen vectors of set of sub-characteristics are listed in fourth column of Table 6.4. For example, in this column Eigen vector (0.122, 0.061, 0.201, 0.109, 0.305, 0.202) is for sub-characteristics of C1, (0.092, 0.356, 0.552) is for sub-characteristics of C2 and so on. In order to normalize Eigen vectors' values of set of sub-characteristics to add to 1.000, multiply these values by weights of corresponding characteristics. For example, weight of SC11 (out of 1.000) =  $0.122 * 0.119 = 0.015$ , weight of SC22 =  $0.356 * 0.182 = 0.065$  and so on. These calculated values are listed in column 5 of Table 6.4.

| Characteristics | Weights for Characteristics | Sub-characteristics | Eigen vectors for sub-characteristics corresponding to their characteristics | Weights for sub-characteristics | Compared Quality of |          |
|-----------------|-----------------------------|---------------------|------------------------------------------------------------------------------|---------------------------------|---------------------|----------|
|                 |                             |                     |                                                                              |                                 | Project1            | Project2 |
| C1              | 0.119                       | SC11                | 0.122                                                                        | 0.015                           | 0.321               | 0.679    |
|                 |                             | SC12                | 0.061                                                                        | 0.007                           | 0.612               | 0.388    |
|                 |                             | SC13                | 0.201                                                                        | 0.024                           | 0.500               | 0.500    |
|                 |                             | SC14                | 0.109                                                                        | 0.013                           | 0.532               | 0.468    |
|                 |                             | SC15                | 0.305                                                                        | 0.036                           | 0.431               | 0.569    |
|                 |                             | <b>SC16</b>         | 0.202                                                                        | 0.024                           | 0.764               | 0.236    |
| C2              | 0.182                       | SC21                | 0.092                                                                        | 0.017                           | 0.500               | 0.500    |
|                 |                             | SC22                | 0.356                                                                        | 0.065                           | 0.532               | 0.468    |
|                 |                             | SC23                | 0.552                                                                        | 0.100                           | 0.764               | 0.236    |
| C3              | 0.212                       | SC31                | 0.401                                                                        | 0.085                           | 0.766               | 0.234    |
|                 |                             | SC32                | 0.260                                                                        | 0.055                           | 0.454               | 0.546    |
|                 |                             | SC33                | 0.136                                                                        | 0.029                           | 0.500               | 0.500    |
|                 |                             | <b>SC34</b>         | 0.203                                                                        | 0.043                           | 0.345               | 0.655    |
| C4              | 0.202                       | SC41                | 0.252                                                                        | 0.051                           | 0.540               | 0.460    |
|                 |                             | SC42                | 0.435                                                                        | 0.088                           | 0.618               | 0.382    |
|                 |                             | <b>SC43</b>         | 0.313                                                                        | 0.063                           | 0.451               | 0.549    |

|       |       |             |       |       |       |       |
|-------|-------|-------------|-------|-------|-------|-------|
| C5    | 0.225 | SC51        | 0.292 | 0.066 | 0.654 | 0.346 |
|       |       | SC52        | 0.153 | 0.034 | 0.765 | 0.235 |
|       |       | SC53        | 0.219 | 0.049 | 0.340 | 0.660 |
|       |       | SC54        | 0.123 | 0.028 | 0.567 | 0.433 |
|       |       | <b>SC55</b> | 0.213 | 0.048 | 0.762 | 0.238 |
| C6    | 0.060 | SC61        | 0.163 | 0.010 | 0.345 | 0.655 |
|       |       | SC62        | 0.312 | 0.018 | 0.651 | 0.349 |
|       |       | SC63        | 0.309 | 0.019 | 0.234 | 0.766 |
|       |       | SC64        | 0.216 | 0.013 | 0.765 | 0.235 |
| Total | 1.000 |             |       | 1.000 |       |       |

**Table 6.4 Eigen Vectors and Weights for Sub-Characteristics**

Twenty five evaluated sub-characteristics are weighted as follows: Suitability (0.015), Accuracy (0.100), Interoperability (0.024), Compliance (0.013), Security (0.036), **Reusability** (0.024), Maturity (0.017), Fault-tolerance (0.065), Recoverability (0.100), Understandability (0.085), Learn-ability (0.055), Operability (0.029), **Complexity** (0.043), Time behavior (0.051), Resource behavior (0.088), **Code-reducibility** (0.063), Analyzability (0.066), Changeability (0.034), Stability (0.049), Testability (0.028), **Modularity** (0.048), Adaptability (0.010), Install-ability (0.018), Replace-ability (0.019), and Conformance (0.013).

We also calculated CRs for samples collected for sub-characteristics and found in all the cases CR value < 0.1, which indicates that estimate is acceptable.

### 6.4.3 Evaluating Quality of AO Projects

We have taken two projects (Laddad, 2003) developed in AspectJ and compared those on the scale of 1.000 for all the 25 sub-characteristics of AOSQUAMO. This comparison has been done by four students of M.Tech.(CSE) who were doing their dissertations in the field of AOP. These compared values of sub-characteristics may not be accurate, but our objective here is just to describe the methodologies. The mean of these compared values are given in columns 6 and 7 labeled project1 and project2 of Table 6.4. Quality of any project can be evaluated by equation (6.6).

$$\text{AO Project Quality} = \sum_{i=1}^n \text{Comparative Value of } SC_i * \text{Weight Value of } SC_i \quad (6.6)$$

where  $n$  is the number of sub-characteristics,  $SC_i$  is sub-characteristic  $i$ . The calculated value of quality of project1 is 0.579 and quality of project2 is 0.421. As we know that quality is an empirical value, hence project1 is better in terms of quality.

In actual situation, comparative values of characteristics and sub-characteristics can be gathered from running projects, which are developed using AO technology. The projects, which are compared here, are not real-life projects. But to show the methodology of comparison and evaluation of quality as a single unit, we have taken above said projects, for which comparative values may not be realistic.

## 6.5 CONCLUSION

In this chapter, a new aspect-oriented software quality model AOSQUAMO is proposed. This model is an extension of ISO/IEC 9126 software quality standards. Four new sub-characteristics, *modularity*, *code-reducibility*, *complexity*, and *reusability* have been added under characteristics maintainability, efficiency, usability, and functionality, respectively. These new sub-characteristics have been added on the basis of new features of aspect-oriented technology over object-oriented technology. Existing characteristics and sub-characteristics of ISO/IEC 9126, which are also part of AOSQUAMO, have been defined in terms of aspect-oriented approach. In order to evaluate quality of proposed model as a single unit, Analytic Hierarchy Process (AHP) is applied. Pair-wise relative

weights of characteristics and sub-characteristics have been taken through a survey on 12 experts of the field and mean of the collected samples have been considered as pair-wise relative weights. On these mean values, AHP is applied to get corresponding weights of characteristics and sub-characteristics in which total quality weight has been considered as 1.000. Quality of two AO projects has been compared using these weights. AOSQUAMO can be used to compare and select the best suitable AO technology based projects.



# CHAPTER 7

## CONCLUDING OBSERVATIONS AND FUTURE SCOPE

---

### 7.1 INTRODUCTION

The primary objective of this study was to design metrics to predict various quality characteristics such as maintainability, reusability, complexity and overall quality of aspect-oriented systems. The research findings transpired from this study are briefly recapitulated point wise and certain pointers to future research in this direction have been demarcated in this chapter.

### 7.2 REVIEW OF THE RESULTS OBTAINED FROM THIS STUDY

In Chapter 2, we have defined new AOS terminology and formalism, in order to represent members of different AOP languages as common and standard names. We have considered Java and three AOP languages AspectJ, CaesarJ and Hyper/J as part of our generic framework. Scope of our generic framework is limited to those AOP languages, which have AO features as well as most of the features of Java programming language. That is why; we have short listed above three AOP languages. Using these terminologies and selection criteria given in the framework, we have proposed metrics for internal characteristics which are cohesion, coupling, complexity and size for AOS. These proposed metrics can be used to determine external characteristics such as maintainability, understandability, reusability, usability, adaptability and modularity.

We have proposed a fuzzy logic model to assess code complexity of generic AOS, which is presented in Chapter 3. Input variables in this fuzzy model are metrics for complexity, defined in Chapter 2. These inputs are complexity of attributes, complexity of operations, and complexity of nested components. Attributes and operations have been divided into three categories, low (L), medium (M) and high (H) as member functions. NIL member function for attribute represents that there is no attribute in the component. NIL member function for operation represents that there is no operation in the component. Similarly, nested component as input variable and component as output variable have been divided into five categories: very low (VL), L, M, H, and very high (VH), as member functions. NIL member function for nested component represents no nested component present in the component. NIL member function for component means complexity of component as output variable is zero. This fuzzy model automates assessment of complexity of generic AOS.

In aspect-oriented technology, changes in crosscutting concerns become easy. In order to predict maintainability of AOS, we have proposed changeability metric in Chapter 4. We determined different type of possible changes at code level in AOS and performed experiment on projects developed in AspectJ and CaesarJ. Impacts of these changes have been evaluated. High change impact of an AO component means, it will absorb low changes or in other words, low changeability. This changeability metric has been validated comparing with an existing technique, which is based on dependency analysis. Correlation factor value, obtained from correlation analysis between our proposed changeability metric and existing technique, is high enough to validate our metric. We also performed correlation analysis of changeability metric and cohesion metric to check whether cohesion metric can be used or not to predict changeability. Our experimental result analysis indicates that cohesion metric cannot be used as indicator for predicting changeability/ maintainability.

Reusability is an important feature of any software technology. We have proposed a model to assess reusability of AO components in AOS. This model presents relationship of reusability with external and internal characteristics of AOS. In order to find out

external characteristics on which reusability depends, we conducted survey on experts of the domain. We observed from the survey that *understandability*, *modularity*, *maintainability*, and *adaptability* are main external characteristics, which determine reusability of AOS. As external characteristics cannot be evaluated directly, some of the direct measureable internal characteristics can be used to determine these indirect measurable external characteristics. A correlation analysis among these indirectly measurable four external characteristics and directly measureable metrics for internal characteristics cohesion, coupling, complexity, and size, has been performed. From the analysis, it has been observed that (i) metrics  $CMPX_{M(c)}$ , NOO, LOC, and CoOI (when component is an aspect), are directly proportional to the reusability of AO component, (ii) metric CoI is inversely proportional to the reusability of AO component, and (iii) there is no strong relation among external characteristics and CoAT, CoPT, CoAR, NOA, and CoOI (when component is a class) metrics. This work is presented in Chapter 5.

In literature, there are many software quality models, but most of them are lacking in explaining new features, which has been included in AO technology. Chapter 6 presents aspect-oriented software quality model by adding four new sub-characteristics namely reusability, complexity, code-reducibility and modularity to the existing ISO/IEC 9126 quality model. These new characteristics are important in context of AOS. In order to get overall quality of AOS, AHP has been applied, wherein weight values to characteristics and sub-characteristics have been assigned through a survey on software professionals working in the field of AO technology. These empirical weight values will help developers to select only those characteristics which are as per their requirement in that domain.

### **7.3 LIMITATIONS AND FUTURE SCOPE OF THE STUDY**

Scope of the study is limited to those AOP languages which are either extended from Java or have most of the features of Java programming language. Shortlisted programming languages of this category are: AspectJ, CaesarJ and Hyper/J. In future work, we may also consider other AOP languages such as AspectC, AspectC++, AspectL,

and AspectXML and features of these AOP languages can also be included in our generic framework.

In the present investigations, experimental case studies are of small/medium size projects. It was very difficult to get large industrial projects of this domain for experiment. However, results obtained from the present study are quite instructive and industry persons will find them quite useful and suitable for implementation in real-life projects. In order to test our proposed metric measures on real-life industrial projects of the domain, we are continuously in search of such projects so that results can be refined.

In the present study, values of different metrics have been evaluated manually on different projects. Manual process is tedious, difficult, time consuming and possibility of error may be there. Especially in large industrial projects, manual evaluation of metric values is not feasible. So, there is a need to develop a tool to get different values of metrics. We are in the process of developing such tool in Java language, which will help software quality team for quality assessment of the domain.



## REFERENCES

---

1. Aggarwal, K. K., Singh, Y., Chandra, P., Puri, M., 2005a. Measurement of Software Maintainability Using a Fuzzy Model, *Journal of Computer Sciences*, 1(4), pp: 538-542.
2. Aggarwal, K.K., Singh, Y., Chandra, P., Puri, M., 2005b. Sensitivity Analysis of Fuzzy and Neural Network Models, *ACM SIGSOFT Software Engineering Notes*, 30(4), pp: 1-4.
3. Aggarwal, K. K., Singh, Y., Kaur, A., Malhotra, R., 2005c. Software Reuse Metrics for Object-Oriented Systems, *IEEE Proceedings of the Third ACIS International Conference on Software Engineering Research, Management and Applications*, pp: 48–55.
4. Aggarwal, K. K., Singh, Y., Kaur, A., Malhotra, R., 2006. Application of Artificial Neural Network for Predicting Maintainability using Object-Oriented Metrics, *Transactions on Engineering, Computing and Technology*, Vol. 15, pp: 285-289.

5. Aggarwal, K. K., Singh, Y., Kaur, A. and Malhotra, R., 2007. Software Design Metrics for Object-Oriented Software. *Journal of Object Technology*. 6(1), pp: 121-138.
6. Aldrich, J., 2004. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. Carnegie Mellon Technical Report CMU-ISRI-04-108, 2004 (Earlier version appeared in Workshop on Foundations of Aspect-Oriented Languages).
7. Alexander, R.T., 2003. The Real Costs of Aspect-Oriented Programming. *IEEE Software*. 20(6), pp: 91–93.
8. Aljasser, K., and Schachte, P., 2009. ParaAJ: toward Reusable and Maintainable Aspect Oriented Programs. In *Proceedings of Thirty-Second Australasian Computer Science Conference (ACSC 2009)*, Wellington, New Zealand. CRPIT, 91. Mans, B., Ed. ACS. pp: 53-62.
9. Anna, C. S., Figueiredo, E., Garcia, A., and Lucena, C. J. P., 2007. On the Modularity Assessment of Software Architectures: Do my architectural concerns count?. 6th International Workshop on Aspect-Oriented Software Development (AOSD'2007), Vancouver, British Columbia, March 12-13.
10. Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K., 2006. Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, LNCS, 3880, pp:135-173.
11. Arisholm, E., Briand, L.C., and Føyen, A., 2004. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering*, 30(8), pp. 491-506.
12. Bandi, R. K., Vaishnavi, V. K., Turk, D. E., 2003. Predicting Maintenance Performance using Object-Oriented Design Complexity Metrics. *IEEE Transactions on Software Engineering*, 29(1), pp: 77–87.

13. Banker, R. D., Kauffman, R. J., Wright, C. R. and Zweig, D., 1991. Automating Output Size and Reusability Metrics in an Object-Based Computer Aided Software Engineering (Case) Environment, Information Systems Working Papers Series, available at: <http://www.ssrn.com/abstract=1289052>.
14. Bansiya, J., Davis, C., and Etzkorn, L., 1999. An Entropy-Based Complexity Measure for Object-Oriented Designs. *Theory and Practice of Object Systems*. 5(2), pp:11-118.
15. Bansiya, J., Davis, C., 2002. A Hierarchical Model for Object-Oriented Design Quality Assessment, *IEEE Transactions on Software Engineering*, Vol. 28, Issue 1, pp: 4-17.
16. Barnard, J., 1998. A New Reusability Metric for Object-Oriented Software, *Software Quality Journal*, Vol. 7, Issue 1, pp: 35–50.
17. Bartolomei, T. T., Garcia, A., Sant’Anna, C., and Figueiredo, E., 2006. Towards a Unified Coupling Framework for measuring Aspect-Oriented Programs. In 3<sup>rd</sup> International Workshop on Software Quality Assurance (SOQUA 2006) Portland, Oregon, USA, November 6, ACM Press.
18. Bartsch, M., and Harrison, R., 2006. An Evaluation of Coupling Measures for AspectJ. Workshop LATE’06, in conjunction with Aspect-Oriented Software Development Conference (AOSD’06), Bonn, Germany, March 20, ACM Press.
19. Basili, V.R., 1980. Qualitative Software Complexity Models: A Summary. In *Tutorial on Models and Methods for Software Management and Engineering*, IEEE Computer Society Press, Los Alamitos, California.
20. Bergmans, L., and Aksits, M., 2001. Composing crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol.44, No.10, October 2001, pp: 51-57.
21. Bertoa, M., and Vallecillo, A., 2002. Quality Attributes for COTS Components. In the Proceedings of 6th International ECOOP Workshop on Quantitative

- Approaches in Object-Oriented Software Engineering (QAOOSE 2002), Spain, 1(2), pp: 128-144.
22. Boegh, J., Depanfilis, S., Kitchenham, B., and Pasquini, A., 1999. A Method for Software Quality Planning, Control, and Evaluation. *IEEE Software*. 16(2), pp: 69-77.
  23. Boehm, B. W., Brown, J. R., and Lipow, M. L., 1976. Quantitative Evaluation of Software Quality. In *IEEE Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, California, United States, pp: 592-605.
  24. Booch, G., 1994. *Object-Oriented Analysis and Design, with Applications* (2<sup>nd</sup> ed.), Benjamin/ Cummings, San Mateo, CA.
  25. Boxall, M. A. S., Araban, S., 2004. Interface Metrics for Reusability Analysis of Components, *Proceedings of. Australian Software Engineering Conference (ASWEC'2004)*, Melbourne, Australia, pp: 40-46.
  26. Briand, L., Devanbu, P., and Melo, W., 1997. An Investigation into Coupling Measures for C++. In *Proceedings of International Conference on Software Engineering (ICSE '97)*, pp: 412-421.
  27. Briand, L. C., Daly, J. W., Wust, J., 1998. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering : an International Journal*, 3(1), pp: 65-117.
  28. Briand, L. C., Daly J. W., and Wust, J., 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*. 25(1), pp. 91-120.
  29. CaesarJ, 2004. <http://caesarj.org/index.php/Caesar/Tutorial>.
  30. Ceccato, M., and Tonella, P., 2004. Measuring the Effects of Software Aspectization. In: *Proceedings of the 1<sup>st</sup> Workshop on Aspect Reverse Engineering*. ACM Press.
  31. Chang, C., Wu, C., and Lin, H., 2008. Integrating Fuzzy Theory and Hierarchy

- Concepts to Evaluate Software Quality. *Software Quality Control*. 16(2), pp:263-276.
32. Chaumon, M.A., Kabaili, H., Keller, R.K., and Lustman, F., 2002. A Change Impact Model for Changeability Assessment in Object-Oriented Systems. *Science of Computer Programming*, vol.45(2-3), Elsevier Science Publishers, pp- 155-174.
  33. Chidamber, S. R., and Kemerer, C. F., 1994. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6), pp. 476–493.
  34. Chulani, S., Boehm, B., and Steece, B., 1999. Bayesian Analysis of Empirical Software Engineering Cost Models. *IEEE Transactions on Software Engineering*, 25( 4), pp: 573-583.
  35. Cote, M., Suryn, W. Georgiadou, E., 2006. Software Quality Model Requirements for Software Quality Engineering, *Proceedings of Software Quality Management and INSPIRE Conference*, pp: 31-50.
  36. Cunha, C. A., Sobral, J. L., and Monteiro, M. P. 2006. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *Proceedings of the 5th international Conference on Aspect-Oriented Software Development (Bonn, Germany, March 20 - 24, 2006)*. AOSD '06. ACM, New York, NY, pp: 134-145.
  37. Dandashi, F., 2002. A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements. In *Proceedings of the 2002 ACM Symposium on Applied Computing, Madrid, Spain*, pp: 997-1003.
  38. Davis, J.S., and LeBlanc, R.J., 1988. A Study of the Applicability of Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9), pp: 1366-1371.
  39. Devanbu, P. Karstu, S., Melo, W., Thomas, W., 1996. Analytical and Empirical Evaluation of Software Reuse Metrics, *Proceedings of 18th International*

- Conference on Software Engineering, Berlin, Germany, pp: 189–199.
40. Deutsch, M. S., and Wills, R. R., 1998. Software Quality Engineering; A Total Technical and Management Approach. Prentic-Hall, Inc.
  41. Dospisil, J., 2003. Measuring Code Complexity in Projects Designed with AspectJ. Informing Science InSITE-“Where Parallels Intersects”, pp: 185-197.
  42. Dromey, R. G., 1995. A Model for Software Product Quality. IEEE Transactions on Software Engineering, 21(2), pp: 146-162.
  43. Dumke, R., Schmietendorf, A., 2000. Possibilities of the Description and Evaluation of Software Components, Metrics News, Vol. 5, Issue 1, pp: 13-26.
  44. Eder, J., Kappel, G., and Schrefl, M., 1994. Coupling and Cohesion in Object-Oriented Systems. Technical Report, University of Klagenfurt, 1994.
  45. Elrad, T., Filman, R. E., Bader, A., 2001a. Aspect-Oriented Programming: Introduction. Communication of the ACM, 44(10), pp: 29-32.
  46. Elrad, T., Aksits, M., Kiczales, G., Lieberherr, K., and Ossher, H., 2001b. Discussing aspects of AOP. Communications of the ACM. 44(10), pp: 33–38.
  47. Figueiredo, E., Sant'Anna, C., and Garcia, A., 2008. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In IEEE Proceedings of the 12<sup>th</sup> European Conference on Software Maintenance and Reengineering (CSMR'08), pp: 183-192.
  48. Fioravanti, F., Nesi, P., 2001. Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems, IEEE Transactions on Software Engineering, 27(12), pp: 1062–1084.
  49. Frakes, W., Terry, C., 1996. Software Reuse: Metrics and Models, ACM Computing Surveys, 28(2), pp: 415–435.
  50. Garcia, V. C., Piveta, E. K., Lucrédio, D., Álvaro, A., Almeida, E. S., Zancanella, L.C., and Prado, A.F., 2004. Manipulating Crosscutting Concerns.

In Proceedings of 4<sup>th</sup> Latin American Conferences on Patterns Languages of Programming (SugarLoafPLOP), Porto das Dunas, CE, Brazil.

51. Gelinas, J.F., Badri, M., and Badri, L., 2006. A Cohesion Measure for Aspects. *Journal of Object Technology*, 5(7), pp. 97–114.
52. Gill, N. S., 2003. Reusability Issues in Component-based Development, *ACM SIGSOFT Software Engineering Notes*, Vol. 28, Issue 4, pp: 1-5.
53. Gill, N.S., 2005. Factors Affecting Effective Software Quality Management Revisited. *ACM SIGSOFT Software Engineering Notes*. 30(2), pp:1-4.
54. Gradecki, J. D., and Lesiecki, N., 2003. Mastering AspectJ: Aspect-Oriented Programming in Java. Wiley, 2003.
55. Grady, R. B., 1992. Practical Software Metrics For Project Management And Process Improvement, Prentice Hall.
56. Grover, P.S., Kumar, R., and Kumar, A., 2008. Measuring Changeability for Generic Aspect-Oriented Systems. *ACM SIGSOFT Software Engineering Notes*. 33(6), pp:1-5.
57. Grover, P. S., Kumar, R., Sharma, A., 2007. Few Useful Considerations for Maintaining Software Components and Component-Based Systems. *ACM SIGSOFT Software Engineering Notes*, Vol. 32, Issue 4, pp: 1-5.
58. Harrison, W., 1992. An Entropy-Based Measure of Software Complexity. *IEEE Transactions of Software Engineering*, 18(11), November, 1992, pp: 1025-1029.
59. Henderson-Sellers, B., 1996. Software Metrics. Prentice Hall, Hemel Hempstead, U.K.
60. Hitz, M. and Montazeri, B., 1995. Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proceedings of International Symposium Applied Corporate Computing*, Monterrey, Mexico, Oct. 1995, pp: 78-84.

61. Hsia, P., Gupta, A., Kung, C., Peng, J., and Liu, S., 1995. A Study of the Effect of Architecture on Maintainability of Object-Oriented Systems. In IEEE Proceedings of International Conference of Software Maintenance (ICSM95), Nice, France, Oct 17-20, pp- 4-11.
62. Hung, D., Gao, J., Hsia, P., Wen, F., Yoyoshima, Y., and Chen, C., 1994. Change Impact Identification in Object-Oriented Software Maintenance. IEEE Proceedings of International Conference of Software Maintenance (ICSM94), pp: 202-211.
63. Hussain, O., Dillon ,T., Chang, E. and Hussain, F., 2009. A Fuzzy Inference Model for Risk based Informed Decision-Making in e-Business. In IEEE Proceedings of 2009 International Conference on Network-Based Information Systems (NBIS, 2009), pp:519-524.
64. IEEE Standard Glossary of Software Engineering Terminology, 1990. IEEE Std 610.12-1990, The Institute of Electrical and Electronics Engineers, Inc.
65. IEEE Standard for Software Maintenance, 1998. IEEE Std 1219-1998. The Institute of Electrical and Electronics Engineers, Inc.
66. Ince, D., 1994. ISO 9001 and Software Quality Assurance, McGraw-Hill, New York.
67. ISO 8402 Quality Management and Quality Assurance - Vocabulary, 1994. International Organization for Standardization, Geneva, 2nd Edition.
68. ISO/IEC 9126, (2001). Institute of Electrical and Electronics Engineering, Part 1, 2, 3, 4: Quality Model.
69. Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., 1992. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley Publishing Company, 1992.
70. Jacobson, I., Booch, G., Rumbaugh, J., 2003. The Unified Software Development Process, Addison Wesley Inc., USA.
71. JBoss, J. Inc. JBoss AOP Beta3, 2004.

<http://www.jboss.org>.

72. Johnson, R., 2003. Introducing the Spring Framework.  
<http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework>.
73. Jung, H. W., Kim, S. G., and Chung, C. S., 2004. Measuring Software Product Quality: A Survey of ISO/IEC 9126. *IEEE Software*. 21(5), pp: 88-92.
74. Kabaili, H., Keller, R. K., and Lustman, F., 2001a. Cohesion as Changeability Indicator in Object-Oriented Systems. In *IEEE Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pp: 39-46, Estoril Coast (Lisbon), Portugal, March 2001.
75. Kabaili, H., Keller, R. K., and Lustman, F., 2001b. Class Cohesion as Predictor of Changeability: An Empirical Study. *International Journal, L'Objet*, 7(4), pp: 515-534. Hermes Science Publications, Paris, France.
76. Kafura, D., Henry, S., 1981. Software Quality Metrics, Based on Interconnectivity, *Journal of Systems and Software*. Vol. 2, pp: 121-131.
77. Kamiya, T., Kusumoto, S., Inoue, K., Mohri, Y., 1999. Empirical Evaluation of Reuse Sensitiveness of Complexity Metrics, *Information and Software Technology*, Vol. 41, pp: 297-305.
78. Karunanithi, S., Bieman, J. M., 1993. Candidate Reuse Metrics for Object Oriented and Ada Software, *Proceedings of IEEE International Software Metrics Symposium*, available at <http://www.cs.colostate.edu/~bieman/Pubs/santhiBiemanMetrics93.pdf>
79. Kassab, M., Ormandjieva, O., Constantinides, C., 2005. Providing Quality Measurement for Aspect-Oriented Software Development. In *IEEE Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, 15-17 December 2005, pp.769-775.
80. Khairuddin, H., Elizabeth, K., 1996. A Software Maintainability Attributes Model, *Malaysian Journal of Computer Science*, Vol. 9, Issue 2, pp: 92-97.

81. Khoshgoftaar, T. M., Seliya, N., and Sundaresh, N., 2006. An Empirical Study of Predicting Software Faults with Case-Based Reasoning. *Software Quality Journal*. 14(2), pp: 85–111.
82. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., and Irwin, J., 1997. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming, LNCS, Vol.1241*, Springer-Verlag, June 1997, pp:220-242.
83. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G., 2001a. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 2072*. Springer-Verlag, London, June 18-22, pp: 327-353.
84. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W., 2001b. Getting Started with AspectJ. *Communication of the ACM*, vol. 44, no.10, October 2001, pp: 59-65.
85. Kitchenham, B. and Kari, K. 1993. Inter item Correlations among Function Points, *Proceedings of the IEEE Computer Society International Software Metrics Symposium*, Baltimore, MD, pp: 11-15.
86. Kumar, A., Kumar, R., and Grover, P.S., 2006a. A Comparative Study of Aspect-Oriented Methodology with Module-Oriented and Object-Oriented Methodologies. *ICFAI Journal of Information Technology*. 2(4), pp: 7-15.
87. Kumar, A., Kumar, R., Grover, P.S., 2006b. A Change Impact Assessment in Aspect-Oriented Software Systems. In the proceedings of *International Software Engineering Conference Russia 2006 (SECR-2006)*, Dec, 2006, pp: 83-87.
88. Kumar, A., Rajesh, K., and Grover, P.S., 2007. An Evaluation of Maintainability of Aspect-Oriented Systems: a Practical Approach. *International Journal of Computer Science and Security*. 1(2), pp: 1-9.

89. Kumar, A., Kumar, R., and Grover, P.S., 2008a. Towards a Unified Framework for Cohesion Measurement in Aspect-Oriented Systems. In IEEE Proceedings of 19th Australian Software Engineering Conference (ASWEC 2008) Perth, Western Australia, March 26-28, pp:57-65.
90. Kumar, A., Kumar, R., and Grover, P.S., 2008b. Towards a Unified Framework for Complexity Measurement in Aspect-Oriented Systems. In IEEE Proceedings of 2008 International Conference on Computer Science & Software Engineering (CSSE 2008), Wuhan, China, Dec-12-14, pp: 98-103.
91. Kumar, A., Kumar, R., and Grover, P.S., 2009a. Generalized Coupling Measure for Aspect-Oriented Systems. ACM SIGSOFT Software Engineering Notes. 34(3), pp: 1-6.
92. Kumar, A., Grover, P.S., and Kumar, R., 2009b. A Quantitative Evaluation of Aspect-Oriented Software Quality Model (AOSQUAMO). ACM SIGSOFT Software Engineering Notes. 34(5), pp: 1-9.
93. Kumar, R., Grover, P.S., and Kumar, A., 2010a. A Fuzzy Logic Approach to Measure Complexity of Generic Aspect-Oriented Systems. Journal of Object Technology (JOT), 9(3), pp: 43-57, May/June 2010.
94. Kumar, A., Kumar, R., and Grover, P.S., 2010b. Unified Cohesion Measures for Aspect-Oriented Systems. Accepted for publication in International Journal of Software Engineering and Knowledge Engineering (IJSEKE), September Issue, 2010.
95. Laddad, R., 2003. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications.
96. Land, R., 2002. Measurements of Software Maintainability, Proceedings of ARTES Graduate Student Conference, ARTES, pp: 1-7.
97. Li, L., and Offutt, A. J., 1996. Algorithmic Analysis of the Impact of Changes to Object-Oriented Software. In IEEE Proceedings of International Conference

- of Software Maintenance (ICSM96), pp: 171-184.
98. Li, B., 1998. Software Reuse, available at <http://sern.ucalgary.ca/courses/seng/693/W98/lib/reuse.htm>
  99. Li, H., 1993. Object-Oriented Metrics that Predict Maintainability, *Journal of Systems and Software*, Vol. 23 Issue 2, pp: 111-122.
  100. Li, J., Kvale, A.A., and Conradi, R., 2006. A Case Study on Improving Changeability of COTS-Based System Using Aspect-Oriented Programming. *Journal of Information Science and Engineering*, 22(2), pp- 155-174.
  101. Lieberher, K., Orleans, D., and Ovlinger, J., 2001. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*. 44(10), pp: 39-41.
  102. Lisa, C., Delugach, H. S., 2001. Dependency Analysis Using Conceptual Graphs, In *Proceedings of the 9th International Conference on Conceptual Structures*, ICCS92001, pp: 117-130.
  103. Losavio, F., Chirinos, L., Perez, M. A., 2002. Quality Models to Design Software Architecture, *Journal of Object Technology*, Vol. 1, Issue 4, pp: 165-178.
  104. Loyall, J. P., and Mathisen, S. A., 1993. Using Dependence Analysis to Support the Software Maintenance. Process," *IEEE Proceedings of International Conference of Software Maintenance (ICSM93)*. pp: 282-291.
  105. MacDonell, S. G., Gray, A. R., Calvert, J. M., 1999. FLSOME: Fuzzy Logic for Software Metric Practitioners and Researchers, In the *Proceedings of the 6<sup>th</sup> International Conference on Neural Information Processing ICONIP'99*, Perth, pp: 308-313.
  106. McCall, J. A., Richards, P. K., and Walters, G. F., 1977. Factors in Software Quality. Griffiths Air Force Base, N.Y. Rome Air Development Center Air Force Systems Command.

107. McCabe, T., 1976. A Software Complexity Measure, IEEE Transactions on Software Engineering, Vol. 2, Issue 4, pp: 308-320.
108. Mehner, K. (2006). On Using Metrics in the Evaluation of Aspect-Oriented Programs and Designs. In Linking Aspect Technology and Evolution Workshop at 5th Aspect-Oriented Software Development Conference (AOSD'06), Bonn, Germany.
109. <http://www.aosd.net/workshops/late/2006/late/submissions/mehner.pdf>.
110. Mili, H., Mili, F., Mili, A., 1995. Reusing Software: Issues and Research Directions, IEEE Transaction on Software Engineering, Vol. 21, Issue 6, pp: 528-561.
111. Musilek, P., Pedrycz, W., Succi, G., Reformat, M., 2000. Software Cost Estimation with Fuzzy Models, ACM SIGAPP Applied Computing Review, Vol. 8, pp: 24-29.
112. Nagib, C. B., Callaos, De, 1994. Designing with a Systematic Total Quality, Educational Technology, Vol. 34, pp: 29-36.
113. Ortega, M., Perez, M.A., Rojas, T., 2002. A Systemic Quality Model For Evaluating Software Products, Software Quality Control, Vol. 11, Issue 3, pp: 219 – 242.
114. Pataki, N., Sipos, A., and Porkolab, Z., 2006. Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric. ECOOP 2006 Doctoral Symposium and PhD Students Workshop.
115. <http://www.ecoop.org/phdoos/ecoop2006ds/ws/pataki.pdf>
116. Pekilis, B. R., 2002. Multi-Dimensional Separation of Concerns and IBM Hyper/J. Technical Research Report, January 22, 2002.
117. Pigoski, T. M., 1997. *Practical Software Maintenance*. John Wiley & Sons, New York, pages: 384.

118. Poulin, J., Caruso, J., Hancock, D., 1993. The Business Case for Software Reuse, IBM Systems Journal, Vol. 32, Issue 40, pp: 567-594.
119. Preiss, O., Weqmann, A., Wong, J., 2001. On Quality Attribute Based Software Engineering, Proceeding of 27th EuroMicro Conference, pp: 114-121.
120. Price, M. W., Demurjian, S. A. Sr., 1997. Analyzing and Measuring Reusability in Object-Oriented Design. In the Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. October 05-09, 1997, Atlanta, Georgia, US, pp: 22-33.
121. Prnjat, O., and Sacks, L., 2001. Complexity Measurements of the Inter-Domain Management System Design. In Proceedings of 9<sup>th</sup> IEEE International Conference on Networks (ICON'01), pp: 2.
122. Rawashdeh, A., and Matakah, B., 2006. A New Software Quality Model for Evaluating COTS Components. Journal of Computer Science. 2(4), pp: 373-381.
123. Reis, R.Q., Lima Reis, C.A., Schlebbe, H., and Nunes, D.J., 2002. Towards an Aspect-Oriented Approach to Improve the Reusability of Software Process Models. In Proceedings of the International Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Association for Computing Machinery, New York.
124. Rombach, H.D., 1990. Design Measurement: Some Lessons Learned. *IEEE Software*. 7(2), pp: 17-25.
125. Rotaru, O. P., Dobre, M., Petrescu, M., 2005. Reusability Metrics for Software Components. Proceedings of the 3rd ACS / IEEE International Conference on Computer Systems and Applications (AICCSA-05), Cairo, Egypt, pp: 24-29.
126. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., 1991. Object-Oriented Modeling and Design, Prentice-Hall, New York, 1991.

127. Ryder, B.G., and Tip, F., 2001. Change Impact Analysis for Object-Oriented Programs. In ACM proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Snowbird, Utah, United States, pp: 46-53.
128. Saaty, T.L., 1980. The Analytic Hierarchy Process. McGraw Hill International.
129. Salamon, W.J., and Wallace, D.R., 1994. Quality Characteristics and Metrics for Reusable Software (Preliminary Report). US DoC for US DoD Ballistic Missile Defense Organization, NISTIR 5459.
130. Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., and Staa, A., 2003. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. XXIII Brazilian Symposium on Software Engineering, Manaus, Brazil, October 2003.
131. Sant'Anna, C., Figueiredo, E., Garcia, A., Lucena, C. J. P., 2007. On the Modularity Assessment of Software Architectures: Do my architectural concerns count?. In Proceedings of the 6th International Workshop on Aspect-Oriented Software Development (AOSD'2007), Vancouver, British Columbia, March 12-13, 2007.
132. Selby, R.W., 1989. Quantitative studies of Software Reuse, Software Reusability, Vol. 2, applications and experience, Addison Wesley, pp: 213-233.
133. Sharma, A., Grover, P. S., Kumar, R., 2006. Investigation of Reusability, Complexity and Customizability Metrics for Component-Based Systems. ICFAI Journal of Information Technology, Vol. 1, Issue 2, pp: 6-11.
134. Sharma, A., Kumar, R., Grover, P. S., 2007. Complexity Measures for Software Components, WSEAS Transactions on Computers, Vol. 6, Issue 7, pp: 1005-1012.
135. Sharma, A., Kumar, R., Grover, P. S., 2008a. Empirical Evaluation of Complexity for Software Components, International Journal of Software

Engineering and Knowledge Engineering (IJSEKE), Vol. 18, Issue 5, pp: 519-530.

136. Sharma, A., Kumar, R., and Grover, P. S., 2008b. Estimation of Quality for Software Components: an Empirical Approach. ACM SIGSOFT Software Engineering Notes. 33(6), pp:1-10.
137. Shen, H., Zhang, S., and Zhao, J., 2008. An Empirical Study of Maintainability in Aspect-Oriented System Evolution Using Coupling Metrics. In the proceedings of 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, pp: 233-236.
138. Sicilia, M. Á., and García-Barriocana E., 2006. Extending Object Database Interfaces with Fuzziness Through Aspect-Oriented Design. ACM SIGMOD Record, 35(2), pp: 4–9.
139. Singh, P., and Singh, H., 2008. DynaMetrics: A Runtime Metric-Based Analysis Tool for Object-Oriented Software Systems. ACM SIGSOFT Software Engineering Notes. 33(6), pp: 1-6.
140. Sivanandam, S. N., Sumathi, S., Deepa, S. N., 2007. Introduction to fuzzy logic using MATLAB, Springer.
141. Stevens, W. P., Myers, G. J., and Constantine, L. L., 1974. Structured Design. IBM Systems Journal, 13(2), pp: 115–139.
142. Swebok, 2001. IEEE, Guide to the Software Engineering Body of Knowledge, 2001, [www.swebok.org](http://www.swebok.org).
143. Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M., 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proceedings of the International Conference on Software Engineering, 1999, pp: 107-119.
144. Tawfiq, S. M., Abd, M. M., Green, S., 2007. A Software Cost Estimation Model Based on Quality Characteristics, Workshop on Measuring Requirements for Project and Product Success, Palma de Mallorca, Spain, Available at: [www-swe.informatik.uni-heidelberg.de/home/events/MeRePDocs/paper2.pdf](http://www-swe.informatik.uni-heidelberg.de/home/events/MeRePDocs/paper2.pdf).

145. Tomer, A., Goldin, L., Kuflik, K., Kimchi, E., Stephen, R., Schach, 2004. Evaluating Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study, IEEE Transactions on Software Engineering, Vol. 30, Issue 9, pp: 601-612.
146. Tracz, W., 1990. The 3 Cons of Software Reuse, Proceedings of the Third Annual Workshop on Software Reuse, Syracuse, NY.
147. Voas, J., Agresti, W., 2004. Software Quality from a Behavioral Perspective, IEEE Computer Society, IT Professional, Vol. 6, Issue 4, pp: 46-50.
148. Ward, W. A., and Venkataraman, B., 1999. Some Observations on Software Quality. In ACM Proceedings of the 37<sup>th</sup> Annual Southeast regional conference, Article No.2.
149. Washizaki, H., Hirokazu, Y., Yoshiaki, F., 2003. A Metrics Suite for Measuring Reusability of Software Components, Proceedings of the 9th International Symposium on Software Metric, pp: 211-223.
150. Weyuker, E. J., 1988. Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering. 14(9), pp: 1357-1365.
151. Wilkie, F.G., and Harmer, T.J., 2002. Tool Support for Measuring Complexity in Heterogeneous Object-Oriented Software. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM.02), Montreal, Canada, October 03-06, pp:152-161.
152. Xia, W., Capretz, L. F., Ho, D., and Ahmed, F., 2008. A new Calibration for Function Point complexity weights. Information and Software Technology, 50(7-8), pp: 670-683.
153. Zadeh, L. A., 1994. Fuzzy Logic, Neural Networks, and Soft Computing. Communications of the ACM. 37(3), pp: 77-84.
154. Zhang, J., Li, H., and Cai, X., 2008. Research on Reusability of Software Connector Based on AOP. In the IEEE Proceedings of 2008 International

- Conference on Computer Science and Information Technology (iccsit2008), pp:113-117.
155. Zhao, J., 1999. Slicing Concurrent Java Programs. In Proceedings of 7<sup>th</sup> *IEEE International Workshop on Program Comprehension*, May 1999, pp.126.
  156. Zhao, J., 2002. Change Impact Analysis for Aspect-Oriented Software Evolution. In Proceedings of the International Workshop on Principles of Software Evolution, pp: 108-112.
  157. Zhao, J., and Xu, B., 2004. Measuring Aspect Cohesion. In: Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE'2004), March 29-31, LNCS 2984, Springer-Verlag, Barcelona, Spain, pp.54-68.
  158. Zhao, J., 2004. Measuring Coupling in Aspect-Oriented Systems. In: 10<sup>th</sup> International Software Metrics Symposium (Metrics 04).
  159. Zuse, H., 1991. Software Complexity Measures and Methods, Walter de Gruyter, Berlin– New York.