Chapter 3: METRICS FRAMEWORK FOR AOS
TESTABILITY ON UML DIAGRAM

3.1 Introduction

Nowadays software is percolating in all walks of our life and society. Software productivity has been increased for past several years. Therefore, the demand of producing and maintaining software products is increasing rapidly. It is challenging for computer professionals to develop more complex software system and maintain them properly. An acknowledgeable number of software projects exceed budgets due to missing schedules which lead to poor quality software and sometime improper functionality. Software professional and researchers have been looking for effective ways to deliver quality software project on time and on cost. The way out to handle this problem is software testability. The definition of testability is “Testability is the degree to which a system or component facilitates the establishment of test criteria. In the literature of software testability framework analysis, many researchers have defined their testability framework. These analysis or frameworks are intended to evaluate external software attribute such as testability. This external attribute testability can be measured with the help of software metrics. Metrics are designed on the basis of programming language like object-oriented programming and aspect oriented programming. The external attribute testability depends on internal characteristics of software system such as coupling cohesion, encapsulation and complexity. The metrics for coupling, cohesion and complexity has been proposed by various researchers. For low testability factors the cohesion should be high, low coupling and low value of complexity.

The main aim of Aspect Oriented Programming (AOP) is to improve modularization concerns Many Concerns are gathered in various classes and are known as crosscutting concerns. The ability to modularize concern tends to improve maintainability, reduce development cost and improve testability. Since Aspect oriented is a new paradigm, so there is need to redefine the definition of coupling, cohesion and complexity for AOP.

Our work is focused on most popular language Aspect J among AOP language Join points, point cut, advice, introduction and aspect are part of Aspect J. Join point includes method calls and the execution of exception handler. A point cut is a language contract that allow pick out a set of join point based on design criteria. Advice is a code that executes before, around or after a join point introduction allows aspects to modify/change
the structure of a basic program. In this chapter, a metric suite for Aspect J has been proposed, which has features of both Java language and Aspect oriented technology.

3.2 Measuring Aspect Cohesion

There are very few assessment frameworks to measure cohesion for Aspect J programming language. Detail description of the measure for cohesion is as follows.

Jianjhn Zhao et al. [165] proposed an approach to evaluate the aspect Cohesion based on dependence analysis. It describes various types of dependencies between attributes and in an aspect. Aspect oriented graph is used to represent these dependencies. Properties of the dependencies have also been described. Total three types of dependencies between attributes and /or modules in an aspect have been defined, i.e. inter-attribute, inter-module, and module-attribute dependence. But there is no inter-attribute dependence in aspect. So finally two types of dependencies are there, that is inter-module call dependence and inter-module potential dependence. Two ways for measuring aspect cohesion have been defined based on inter-attributes; inter modules and model attribute dependencies. An aspect A is assumed to consist of k attributes and n models, where k, n ≥ 0.

*Inter-attribute cohesion (Iₐ)*: This is about the internal attraction between attributes in an aspect. It is defined as:

\[
Iₐ(A) = \begin{cases} 
0 & k = 0 \\
\frac{1}{k} \sum_{i=1}^{k} \frac{|Dₐ(a_i)|}{K-1} & k > 1 \\
\frac{1}{k} \sum_{i=1}^{k} |Dₐ(a_i)| & k = 1
\end{cases}
\]

where \( \frac{|Dₐ(a_i)|}{K-1} \) show the degree on which \( a_i \) depends on other attributes in aspect A

*Cohesion between Module and Attribute (Iₘₐ)*: This is cohesion about the attraction between modules and attributes i.e.

\[
Iₘₐ(A) = \begin{cases} 
0 & n = 0 \\
\frac{1}{n} \sum_{i=1}^{n} \frac{|Dₘₐ(m_i)|}{|Dₘₐ(m_i)|} & n = 1 and |Dₘₐ(m_i)| \neq 0 \\
\text{others}
\end{cases}
\]

where \( \frac{|Dₘₐ(m_i)|}{|Dₘₐ(m_i)|} \) denotes the ratio between the number of attributes which are referred in \( m_i \), and relevant to other, to the number of all attribute linked in \( m_i \).
Cohesion between the models ($I_m$): This is evaluated by the connection between two modules. It is shown as:

$$I_m(A) = \begin{cases} 
0 & n = 0 \\
1 & n = 1 \\
\frac{1}{n} \sum_{i=1}^{n} \frac{|D_m(m_i)|}{n-1} & n > 1 
\end{cases}$$

where $\frac{|D_m(m_i)|}{n-1}$ represents the connection between $m_i$ and other module in aspect A.

**Aspect Cohesion:** This also defines another way of cohesion measurement where three facets have been integrated and obtain a discrete view of the cohesion of an aspect as follows:

$$\Gamma(A) = (I_a, I_{ma}, I_m)$$

Integrate these three facets as a whole by assuming

$$x = \alpha_1 \cdot I_a + \alpha_2 \cdot I_{ma} + \alpha_3 \cdot I_m,$$

$$\Gamma(A) = \begin{cases} 
0 & n = 0 \\
x & \text{other} \\
\alpha \cdot I_m & k = 0 \text{ and } n \neq 0 
\end{cases}$$

where $\alpha \in (0,1)$, $\alpha_1, \alpha_2, \alpha_3 > 0$ and $\alpha_1 + \alpha_2 + \alpha_3 = 1$

This method for evaluating cohesion suggests a difficult way as it can create problem in real time development process. Also aspect intendance is not included in this approach.

### 3.3 Cohesion Metric for AOSD

According to the IEEE Standard Terminology, "cohesion is the degree to which the tasks performed by a single module are functionally related." The cohesion of a component is a measure of the closeness of the relationship between its internal components. A software system is said to incur a high degree of cohesion if the elements in that unit incurs a high degree of semantic likeness. The high cohesion software development pattern suggests keeping the highest level of cohesion possible in software modules. In other words, each element in the module shall be essential for that module to achieve its purpose.

Claudio Nogueria et al. proposed a metric “Lack of cohesion in operation” in three papers Reuse and Maintenance of Aspect oriented software. It measures the amount of method pairs that do not access to the some instance variables. This metric derived from c & k metric is LCOM i.e. Lack of cohesion in methods. It measures the lack of cohesion of
a component. Component C1 is considered to have n operations which may be a method or advice, \( O_1, \ldots, O_n \) then it is set of instance variable used by operation \( O_j \). By assuming \(|Q| - \) number of non empty intersection between instance variable sets and \(|P| \) is number of null intersection. Then LCOO is

\[
LCOO = \begin{cases} 
0 & \text{otherwise} \\
|P| - |Q| & \text{if } |P| > |Q|
\end{cases}
\]

But still it is not defined that inherited operations and attribute have been included or not.

Gélinas, J., F., Badri, M., Badri, L. (2006) proposed a metric Acoh to measure aspect cohesion. The cohesion is measured on the basis of dependency analysis. The interactions between aspect’s member are defined using two aspect cohesion criteria such as Modules Data level and Modules-Modules level. These interaction define many relation between aspect member i.e.

Module-data connection criterion: Let the set of attributes used directly or indirectly by the module \( M_i \) is \( UA_{mi} \). An attribute is used directly by a module \( M_i \) if the attribute appear in its body. There are n sets \( UA_{m1}, \ldots, UA_{mn} \). Two modules \( M_i \) and \( M_j \) of an aspect are related by the \( UA_m \) relationship if \( UA_{mi} \cap UA_{mj} \neq \emptyset \). This means that there is at least one shared attribute by two modules.

Modules-modules connection criterion: \( UM_{mi} \) be the set of attributes used directly or indirectly by module \( M_i \). A module \( M_i \) is used directly by module \( M_j \) if \( M_i \) appears in the body of \( M_j \). There are n sets \( UM_{m1}, UM_{m2}, \ldots, UM_{mn} \). Two modules \( M_i \) and \( M_j \) of an aspect are related by the \( UM_m \) relation if \( UM_{mi} \cap UM_{mj} \neq \emptyset \). This means that there is at least single information shared by the two modules.

Aspect cohesion: It defines the cohesion of an aspect by the degree of relatedness of its modules. There modules can be connected by sharing attributes or by sharing modules or both. Let \( NM \) is the maximum number of connection between aspect modules. Thus in an aspect having N modules,

\[
NM(Aspect) = N*(N-1)/2, N>1
\]

\( NC(Aspect) \) is the number of connection between modules in undirected graph. Then new metric for aspect cohesion is defined as

\[
Acoh(Aspect) = NC(Aspect) / NM(Aspect) \in [0, 1]
\]

In this paper the problem related to inherited member are not clear and no indirect relationship between aspect members has been considered.
3.4 Coupling metric for AOSD

Zhao, J. (2004) proposed a coupling measure framework for evaluating the coupling in Aspect oriented systems. Coupling measure framework has been designed to count the dependencies between aspect and the classes in the systems. The mathematical properties of this measure have been also discussed. The dependencies are as follows.

**Attribute-class dependence:**

There is an attribute class dependence between aspect \( a \) and class \( c \), then the number of attribute class dependences from \( a \) to \( c \) can be defined as

\[
\text{AtC}(a, c) = | \{ x \mid x \in A^\alpha(a) \land T(x) = c \} |
\]

**Module class Dependence:**

Let \( S \) be AO system, \( a \in A(S) \) be an aspect of \( S \), and \( c \in C(S) \) be a class of \( S \). Then Advice class dependence (AC) is

\[
\text{AC}(a, c) = \sum_{a \in A(a)} | \{ x \mid x \in \text{Par}(a) \land T(x) = c \} |
\]

where \( c \) is the return type of \( a \) and \( x \) be attribute of an aspect.

**Module-method dependence:**

Advice method Dependence (AM) is

\[
\text{AM}(a, c) = \sum_{a \in A(a)} \sum_{m \in M(c)} | \text{NSI}(a, m) |
\]

It is not considering dependence between aspects or between classes.

Briand, L. C., Daly, J. W. (1999) proposed a coupling framework for Aspect Oriented System. This version of framework described definition of different coupling metrics. The Briand’s Framework has been refined and specifically targeted at the composition model supported by AspectJ. The framework has been divided in two sets of distinct AO composition model; one for AspectJ and another for feature oriented decomposition system empirical validation. But it has not provided the optional use of coupling criteria as in Brand’s Framework.
Ceccato et al. (2004) defined coupling metric for AspectJ. Classes and aspects have been considered as modules and operations as a methods and aspect advices. Following metrics for coupling have been defined such as coupling on advice execution (CAE), Coupling on intercepted module (CIM), coupling on Method calls (CMS), coupling on field Access (CFA), Response for a module (RFM) and crosscutting Degree of Aspect (CDA).

### 3.5 Design Metrics for AOS Testability Framework

Testability is broadly defined as the ease of testing a module of software design. It is a process for the software designer to test systems which are hard to test. Testability design framework plays an important role in software engineering. This tends to make validation phase more efficient by exposing the faults during testing so as to increase the quality of end product. Testability measurement of a software helps in planning all the activities associated with testing and assign the required resources. Testability of software can be effectively measured from the testability effort. In this chapter set of metrics-based on UML, Which measure testability of UML class diagrams has been defined in the testability framework. It has strong relationship between the aspect oriented metrics and the testability of the software. Following section describes the methodology for designing the proposed framework to improve testability.

#### 3.5.1 Methodology

Software with good testability is important because test tasks are eased and test costs are reduced. The ISO model defines testability of software system as the attributes of software that abide on the effort needed to validate a software product. The proposed framework has been computed from class diagram using various metrics for AOP as shown in figure 3.1. A methodology has been implemented step by step for designing the framework. In particular it has been attained as follows:

**Step 1:** The proposed method consists in analyzing testability on the class diagram. The testability of class diagrams is essential for later design work and could be a major determination for the quality of the software product. Quantitative measurement mechanisms have been useful to assess class diagram quality in an objective way.
**Step 2:** Minimum and maximum values for each metric have been defined to measure the favorable value of testability. These values have been considered on the basis of past software development experiences and by the recommendation of the experts in this field. Decision of modifying the UML diagram of module has been taken on the basis of these values at each step.

**Step 3:** Metrics suite has been proposed for aspect oriented system. In this framework of 9 metrics some of the metrics has been extended, to make them valid to the AOP software. Metrics values for various class diagrams have been computed by applying metric collection for Aspect Oriented Systems.

**Step 4:** From the metric values calculated in step 3 average values per module have been computed for each metric. The metric values for each class of a particular module have been added together and divided by total number of classes. This gives the average metric values for each module.

**Step 5:** The testability of a class diagram has been evaluated by making a Class Dependency Graph (CDG) from the UML class diagram. If the testability is high then the feedback is sent back for design modification.

**Step 6:** A performance analysis has been done with the help of Kiviat diagram. A benchmark has been defined on the basis of minimum and maximum values computed in step 2. The values of metrics computed in step 3 have been put into Kiviat diagram and decision has been taken by referring the benchmark values.

A high degree of testability indicates that any existing faults can be revealed relatively during testing, inputs can easily be selected to satisfy some testing criteria and outputs of state variables can be observed during testing. If the values of metrics still exceed the recommended values and there is no possibility to modify the existing design then the design is considered as rejected. Otherwise the design is accepted and the testability efforts are predicted. Further the design is sent for implementation.
3.5.2 Metrics collection for UML class diagram

- **Depth of Inheritance Tree (DIT):** Defines the length of the longest path form a given module to the class hierarchy root. Aspect can change the inheritance relationship by means of static crosscutting. During computing this DIT metric the effects of aspectization must be consider.

**Average Depth of Inheritance (ADIT):**

\[
ADIT = \frac{1}{n} \sum_{i=0}^{n} DIT_i, \quad DIT_i \neq 0 \quad \ldots \quad (3.1)
\]

Where \( DIT_i \) is the Depth of inheritance tree for class \( i \), and \( n \) is the total number of classes.
• **Number of Children (NOC):** Gives the number of immediate subclass or sub aspect of given module. NOC evaluate in the reverse direction with respect to DIT the number of children of a class indicates the proportion of class/module potentially dependent on properties inherited from the given position.

**Average Number of Children (ANOC):**

\[ ANOC = \frac{1}{n} \sum_{i=0}^{n} NOC_i, \quad NOC_i \neq 0 \quad \ldots \quad (3.2) \]

Where \( NOC_i \) is the number of children for class \( i \), and \( n \) is the total number of classes.

• **Weighted Method in Class (WMC):** Represents the number of Methods/operation in class or aspect. It defines the internal complexity of a class in terms of the number of implement functions. In more detail this metric can be obtained by giving appropriate weights to methods with various initial complexities.

**Average Weighted Method in Class (AWMC):**

\[ AWMC = \frac{1}{n} \sum_{i=0}^{n} WMC_i, \quad WMC_i \neq 0 \quad \ldots \quad (3.3) \]

Where \( WMC_i \) is the weighted method for class \( i \), and \( n \) is the total number of classes.

• **Coupling on Method Call (CMC):** Denotes the number of interfaces declaring methods that are called by a given Aspect. In this metric aspect introductions must be taken into account when introduction invoked methods. A number of methods from various modules or class indicate that the function of the given class cannot be easily isolated from the others. High coupling is associated with dependence from the function is other classes.

**Average Coupling on Method Call (ACMC):**

\[ AMC = \frac{1}{n} \sum_{i=0}^{n} CMC_i, \quad CMC_i \neq 0 \quad \ldots \quad (3.4) \]

Where \( CMC_i \) is the coupling on method call for class \( i \), and \( n \) is the total number of classes.

• **Coupling Between Modules (CBM):** This metric is defined as a number of modules or interfaces declaring methods or fields that are called by a given module.
Average Coupling Between Modules (ACBM):

\[ ACBM = \frac{1}{n} \sum_{i=0}^{n} CBM_i, \quad CBM_i \neq 0 \quad \ldots \quad (3.5) \]

Where \( CBM_i \) is the coupling between modules for class \( i \), and \( n \) is the total number of classes.

- **Response For a Class (RFC):** This is a measure by number of methods and advices potentially executed in response to a message received by a given class. It is associated with the implicit response that is triggered whenever a point cut intercepts an operation of the given class.

Average Response For a Class (ARFC):

\[ ARFC = \frac{1}{n} \sum_{i=0}^{n} RFC_i, \quad RFC_i \neq 0 \quad \ldots \quad (3.6) \]

Where \( RFC_i \) is the Response for a class for class \( i \), and \( n \) is the total number of classes.

- **Crosscutting Degree of an Aspect (CDA):** Number of classes affected by the point cuts and by the introduction in a given aspect. The proposed metric can be used to compose alternative AOP implementations, and cannot be applied when an OOP program is migrated to AOP. CDA gives the number of classes that are affected by an aspect without being referenced explicitly by the aspect. This indicates the degree of generality of an aspect. High value of CDA is desirable. CDA gives an idea of the overall impact an aspect has on the other classes or modules.

Average Crosscutting Degree of an Aspect (ACDA):

\[ ACDA = \frac{1}{a} \sum_{i=0}^{a} CDA_i, \quad CDA_i \neq 0 \quad \ldots \quad (3.7) \]

Where \( CDA_i \) is the Crosscutting Degree of an Aspect for class \( i \), and \( a \) is the total number of aspects for a class.

- **Coupling on Advice Execution (CAE):** Its behavior is opposite to the CDA metric. CAE is number of aspects containing advices possible triggered by the execution of operation in a given class. If the working of an operation can be changed by an aspect advice, due to an intercepting pointcut, then there is a dependence of operation from the advice. Thus the given class is coupled with the aspect and any change afterwards might affect the former.
Average Coupling on Advice Execution (ACAE):

\[ ACAE = \frac{1}{a} \sum_{i=0}^{n} CAE_i, \quad CAE_i \neq 0 \quad \text{…… \ (3.8)} \]

Where \( CAE_i \) is the coupling on advice execution for class \( i \), and \( a \) is the total number of aspects for a class.

- Number of Pointcuts (NOP): Defines the number of pointcut belonging to a given aspect. NOP captures the direct knowledge an aspect has of the rest of the system. High value of NOP define high coupling of the aspect with the given software and indicated law reusability. The applicability of a measure depends on the level of details of the UML class diagram which may vary from one development environment to another.

Average Number of Pointcuts (ANOP):

\[ ANOP = \frac{1}{a} \sum_{i=0}^{n} NOP_i, \quad NOP_i \neq 0 \quad \text{…… \ (3.9)} \]

Where \( NOP_i \) is the number of pointcuts for class \( i \), and \( a \) is the total number of aspects for a class.

Once the value of various metrics has been calculated for testability measure from UML class diagrams, then how to use it for decision making? Various approaches have been used to predict the testability from this data collected from metrics. Such as taking average of metrics value per class, knowledge based approach using previous experiences.

**Hypothesis 1:** From the metric values associated with coupling and cohesion, a high value of coupling and inheritance indicates that the class possesses low testability and therefore it shall be difficult to test. Further we can make some changes to the design on the basis of the metric values computed which are not acceptable.

**Hypothesis 2:** According to the past project experiences if the average metric values computed so far is above the preferred range, this indicates that the particular module needs some modifications to satisfy these values.
3.5.3 Complexity of a Aspect Oriented class diagram

Class dependency graph (CDG) has been derived from a UML class diagram. This serves as a basis for applying classical graph algorithm to detect interaction and measure their complexity. It consists of set of vertex and a set of directed edges.

\[ CDG = \{V, E\} \]

where V is the set of vertices, each vertex represents a class or aspect of an Aspect oriented system one class aspect is represented by one vertex. E is a set of directed edges. An edge is between two classes or between class and aspect.

Even path of CDG must cross the hierarchy only in one direction. This reflects that only one edge going form parent vertex to child vertex and vice versa in a particular path. The complexity of CDG will increase due to the classes in an inheritance hierarchy communicating with their ancestor and child class. This shows the effect on the testability of the class diagram since many paths are involved the more test cases are represent to test the entire path. Also of the length of path is more than more inheritance hierarchies at traverse and it is difficult to write test case for long path. The complexity of an interaction is the combination of the complexity of all paths in the interaction. Descendent path is the set of classes crossed by a path going from the root class of the hierarchy to a leaf class complexity of descendents path.

Complexity (Com) of descendant path of a class diagram is

\[ Com = \sum_{i=1}^{n} h(h - 1) \]

Where h is the height of given class in the tree.

Hypothesis 3: The equation above gives the complexity of a class diagram which is directly proportional to the height of class. It has been seen that longer distance between the vertices or large number of paths leads to high complexity.

3.5.4 Performance Analysis

The data collected from various metrics has been used in making decision for testability. A benchmark has been decided on the basic of data collected from past projects. Typical range has been decided for each metric for testability. For good measure of testability a minimum and maximum value has been fixed for each testability metrics. A
Kiviat diagram has been used for this decision making process. Kiviat diagram contains radically extending lines each of which represents a testability measure and two concentric circles represent the acceptable minimum and maximum value that has been fixed for testability metric.

![Figure 3.2 Kiviat Diagram](image)

Kiviat diagram contains radically extending lines. Each line represents a testability measure and the two concentric circles represent acceptable minimum and maximum value of metric as taken from the past projects. Kiviat diagram has been used to assess the software products and to identify unused cases with respect to various measurement dimensions. Kiviat diagram’s visualizing tool is used for measuring project characteristics with the help of benchmarks on multiple dimensions. This helps in making decisions or suggesting any improvements in the design of a software project. The estimation for a new
system has been developed by retrieving already developed software systems with similar metric values.

### 3.6 Properties of Metrics used in the Framework

Various properties of metrics have been discussed in table 3.1. All the ten metrics described in section 3.5.1 have been considered. The properties show the measurement granularity, measurement entity, measurement type and fan in and fan out of a particular metric. Metrics related to inheritance have been evaluated at class level. Those related to coupling have been evaluated at module/class level by examining the interactions between methods or advices.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Measurement Granularity</th>
<th>Measurement Entity</th>
<th>Measurement Type</th>
<th>Fan in Fan out</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIT</td>
<td>Class</td>
<td>Class / aspect</td>
<td>Inheritance</td>
<td>NA</td>
</tr>
<tr>
<td>WMC</td>
<td>Class</td>
<td>Method / advice</td>
<td>Environmental</td>
<td>NA</td>
</tr>
<tr>
<td>NOC</td>
<td>Module</td>
<td>Class / aspect</td>
<td>Inheritance</td>
<td>NA</td>
</tr>
<tr>
<td>CBM</td>
<td>Class</td>
<td>Method / advice</td>
<td>Environmental</td>
<td>FO</td>
</tr>
<tr>
<td>RFC</td>
<td>Class</td>
<td>Method / advice</td>
<td>Environmental</td>
<td>FO</td>
</tr>
<tr>
<td>LCO</td>
<td>Class / aspect</td>
<td>Method / advice</td>
<td>Environmental</td>
<td>NA</td>
</tr>
<tr>
<td>CDA</td>
<td>Module / class</td>
<td>Pointcut / introduction</td>
<td>Environmental</td>
<td>NA</td>
</tr>
<tr>
<td>CAE</td>
<td>Module / aspect</td>
<td>Advice</td>
<td>Environmental</td>
<td>FO</td>
</tr>
<tr>
<td>NOP</td>
<td>Class / aspect</td>
<td>Pointcut</td>
<td>Environmental</td>
<td>NA</td>
</tr>
</tbody>
</table>

### 3.7 Conclusion

The proposed framework is useful in describing the existing metrics and measures for AOS. A Metrics suite for cohesion, coupling and complexity has been defined for AOS such as DIT, WMC, NOC, CBM, RFC, LCO, CDA, CAE, NOP and Class Dependency Complexity. These metrics are designed to work for AspectJ. The proposed metrics have a large influence on the testability of Aspect Oriented Software Systems. Empirical and quantitative validations of the proposed metrics have been performed by computing average metrics per class and applying fuzzy approach taking some of the most effective metrics in chapters 4 and 5. Efforts have been made in designing the framework for AOSD which is useful in describing existing metrics and measure for internal characteristics of Aspect Oriented Software Systems. The results indicate that the framework is useful for computing testability index.