Chapter - 2

Background and Review of Literature
CHAPTER - 2

BACKGROUND AND REVIEW OF LITERATURE

2.1 Introduction

Software metrics are essential means to measure the quality of software products. Software Engineering metrics are units of measurement that are used to characterize products, processes and resources. Measurement is used by practitioners to assess situations, track progress, and evaluate the effectiveness and a lot more.

According to Fenton [Fen, 97], “Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way that it describes them according to clearly defined rules”. These symbols and numbers are abstractions of the perceptions of the real world. It helps to analyze and to pass judgments about the entities. As suggested by Galileo Galilei, “What is not measurable make measurable”. When entities are measurable, they become more visible and understandable. So, measurement is an essential process.

The primary objectives of software measurement [Joh, 13] are to determine the quality of the current product, to predict the qualities of software, to improve the quality of the product and the process that produce the product.

This chapter enumerates the essential properties of software metrics, benefits and difficulties associated with software measurement. It explains different types of metrics, differences between Object Oriented (OO) and AOP metrics and some of the metrics suites which are discussed widely in this chapter.
2.2 Introduction to Aspect-Oriented Programming

It is important to note that AOP is not a standalone programming paradigm like object oriented programming. AOP is always used together with other programming paradigms. It can best be described in terms of concerns, join points and aspects. Concerns are system properties or areas of interest in a system. Separation of concerns is a main principle in software engineering. Concerns crosscut if the methods related to those concerns intersect, either inside a class or over several classes. AOP provides a way of encapsulating crosscutting concerns. This can be seen in Figure 2.1, where all the marked methods have to check user rights.

![Crosscutting concerns in UML class diagram](image)

**Figure 2.1 Crosscutting concerns in UML class diagram**

Checking user right concerns can be based on both functional and non-functional requirements, for example, logging, security, caching and buffering. Joinpoints are the locations which are affected by one or more crosscutting concerns. Figure 2.2 shows an example of joinpoints in the calling and returning points of a method. Joinpoints are the locations where the user can hook on new actions before or after executing the original code.
2.3 AspectJ

AspectJ [Ana, 15] is a general purpose Aspect-Oriented extension of Java. It is a free implementation and language specification developed at Xerox PARC. The language specification defines several constructs and their semantics to support Aspect-Oriented concepts. The language implementation consists of tools for compiling, debugging, and documenting code. AspectJ’s language constructs extend the Java programming language. Every valid Java program is also a valid AspectJ program. The byte code produced by the AspectJ compiler is standard Java byte code, thus it keeps the advantages of Java.

AspectJ enables both name-based and property-based crosscutting. Aspects that use name-based crosscutting tend to affect a small number of other classes. But despite their small scale, they can often eliminate significant complexity compared to an ordinary Java implementation. Aspects that use property-based crosscutting can have small or large scale projects. Aspects are fundamental to the definition of AOP. Aspects are design decisions that are difficult to address in regular OO-code because
they crosscut the system. With AOP the user can separate the aspects from the underlying structure of the code, as in the example shown in Figure 2.1. The user can move the updating of the display to a separate subprogram. This subprogram is called an aspect. The core constructs of AspectJ are discussed in the following sub-sections.

### 2.3.1 Joinpoints

Joinpoints are well-defined points in a program’s execution. In AspectJ the following points in the program execution can be used as joinpoints:

- Method call and execution
- Constructor call and execution
- Read/write access to field
- Exception handler execution
- Object and class initialization execution

### 2.3.2 Pointcuts

Pointcuts are program constructs used to designate joinpoints; they let the user specify a joinpoint collection. They can also expose context at the joinpoint location for use in an advice implementation.

The following example depicts the pointcut:

```java
pointcut publicGet(Object key) : execution(*AdressList.get*(..) && args(key));
```

The Figure 2.3 depicts pointcut, which defines pointcut. PublicGet is the pointcut’s name and Object key is the context that is collected from the joinpoint location.
2.3.3 Advices

Advices specify the executable code when reaching a certain pointcut. There are three types of advices, as seen in Table 2.1.

Table 2.1 The Advice-Types available in AspectJ

<table>
<thead>
<tr>
<th>Advice Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before()</td>
<td>A before advice runs just before the joinpoint.</td>
</tr>
<tr>
<td>After()</td>
<td>An after advice runs just after the joinpoint, and can be specified to run after a normal return, after throwing an exception, or regardless of what kind of return from a joinpoint it is.</td>
</tr>
<tr>
<td>Around()</td>
<td>An around advice encapsulates a joinpoint and controls if the joinpoint is to be executed or not. It can also execute a different argument set.</td>
</tr>
</tbody>
</table>
Figure 2.3 shows two around advices. One of them is:

```java
after(Object key) returning(value): publicGet(key)
{
    cacheValue(key, value);
}
```

This advice is executed after the method publicGet has executed and returned (but before execution is handed over to the method that called publicGet). The advice fetches the context Object key, and executes the method cacheValue, with key and value as arguments. Then execution continues as normal.

### 2.3.4 Intertype Declarations

Inter-type declarations are declarations that cut across classes and their hierarchies in AspectJ. They may declare members that cut across multiple classes, or change the inheritance relationship between classes. The problem of expressing a capability shared by some existing classes that are already parts of a class hierarchy, i.e. they already extend a class. In Java, one creates an interface that captures this new capability, and then adds a method to each affected class that implements the interface. AspectJ can express the problem in a single place, by using intertype declarations. The aspect declares the methods and fields that are necessary to implement the new capability, and associates the methods and fields to the existing classes.

### 2.3.5 Aspect

Aspects are AspectJ’s unit of modularization, the same way classes are in Java. An aspect contains pointcuts and advices, and as classes, can have methods and
fields, extend other classes and aspects and implement interfaces. Aspects differ from classes in that the user cannot create an aspect object using new, as aspects cannot be accessed directly. By default, each aspect is a singleton, so one aspect instance is created. Only aspects can hold advices, but classes can declare static pointcuts. Both aspects and pointcuts can be declared as abstract and act similarly to a class’ abstract methods; they let you defer details to the derived aspects. A concrete aspect extending the abstract aspect can then provide concrete definitions for abstract pointcuts. An aspect can be declared dominant over others to control precedence when multiple aspects affect the same joinpoint. If an aspect is declared privileged, it is given access to private members of aspected classes.

The introduction of AspectJ was given by an example of how caching can be implemented as an aspect. In the example, shown in Figure 2.3, an address list which can be accessed using keys like phone number or name, and it return values like address or name. The real address list might be stored in a database and accessed with SQL statements. Therefore caching might be needed to improve performance. The aspect provides this function by checking if the called key is found in a caching table. Every time, when returning a value that is not cached, the value and its key is cached.

2.4 Software Metrics

Software measurement is concerned with deriving a numeric value for an attribute of a software product or process. It can be used to determine the quality of the current products or process, predict the qualities of a product/process and also to improve the quality of a product/process. Software metric is a quantitative measure of some property of a piece of software or its specifications. It is a measure of the
difficulty of testing, understanding, or maintaining a piece of software. It also provides a measure of ease of using software. Software metrics provide a quantitative basis to gain insight into the efficiency of the software process and development. Metrics can be used to assess and improve the quality. The better use of software metrics can improve product quality, productivity, better cost estimates and help in effective management and control of development process.

Metrics for aspect-oriented system may be 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.

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 prerequisite 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.

Software metric is the objective, quantifiable and reproducible measure of some properties of software or its specifications. It is a mechanism for characterizing, evaluating, and predicting various software processes and products. The role of software metrics is to analyse and significantly improve the software quality, the process of efficiency, the effectiveness of the product and direct management to take
managerial and technical decisions [Sri, 14a] [Sri, 14b] [Sri, 15]. Therefore, the only way to improve the quality of any software process or product is to measure the specific attributes of the process or product and to develop a set of metrics based on quality attributes such as maintenance, portability etc. and by using those metrics to provide indicators that will lead to strategies for improvement [Sri, 14b].

2.4.1 Types of Software Metrics

In general, software metrics can be classified into three categories: product metrics, process metrics, and project metrics. Strictly speaking, the project metrics is not required in the main classification because software engineering is mainly concerned with process metrics and product metrics. Any metric in software measurements may come under only these two classifications [Sri, 15].

**Project Metrics:** These are concerned with the software project quality. Project metrics describe the project characteristics, forecasting the time duration, required resources, cost estimation, and execution such as staffing number and pattern over the software life cycle. They indicate whether process executions are on track. They are used to quantify defects, cost, schedule, productivity and estimation of various project resources and deliverables. They are also used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.

**Process Metrics:** These are metrics that pertain to process quality. Process metrics deal with the software production process and are used to improve software development. They are used to measure the properties of the development process and embed aspects of maintainability, testability etc. The process metrics are also used to improve various features of running the software system. They measure the efficiency
and effectiveness of various processes. These metrics are known as software management metrics [Gur, 11]. They include the cost metrics, efforts metrics, advancement metrics, and reuse metrics. Process metrics help in predicting the size of final system and portability of software on various hardware and software platforms.

**Product Metrics**: Product metrics pertain to product quality, and it is an effort to increase its quality. Product metrics relate directly to the result of a software development process. The software product metrics are software measures of the product at any stage of its software development. Product metrics include the architectural measures composed of internal or direct or quality attribute metrics such as program or module or component size, count of components like variables, methods, classes etc. and external or indirect metrics. Quality criteria like coupling, cohesion, polymorphism, inheritance, accessibility, information hiding, functionality, portability, reliability, usability, maintainability etc. are also included in product metrics. Product metrics are also known as quality metrics [Gur, 11].

### 2.4.2 Benefits of Software Metrics

The primary benefit of Software metrics is to help the programmers as well as the managers to monitor and to improve the quality of the software. Apart from this, the basic benefits of software measurement [Xie, 99] [Gur, 11] are as follows:

- Aids to form a baseline for cost and time estimation for future software.
- Helps to understand both design and architecture information of the software system by providing a neat picture of the complexity of the code.
- Assists in the preparation of software quality specifications and verification of compliance with software systems requirements and specifications.
• Provides useful information about the current state of the products, processes and resources in different stages of software development life cycle.
• Helps to assess the situation so that the managers and practitioners can make timely decisions and avoid major risks.
• Enables to track progress at any given situation and also continuously aids to improve the software productivity.
• Supplies critical information about the reliability of the software being built.
• Assists to uncover the underlying errors in the software design at an early stage of the software development cycle.
• Serves information on maintainability of the system and helps to embed appropriate mechanisms in the software development process.
• Empowers to assess the quality of design and helps to identify complex modules for the further division to achieve better design.
• Serves to evaluate the quality, productivity and competency of the developers.
• Assists to test the effectiveness of software and the technology that is used such as the OO reuse technology etc., based on the requirement analysis.
• Helps to do a comparative study of various design methodologies of software systems.
• Promotes analysis, comparison, and critical study of various programming languages concerning their characteristics.
• Enables to assess the benefits derived from new software engineering tools, methods, and training and to justify for further procurement of new tools and trainings.
• Guides resource managers for optimal utilization of the resources.
• Helps to identify maintenance problems after the release of the software, to make design trade-offs between software development and maintenance cost.
• Helps to estimate the required effort for design and development of the software.

2.5 Object-Oriented Metrics

Traditional metrics such as Lines of code, Cyclomatic Complexity, Function Points, COst COnstructive MOdel (COCOMO), Halstead metrics and others are outside the scope of this research since it focuses only on the metrics connected with the OO paradigm. Further, Chidamber and Kemerer (CK) [Chi, 94] argue that because of the fundamental difference between traditional approach and OO approach, software metrics developed with traditional methods in mind do not consider the OO features like encapsulation, inheritance, polymorphism, and message passing. So, traditional metrics do not support key OO concepts, and therefore, it is suitable to have new metrics especially designed to measure the unique aspects of the OO design. A multitude of OO metrics are proposed by software engineers and research scholars. Some of them have proposed even suites of OO metrics. Some of the popular and often referenced OO complexity metric suites are Metrics for Object Oriented Software Engineering (MOOSE) proposed by Chidamber and Kemerer in 1991 and popularly known as Chidamber and Kemerer (CK) metric suite [Chi, 91], Lorenz and Kidd (LK) metric suite suggested by Mark Lorenz and Jeff Kidd [Lor, 94], MOOD proposed by Fernando Brito Abreu and Rogério Carapuça in 1994 [Abr, 94], the second set of MOOD metrics called the MOOD2 in 2001 [Abr, 01] and the Quality
Metrics for Object Oriented Design (QMOOD) proposed by Jagdish Bansiya in 2002 [Ban, 02].

2.5.1 Object-Oriented Cognitive Complexity Metrics

This section traces chronologically the metrics proposed specifically for Object-Oriented measurements and cannot be applied to any another programming style. Because, it is conceived that OO structural properties such as encapsulation, coupling, cohesion, functional complexity, inheritance, and polymorphism have an impact on the Class Complexity (CC) of the system. The metrics presented here are class level CC metrics.

Sanjay Misra [San, 07] proposed ‘Class Complexity’ in 2007, which sums up Weighted Classes (WCs) of methods in the class and to find the overall software system class complexity, the class WCs are summed up if they are in the same level and multiplied in case of subclass or children. Later, he modified it and called ‘Weighted Class Complexity’ metric [San, 08] by adding the complexity of attributes and message calls. Arockiam et al. [Aro, 09] extended it and proposed Extended Weighted Class Complexity (EWCC) metric by adding the Cognitive Weights (CWs) due to inheritance. They have also validated sample program comprehension and done a comparative study with similar class level complexity metrics. Mishra [Mis, 09] suggested two metrics for inheritance, ‘Class Complexity due to Inheritance’ which is the addition of sum of inherited class complexity and method complexity within that class based on Cyclomatic Complexity, and ‘Average Complexity of a program due to Inheritance’ which is the ratio of sum of class complexity due to inheritance to the total number of classes in the program. Sanjay Misra et al. [San, 07] proposed
‘Cognitive Code Complexity’ which is the product of the sum of class complexity and validated theoretically and empirically. Usha Chhillar et al. [Chh, 11] proposed the ‘Weighted Composite Complexity Measure’ based on the complexities of inheritance level, type and nesting level of control structures and size of the class/program. Chhabra proposed the ‘Code Cognitive Complexity’ based on the spatial distance of module call from its definition.

Arockiam et al. proposed [Aro, 11] Attribute Weighted Class Complexity (AWCC) which is the addition of the summation of the attribute, method and inherited class complexities. The attribute complexity includes the cognitive complexity due to different data types of the attributes such as primary, derived and user-defined data types. They have also proposed class level Cognitive Weighted Response For Class (CWRFC) [Alo, 12b] which includes the internal complexity of the class and the response set complexity. They have proposed yet another class level complexity metric called Cognitive Weighted Coupling Between Objects (CWCBO) [Alo, 12a] which is the sum of the cognitive weighted control coupling, global coupling, internal coupling, data coupling, and lexical content coupling.

In 2012, Sanjay Misra et al. [San, 12] proposed a suite of cognitive complexity metrics to measure method complexity, message complexity, attribute complexity and class complexity. All of them are based on CC except the attribute complexity. It includes the ‘Method Complexity’ by considering the cognitive weights of structures in a method of a class. Next is the ‘Coupling Weight for a Class’ which is nothing but message complexity. It is the sum of weights of call and the weight of called methods. The ‘Attribute Complexity’ is a non-cognitive complexity which is nothing but the
total number of attributes. The ‘Weighted Class Complexity’ which is the sum of all the attribute and the method complexities of a class. The ‘Code Complexity’ calculates the complexity of relations among the classes of the entire system namely, inheritance. It adds the weights if the classes are on the same level, else it multiplies in the case of subclasses. The ‘Average Method Complexity’ is the ratio of the sum of complexities of all the methods in a class to the total number of methods in that class. The ‘Average Method Complexity per Class’ is the average method complexity for the entire system. The ‘Average Class Complexity’ is calculated by dividing the sum of the complexity of the classes to the total number of classes. The ‘Average Coupling Factor’ is the complexity of all the external method calls to the total number of messages. The ‘Average Attributes per Class’ is calculated by dividing the sum of attribute complexity of all classes to the total number of classes.

Isola et al. [Iso, 16] proposed ‘New Cognitive Complexity of Program’ which is based on the operands, internal behaviour of the software and the data objects including input and output. Rim et al. [Rim, 14] defined the ‘Scope Information Complexity Number’ and prevailed the cognitive complexity based on the functional decomposition of software, including theoretical validation through nine Weyuker's properties.

2.6 Aspect-Oriented Programming (AOP) Metrics

Some AOP metrics have so far been proposed for AO programs. Some of them have been adapted from object-oriented and transposed to account for AO mechanism. For instance, both Ceccato and Tonella [Cec, 04] and Sant’ Anna et al. [San, 03] have proposed maintainability metrics adapted from an Object-Oriented
(OO) metrics suite proposed by Chidamber and Kemerer [Chi, 94]. These metrics can be applied to both OO and AO programs.

Table 2.2 AOP Metrics

<table>
<thead>
<tr>
<th>Author</th>
<th>Measures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kotrappa Sirbi, Prakash Kulkarni [Kot, 10]</td>
<td>Response To Module (RFM)</td>
<td>Some methods and advice potentially executed in response to a message received by a given module are called RFM.</td>
</tr>
<tr>
<td></td>
<td>Weighted Operations in Module (WOM)</td>
<td>WOM metric defines Number of operations in a given module and its equivalent to the WMC metrics from CK metrics suite.</td>
</tr>
<tr>
<td>Lopez-Herrejon, Apel [Lop, 07]</td>
<td>Feature Crosscutting Degree (FCD)</td>
<td>FCD corresponds to the number of classes that are crosscut by all pieces of advice in a feature and those crosscut by the inter-type declarations.</td>
</tr>
<tr>
<td></td>
<td>Advice Crosscutting Degree (ACD)</td>
<td>ACD corresponds to the number of classes that are crosscut exclusively by the pieces of advice in a feature.</td>
</tr>
<tr>
<td></td>
<td>Homogeneity Quotient (HQ)</td>
<td>HQ is a division of the advice crosscutting degree (ACD) by the feature crosscutting degree (FCD).</td>
</tr>
<tr>
<td></td>
<td>Program Homogeneity Quotient (PHQ)</td>
<td>PHQ corresponds to the summation of the homogeneity quotients for all the features in a program, divided by the number of features (NOF).</td>
</tr>
<tr>
<td>Cecceto and Tonella [Cec, 04]</td>
<td>Coupling on Intercepted Modules (CIM)</td>
<td>CIM defines some modules or interfaces explicitly named in the pointcuts belonging to a given aspect.</td>
</tr>
<tr>
<td></td>
<td>Coupling on Method Call (CMC)</td>
<td>CMC defines some modules or interfaces declaring methods that are possibly called by a given module.</td>
</tr>
<tr>
<td></td>
<td>Coupling on Field Access (CFA)</td>
<td>CFA defines some modules or interfaces declaring fields that are accessed by a given module.</td>
</tr>
<tr>
<td></td>
<td>Coupling on Advice Execution (CAE)</td>
<td>CAE defines of aspects containing advice possibly triggered by the execution of operations in a given module.</td>
</tr>
</tbody>
</table>
Table 2.2 gives the brief details of the so far defined aspect-oriented metrics. Some of them are adapted from OOP metrics suite while some of them are new to AOP. The most frequent metrics were adapted from Object-Oriented (OO). Adapted metrics hold the advantage of being based on OO metrics that are widely used, and can be assumed reliable. The implicit reasoning is that adapting OO metrics to AOP maintains their usefulness. Dependence between aspect and class in an AO system is a key point to determine the mechanism by which an aspect and class are coupled. Some of these metrics which have been converted from OO to AO are: RFC (Response For a Class) evolved to RFM (Response For a Module), WMC (Weighted Methods per Class) evolved to WOM (Weighted Operations per Module) and many more.
2.6.1 Aspect-Oriented Programming (AOP) Cognitive Complexity Metrics

E. Alemneh [Ale, 14] presented the current state of Aspect-Oriented Programming metrics. AOP is a new technology that has better capability to handle crosscutting concerns than Object Oriented. However, concerning metrics for this new programming paradigm, the work is in its infancy. But he had surveyed, summarized and reviewed all internal metrics for AO systems.

One of the metrics proposed by Ceccato et al. [Cec, 04a] and Kotrappa Sirbi et al. [Kot, 10] is Weighted Method per Class (WMC). It's an equivalent of the WMC metric from CK Metrics suite [Chi, 94] in Object Oriented Programming. WMC counts some methods or advice in a given module. This metric does not consider the various types of advice. Coupling on Advice Execution (CAE) is some aspects containing advice possibly triggered by the execution of operations in a given module. Such kind of coupling is absent in Object Oriented (OO) systems.

Parthipan, Senthil Velan and Chitra Babu [Par, 14] presented the Design Level Metrics to Measure the Complexity of Versions of AO Software. The value of WAA metric is calculated as the sum of the cognitive weight of all advice types in an aspect. The advice are classified into three types, namely, before, after and around advice. The cognitive weights assigned to the advice types are based on the cognitive complexity. The Weighted Joinpoint (WJP) per class or aspect is the sum of cognitive weights of types of joinpoints shadow in classes or aspects. The cognitive weight assigned to the identified designators based on its cognitive complexity. WPA is calculated by adding the cognitive weight of the pointcut designator and cognitive weight of the joinpoint signature used in an aspect.
Bartsch and Harrison [Bar, 06] suggested that all joinpoints can also cause the execution of advice. AspectJ supports more types of joinpoints such as object initialization joinpoints, exception handler joinpoints, call joinpoints and advice execution joinpoints. A valid measure of coupling on advice execution needs to counts all of these joinpoint coupling mechanisms.

The following Table 2.3 shows that the various metrics of AOP and their attributes.

**Table 2.3 Metrics of AOP and their Attributes**

<table>
<thead>
<tr>
<th>Authors</th>
<th>Measures</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kottrappa Sirbi, Prakash J Kulkarni [Kot, 10]</td>
<td>RFM (Response For Module)</td>
<td>Software quality</td>
</tr>
<tr>
<td></td>
<td>WOM (Weighted Operations in Module)</td>
<td></td>
</tr>
<tr>
<td>Roberto E. Lopez-Herrejon, Apel [Rob, 07]</td>
<td>Feature Crosscutting Degree (FCD)</td>
<td>Crosscutting</td>
</tr>
<tr>
<td></td>
<td>Advice Crosscutting Degree (ACD)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Homogeneity Quotient (HQ)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program Homogeneity Quotient (PHQ)</td>
<td></td>
</tr>
<tr>
<td>Cecceto and Tonella [Cec, 04]</td>
<td>CIM (Coupling on Intercepted Modules)</td>
<td>Coupling</td>
</tr>
<tr>
<td></td>
<td>CMC (Coupling on Method Call)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CFA (Coupling on Field Access)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CAE (Coupling on Advice Execution)</td>
<td></td>
</tr>
<tr>
<td>Sant’Anna, Garcia, Chavez, Lucena, Staa [San, 03]</td>
<td>LCOO (Lack of COhesion in Operations)</td>
<td>Cohesion</td>
</tr>
</tbody>
</table>

**2.7 Analysis of various AOP Metrics**

AOSD has increasingly adopted technique in software development, but these are very few metrics for AOP to prove its low complexity. Another important issue of measuring the metrics is a cognitive aspect of programming. Researchers concentrated on the Cognitive complexity of AOP was limited. Hence, existing metrics are not
appropriate for determining aspect complexity. The following aspects of metrics are to be developed for Aspect-Oriented software in future is tabulated in Table 2.4.

Table 2.4 Analysis of various AOP Metrics

<table>
<thead>
<tr>
<th>Component Metric</th>
<th>Software quality metrics</th>
<th>Cross-cutting metrics</th>
<th>Coupling metrics</th>
<th>Cohesion metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFC (Understandability, Testability)</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WMC (performance, maintainability, complexity)</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FCD</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ACD</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HQ</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PHQ</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CAE</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>CFA</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>CIM</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>CMC</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>LCOO (reusability, efficiency)</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Cognitive Complexity</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
2.8 Conclusion

This chapter presents various metrics of Aspect-Oriented Software Development using their software quality metrics, crosscutting metrics, coupling metrics and cohesion metrics. Even though the Aspect-Oriented software development is increasingly being adopted for software development, measuring the cognitive complexity of the metric is still a difficult task. By using difficult metrics the aspect understandability, testability and complexity can be designed. There is a need for complexity metric that can measure the time with all the aspects of the software. Ultimately, this literature survey has motivated to propose a cognitive complexity metrics suite to reduce the problems found in the existing metrics and significantly enhances the quality of software in Aspect-Oriented (AO) system. This research work concentrates on cognitive complexity metrics for AO features such as Advice, Joinpoint, Pointcut and Reference Attribute. The following chapter describes the cognitive complexity metrics suite Framework for Aspect Oriented Programming.