Chapter - 1

Introduction
1.1 Overview

Software development is a highly complex and intellect-oriented activity. In the early days of program development, the developers wrote programs using machine language in which they spent more time on machine's instruction set than the problem at hand. Gradually, programmers migrated to high-level languages like C, C++, Java. There are different approaches are being practiced for developing error-free and cost-effective software. In the mid-seventies, Module-Oriented Approach (MOA) was the dominant approach for software development. Modular programming is a programming paradigm based upon the concept of the procedure (also known as subroutine, method and function) call, where 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, involving moderate complexity and easy maintainability. Some of the 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 and

(iv) No proper binding of data with operations.
Nowadays, OOA has become a mainstream programming paradigm where real-world problems are decomposed into objects that abstract behaviour and data in a single unit. OOA aimed at removing some of the flaws encountered with MOA. Object-Oriented Programming (OOP) encourages software reuse by providing design and language constructs for modularity, encapsulation, inheritance and polymorphism. Some of the important features/limitations of OOA include:

(i) Emphasis on data rather than procedure.
(ii) Allows viewing a system as a set of interacting objects.
(iii) Divides the 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.

Although OOP/OOA has met great success in modelling and implementing complex software systems, it has its limitations. Practical experiences with large projects have shown that developers may face problems with maintaining their code. 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 [Ale, 14]. The techniques of AOSD make it possible to modularize crosscutting concerns of a system. Like, objects in Object-Oriented Software Development (OOSD), aspects in AOSD may arise at any
stage of the software life cycle, including requirements specification, design, implementation, etc. The crosscutting concerns like exception handling, synchronization, logging, and resource sharing are some of the examples. Exhaustive research and development work has been carried out on software metrics for module oriented systems and object-oriented systems. However, from the literature, it is observed that sufficient research has not been carried out on measuring the characteristics and metrics for the emerging Aspect-Oriented Approach (AOA).

1.2 Definition of the Problem

AOP is not a standalone programming paradigm like object oriented programming and imperative programming. AOP is always used together with other programming paradigms. It can best be described in terms of concerns, joinpoints and aspects, where 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.

Such concerns can be based both on functional and non-functional requirements. Examples are logging, security, caching and buffering. Joinpoints are the locations affected by one or more crosscutting concerns.

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 a standard Java byte code, thus it keeps Java’s advantages.

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. Aspects are fundamental to the definition of AOP. Aspects are design decisions that they are difficult to address in regular OO-code because they crosscut the system. We can move the updating of display to a separate subprogram called an aspect.

Aspects are AspectJ’s unit of modularization, the same way classes are in Java. An aspect contains pointcuts and advices, similar to Java classes, which have fields and methods. It can be extended like Java classes. 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 as an abstract methods. 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.
1.3 Aim and Objectives

The overall aim of the research work is to propose a Cognitive Complexity Metrics Suite (CCMS) for AOP. The sub objectives in achieving this are:

- To propose metrics based on cognitive complexity.
- To develop a tool to collect and analyze the proposed CCMS.
- To validate the proposed CCMS theoretically and empirically.
- To compare the cognitive complexity metrics with existing metrics.
- To propose a software maintenance effort prediction model using cognitive complexity metrics.

1.4 Scope of Research Work

Software metrics, as mentioned earlier, can be classified into three categories: product metrics, process metrics, and project metrics. The scope of this thesis is restricted to product metrics that deals with the structural aspects of the program. The product metrics can be either static metrics or dynamic metrics that measures the quality of executing the program. The focus is on static metrics dealing with the architectural complexity. There are mainly two groups in static metrics, namely, design metrics, and size metrics. Out of these two groups, design metrics at the class level is zeroed in for the research study. The product design metrics at the static level deals with the structural and cognitive aspects of the program. The diagrammatic representation of the scope of this thesis is given in Figure 1.1.
As the size and complexity of the program increased, industries started migrating to the new AO paradigm due to various valuable reasons as mentioned earlier, apart from reducing the sheer volume of management, reuse of standard modules, and cost reduction. This paradigm shift is the reason behind limiting the scope to the AO software and especially the fully object oriented, most popular and widely used Java language, although it can be extended and applied to other languages with their own bindings as Abreu et al have done for C++ and Eiffel [Abr, 95] [Abr, 96].

The advent of AO paradigm, though it eased the product management, has also increased the cognitive load of the developer due to the raise of the structural complexity of the software. The term “Software Complexity” refers to the difficulty...

![Figure 1.1 Overview of Metrics and Scope of the Thesis](image-url)
to understand, change and maintain the software [Wan, 03] [Wan, 09]. So, there is a need for introducing new measures to capture this cognitive based complexity. Several researchers have proposed various metrics for AO systems. But, except a few, as mentioned in section 1.2, most of these researches do not take into account the cognitive complexity that reflects the cognitive load in AOP, which plays a vital role in measuring the software complexity. Knowing the importance of this fact, the scope of this research work is to propose cognitive complexity metrics suite consisting of the important AOP features, which are not already considered and proposed, like advice, joinpoint, pointcut and reference attribute.

Among the various software quality attributes like reliability, correctness, usability, integrity, efficiency, testability, portability, maintainability, flexibility, reusability, interoperability, understandability, etc., [Fen, 14] maintenance and understandability are the most important and key qualities that are desired in the industry due to overall reduction in developing cost, time and effort [Som, 04]. Recently, the majority of the software metrics are developed for these two most important attributes [San, 12]. The maintenance cost, after the release of the software product, is as high as 60% to 80% of the total cost of the software [Ogh, 14]. The customer demand and evolutionary changes of the software inevitably call for such maintenance effort [Ale, 16]. In fact, it is the key for the survival of the product in the highly competitive market and to bounce back successfully as it faces many challenges from the constantly changing hardware and software environments. Hence, the empirical study is limited to maintenance.
1.5 Methodology

The proposed metrics suite consists of three metrics, namely, Cognitive Weighted Method per Class (CWMC), Cognitive Weighted Coupling on Advice Execution (CWCAE), Cognitive Weighted Pointcut per Aspect (CWPA), and Cognitive Weighted Coupling on Attribute Reference (CWCoAR) which are used to measure the cognitive complexity arising due to advice, joinpoint, pointcut and attribute reference respectively. These AO features play a vital role in determining the quality of design in AO paradigm. The CWMC is proposed to measure the complexity of the advice which is one of the feature of an AO system.

The CWCAE is proposed to measure the complexity of different types of joinpoint. Coupling on Advice Execution (CAE) is the number of joinpoint which is a well-defined point in the execution of a program, such as a call to a method, access to an attribute, an object initialization, an exception handler etc.

The CWCoAR is proposed to measure the coupling complexity of the attribute references arising due to Static Reference Variable (SRV), Dynamic Reference Variable (DRV), and Inheritance Reference Variable (IRV). The CWCoAR also includes relative CWs that are needed to comprehend these types of attribute references.

1.6 Research Contributions

In this research work, an Analysis of Metrics for Aspect-Oriented Software Development (AOSD) has been presented. In this research, four AOP metrics are proposed that suite their capability to predict maintenance effort with the help of metric measures. The proposed metrics suite for the AOSD is empirically analysed.
Chapter – 1                   Introduction

Cognitive Complexity Metric Suite for Aspect Oriented Programming (CCMSAOP)

Contribution 1: Cognitive Weighted Method per Class

The proposed metric, Weighted Methods per Class (WMC) is to measure the Complexity of Advice such as before, after, and around. The Metric takes cognitive weights into consideration, and data collection satisfies the Fenton et al. [Fen, 97] properties.

Contribution 2: Cognitive Weighted Coupling on Advice Execution

The proposed metric called Cognitive Weighted Coupling on Advice Execution (CWCAE), reveals the cognitive complexity of the different types of joinpoint such as object initialization join points, exception handler join points, call join points, and advice execution join points. CWCAE metric takes cognitive weights into consideration, and data collection satisfies the Fenton et al. [Fen, 97] properties.

Contribution 3: Cognitive Weighted Pointcut per Aspect

The proposed metric called Cognitive Weighted Pointcut per Aspect (CWPA), which is adding Cognitive Weighted Pointcut Designator (CWPD) and Cognitive Weighted Method per Class (CWMC) used in an aspect. This is a better indicator than the existing Weighted Pointcut per Aspect (WPA). The weight of each type of pointcut is calculated by using cognitive weights and weighting factor similar to which is suggested by Wang et al. [Wan, 03].

Contribution 4: Cognitive Weighted Coupling on Attribute Reference

The Cognitive Weighted Coupling on Attribute Reference (CWCoAR) novel complexity metric augments the cognitive complexity based on the different types of attribute allusion to the components. The range of attribute allusion is implemented statically by inheritance or dynamically.
**Contribution 5: Cognitive Complexity Metric Analysis Tool - Aspect Oriented Programming**

The proposed Cognitive Complexity Metric Analysis Tool (CCMAT) is implemented using the Integrated Development Environment (IDE) called “Eclipse Luna”. The Java parser is developed to parse the given Java project file to collect the metrics information. A new IDE is developed to integrate and deploy our built-in framework. The CCMAT is used to carry out a comparison between Aspect Oriented Programming (AOP) metrics and AOP Cognitive Complexity Metrics. AOP metrics are used since they have been widely accepted by the researchers as standard metrics for the Aspect-Oriented system. This tool is used to prove the effectiveness of the proposed Cognitive Complexity Metric Suite.

**Contribution 6: Empirical Validation of CCMSAOP using Maintenance Effort Prediction**

This study has been conducted to find the relationship between design complexity and maintenance time. It also proposes a model to predict the maintenance effort. There are numerous ways to assess the relationship between two variables. Some of them are (Analysis of Variance) ANOVA, correlation and regression. They can be used to observe whether each complexity metric is a reliable indicator of expected maintenance time. The complexities of both the treatments, with each of the four metrics, CWMC, CWCAE, CWPA and CWCoAR are measured. Tests are conducted to validate the proposed metrics CWMC, CWCAE, CWPA and CWCoAR empirically, the primary objective of the experiment.
The proposed metrics CWMC, CWCAE, CWPA and CWCoAR are theoretically validated through the Fenton’s properties [San, 08] [Fen 97]. From the validation, it is found that all the proposed metrics satisfy Fenton’s Data Collection properties. Further, it is experimentally validated using a test case to show how the Cognitive Complexity metrics are computed for an AspectJ program. The proposed metrics are empirically validated to prove that these metrics are better indicators of complexity than the existing metrics. Empirical validation is carried out for CCMS using maintenance effort. Cognitive Complexity Metrics Analysis Tool (CCMAT) has been developed using Java Programming language. This tool is used to collect and analyze the cognitive complexity metrics, namely, CWMC, CWCAE, CWPA and CWCoAR.

1.7 Organisation of Thesis

In this thesis, a set of Aspect-Oriented software metrics are used. This set of metrics is targeted towards predicting the maintenance efforts required to test Aspect-Oriented programs. The results are justified based on analytical approaches, rather than compared. The thesis has been organized as follows: Chapter 1 gives the introduction and overall view of the research study. Chapter 2 is devoted to literature survey. Chapter 3 is dedicated to Cognitive Complexity Metrics Suite (CCMS) Framework. Following these chapters, Cognitive Weighted Method per Class (CWMCC) Metric is discussed in chapter 4. Cognitive Weighted Coupling on Advice Execution (CWCAE) metric is elaborately presented in chapter 5. Cognitive Weighted Pointcut per Aspect (CWPA) metric is focussed in chapter 6. In Chapter 7, Cognitive Weighted Coupling on Attribute Reference (CWCoAR) metric is discussed. Cognitive Complexity Metric Analysis Tool - Aspect Oriented Programming (CCMAT-AOP) is
presented in Chapter 8. Empirical Validation of CCMS using Maintenance Effort Prediction is investigated in chapter 9. The last chapter presents the conclusion of the work and the future research directions.

1.8 Conclusion

This chapter provides the basis for the research and justifies the need of this research work. It consists of basic information regarding software metrics and emphasizes the importance for the AOP development paradigm. It explains the need for complexity metrics in software measurement. The CCMS provides the real complexity of software including the cognitive load experienced by software engineers to understand the source code of the software. This chapter highlights the need for the migration of complexity metrics into the CCMS, which is defined in this chapter. Various objectives of the research that are achieved as a result of the research work is also presented. The presentation includes the boundary within which the current research work is conducted. It describes CCMS framework for defining and validating the proposed research work. Finally, this chapter is concluded with the chapter organization which briefly discusses the purpose of each chapter in the thesis. The next chapter reviews the literature and previous research findings. It provides the motivation for the proposed research work.