Cohesion and coupling measures for aspect oriented systems

Ms. Puneet Jai Kaur¹*, Ms. Sakshi Kaushal ²

¹Assistant professor, UIET, Panjab University, Chandigarh
²Reader, UIET, Panjab University, Chandigarh

Abstract

Software metrics aim to measure the inherent complexity of software systems with a view toward predicting the overall project cost and evaluating the quality and effectiveness of the design. Aspect Oriented Software design is an emerging paradigm that provides new mechanisms to support the modularization of concerns, which otherwise would crosscut the OO system decomposition. There are various metrics to measure quality characteristics of object oriented system but very few metrics are for aspect oriented systems. Various Quality Attributes are cohesion, coupling, complexity, size, reusability, changeability, maintainability etc. In this paper we provide the review of quality metrics designed for AOSD depending on the different attributes of quality.

Keywords: AOSD, metrics, complexity, cohesion, coupling

1. Introduction

Software engineering is related to the development and evolution of large, complex and critical software intensive system. These systems are expected to be more flexible, scalable and reusable. In order to achieve these objectives, development techniques that support abstraction and modularization in Software development system can be useful. Aspect Oriented Software Development (AOSD) and other new types of modularity and abstraction approaches are attracting lot of attention over many domains within and beyond computer science. AOSD is comparatively a modern programming paradigm aimed at improving modularity under the umbrella of aspects. It has been frequently claims that applying an AOSD method will eventually lead to quality software. In order to verify the claims concerning the maintainability, reusability and reliability of the systems developed using aspect oriented techniques, software measures are required. Aspect oriented structural quality must be related to some critical process attributes concerning maintainability, reusability and reliability.

Cohesion is a structural attribute whose importance is well recognized in the software engineering community [3]. Cohesion is an internal software attribute representing the degree to which the components are bound together within a software module. Coupling is the software attribute that defines the degree of interdependence between two classes or aspects. It is one of the few attributes that has been shown to have large impact on external software quality characteristics like maintainability, modularity, reusability, testability and usability. A system should have high cohesion for better quality. The remainder of this paper is organized as follows: section 2 represents the introduction to AOSD, section 3 gives the coupling and cohesion measures for AOSD and section 4 gives the concluding remarks.

2. Aspect oriented software development

Aspect oriented software development (AOSD) is a relatively new software development paradigm that complements and improves on many contemporary development paradigms. Traditional software development focuses on decomposing systems into units of primary functionality, while recognizing that there are other issues of concern that do not fit well into the primary
decomposition. The traditional development process leaves it to the programmers to code modules corresponding to the primary functionality and to make sure that all other issues of concern are addressed in the code wherever appropriate [15] [1]. Programmers need to keep in mind all the things that need to be done, how to deal with each issue, the problems associated with the possible interactions, and the execution of the right behavior at the right time. These concerns span multiple primary functional units within the application, and often result in serious problems faced during application development and maintenance. The distribution of the code for realizing a concern becomes especially critical as the requirements for that concern evolve — a system maintainer must find and correctly update a variety of situations. Aspect-Oriented Software Development focuses on the identification, specification and representation of cross-cutting concerns and their modularization into separate functional units as well as their automated composition into a working system.

Aspect oriented programming (AOP) considers acknowledged improvements to separation of concerns provided by previous technologies (mainly object oriented programming), while supporting new mechanisms to deal with crosscutting concerns, that is, special concerns that are not properly modularized by these technologies. AOP introduces aspects as a new composition mechanism for weaving aspects back into components at well defined join points.

2.1. Basic concepts in AOSD

2.1.1. Separation of Concern:

Concerns are the different aspects of software functionality. For instance, the "business logic" of software is a concern, and the interface through which a person uses this logic is another. The separation of concerns is keeping the code for each of these concerns separate. Changing the interface should not require changing the business logic code, and vice versa. The principle of separation of concerns essentially states that each concern that is relevant to the software application is best treated separately from the other concerns. This principle is not only valid in a software engineering context. To better understand this principle, let us first consider it in the context of an architect’s work in the design of a building [16] [4].

When designing a building, architects do not make one single plan that describes the overall structure of the entire building. Instead, they use many different plans that each focuses on a single part of the building: front and side views, floor plans, cross-sections, foundation, drainage system, electrical wiring, central heating, and so on. Each of these plans addresses a single concern of the building. They are separated because they are supposed to be used by different persons: clients, bricklayers, electricians, plumbers, and so on. Their separation makes them easier to understand and facilitates the modification of each single concern of the building.

2.1.2. Concerns

The IEEE defines the concern as, those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. In the above examples each single plan i.e. plan for front and side view, Floor plans, cross sections etc are the single concerns. The principle of SOC states that each of the concern must be considered in isolation throughout the entire software development lifecycle.

2.1.3. Crosscutting Concerns

Crosscutting is defined as a phenomenon that is observed whenever two properties being programmed must compose differently and yet to be coordinated. The majority of applications you design will contain common functionality that spans layers. This functionality typically supports operations such authentication, authorization, caching, communication, exception management, logging, and validation. Such functionality is generally described as crosscutting concerns because it affects the entire application, and should be centralized in one location in the code where possible. For example, if code that generates log entries and writes to the application logs is scattered throughout your layers, and the requirements related to these concerns change (such as logging to a different location), we may have to update the relevant code throughout the entire system. Instead, if we centralize the logging code, you can change the behavior by changing the code in just one location. Crosscutting concerns are problematic because their implementations are tangled with other concerns and/or scattered throughout the entire application. Aspect Oriented software development explicitly represents crosscutting concerns as separate entities, solving many of the associated evolution and reusability problems.

2.1.4. Aspect

To modularize the crosscutting concerns, software developers need a different decomposition technique. Modules in contemporary programming languages are all based on some form of functional decomposition (e.g. subroutines, functions, objects etc). Aspect oriented
proposes a fundamentally new kind of modularization that goes beyond functional decomposition: an aspect. Aspect is a feature that is linked to many parts of a program, but which is not necessarily the primary function of the program. Aspect is the term used to denote the abstraction that aims to support improved isolation of crosscutting concerns. Aspects are modular units of crosscutting concerns that are associated with a set of classes or objects. An aspect can affect, or crosscut, one or more classes and/or objects in different ways. Aspect-oriented system designs are decomposed into classes and aspects; aspects modularize crosscutting concerns and classes modularize non-crosscutting concerns. An aspect’s implementation consequently consists of two conceptually different parts: the aspect functionality code and the aspect applicability code. The aspect’s functionality code is not essentially different from ‘regular’ code and is executed when the aspect is invoked. This invocation of the aspect is determined by the aspect applicability code. This code contains statements that specify where or when the aspect needs to be invoked. In standard AOSD terminology, this aspect applicability code is referred to as a pointcut expression and the aspect functionality code is referred to as the aspect’s advice code. Since a single aspect can consist of multiple different functionalities that need to be invoked from various different places in the code, an aspect’s implementation can consist of several pointcuts and advice code segments.

2.1.5. Join points and Point cuts
In all aspect-oriented programming languages, aspects can only be invoked at some well defined and principled points in the program’s execution. These points are referred to as join points. A join point is a point of interest in some artifact in the software lifecycle through which two or more concerns may be decomposed. Possible kinds of join points in the execution of a program are, for example, assignment statements, method or constructor calls, variable references, etc. Since an aspect’s behavior can only be invoked at a join point, a pointcut describes the set of join points where the aspect’s advice needs to be invoked. A pointcut is a predicate that matches join points. A pointcut is a relationship ‘join point -> boolean’, where the domain of the relationship is all possible join points. A pointcut is indeed often expressed using a set of dedicated predicates that reason about the entire possible set of join points in a software application to determine the correct set of join points for a specific aspect.

2.1.6. Advice
Advices are the executable part of the Aspects. It defines what code to run when a join point is fired. It exist three basic kinds of advices; before, after and around. They all works as a method call with some limitations. The before and after advices can’t return anything and around must return the declared type. Before and after advices are executed as it sounds before or after for example methods calls and around replace the method calls.

2.1.7. Weavers
Aspect languages rely on a specific kind of compilers (or interpreters), called weavers, that compose the aspects’ implementation with the other modules. Although one could argue that weavers are simply compilers for aspect languages and that many weavers are currently implemented as source-to-source or byte code transformations.

3. Quality metrics for AOSD
There are various models to evaluate the quality of object-oriented and module-oriented approaches. Various Quality model for legacy system are given by Boehm, McCall, Dromey, Ward, ISO/IEC-9126, etc. But very less work is done to evaluate the quality characteristics of aspect-oriented system. Various quality characteristic are cohesion, coupling, complexity, size, reusability, changeability, maintainability etc.

Cohesion is degree of relatedness among elements of the modules. High cohesion is desirable.
Coupling is degree of relatedness among the modules that is connection between modules. Low coupling is desirable.
Complexity is efforts needed to analyze code, efforts needed in modification and modification of modules.
Maintainability is modification of software product after delivery.
Reusability is using the module again to reduce the coding.

There are various software metrics for these characteristics for legacy systems. But very few metrics are discussed for aspect-oriented system. Software metrics act as indicator of quality of a system i.e. provide quantitative basis. Only few papers are addressing the concept of cohesion in existing literature. We will discuss various software metrics for aspect-oriented system in the following sections. Software quality metrics for AOSD are categorized based on the various quality characteristics.

3.1. Metrics for measuring cohesion

Cohesion is one of the important quality attribute. High cohesion is a desirable property of software components. Cohesion is an underlying goal to continually be considered during the design process. It is widely recognized that highly cohesive components tend to have high maintainability and reusability. Cohesion refers to the degree of relatedness between component members. Aspect Oriented language introduces the new kind of component called aspect to model the crosscutting concerns in a software system. An Aspect with its encapsulation of state (attributes) and associated modules (operations) such as advice, introduction, point cuts and methods is a different abstraction in comparison to a procedure within procedure oriented and a class within object oriented systems. The cohesion of Aspect is therefore mainly about how tightly the aspect attributes and modules cohere.
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 [6] [8] to measure cohesion in AO systems and most of the measures are for AspectJ programming language. Since then, many other languages, such as CaesarJ, HyperJ have also been designed and used in various applications.

Anna et. al proposed [2] a new metric Lack of Cohesion in Operations (LCOO), which measures the amount of method/advice pairs that do not access the same instance variables. It is an extension of the well-known Lack of Cohesion Methods (LCOM) metric developed by C&K. This metric measures the lack of cohesion of a component (class and aspect). The author proposed that the high LCOO value indicates disparateness in the functionality provided by the aspect. The definition of LCOO metric is almost operational.

Anna et. al also [2] 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 a very useful framework for evaluating internal software quality attributes such as cohesion, coupling, complexity and modularity at architecture level. But it has not been specified how to quantify aspect cohesion for empirical evaluation. It is also not clear whether inheritance has been accounted or not.

Zhao and Xu [9] propose a framework for assessing aspect cohesion, based on the analysis of dependencies. This framework is based on the aspect dependence graph, and it analyzes the degree of coherence between aspects attributes and modules (advice, introductions, pointcuts and methods). The approach has focused on the features of the aspect itself, and does not consider the application context in which the aspect is placed. The measurements proposed by this approach are quite complex and the choice of the metric weights is ad hoc.

Gelinas, Badri, proposed metric ACoH [8] 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.

Kumar et. al [13] 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 the work by Gelinas et. al [13] and defined Unified Aspect Cohesion as UACoh. Further, they extended their work [13] 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.

3.2. Metrics for Measuring Coupling

In an aspect-oriented (AO) system, the basic units are aspects and classes. An aspect with its encapsulation of state (attributes) with associated modules such as advice, intertype declarations, pointcuts, and methods (operations) is a significantly different abstraction in comparison to the class within object-oriented systems. An aspect may also interact with one or more classes in many ways through advice, inter-type declaration, pointcut, and method call. Thus, in AO systems, the coupling is mainly about the degree of interdependence among aspects and classes. However, although coupling has been widely studied for object-oriented systems, it has not been studied for AO systems yet. Moreover, existing approaches to measuring the coupling of object-oriented systems can not be directly applied to aspect-oriented systems since an AO system contains some new types of interactions between aspects and/or classes that may have great impact on the coupling of the system.

Zhao in his paper [10] had proposed a coupling measure suite for assessing the coupling in aspect-oriented systems. Author first presented a coupling framework for AO systems which specially designed to count the dependencies between aspects and classes in the systems. Based on this framework, the author had formally defined various coupling measures in terms of different types of dependencies between aspects and classes. He also discussed the mathematical properties of these measures. Cecatto and Tonello [7] had proposed the coupling metrics based on C & K metric suite. These are:

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 point cuts 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 point cuts and by the introductions in a given aspect.
4. Conclusion

AOSD is motivated by rigorous quality factors, such as maintainability, reusability, cohesion, coupling etc. In this paper we have given the metrics for measuring the various quality attributes for AOSD. Emphasis has been given on cohesion and coupling because the principle for a good quality software is “Low Coupling and High Cohesion”. In future we will be concentrating on other major quality issues like reliability, security, performance etc and their impact on quality of AOSD.

References