CHAPTER I

INTRODUCTION

1.1 Motivation

Software quality has become increasingly important to the world of software development. Although it is not a major requirement in developing software, it is a desirable requirement which helps developers to decrease the development cost, to ensure on-time project delivery, and to increase the user satisfaction. Software quality factors consist of *functionality*, *reliability*, *efficiency*, *maintainability*, *portability*, and *usability* [1]. From these factors, maintainability is a significant factor that developers should concern because two-thirds of the software cost involves maintenance [2].

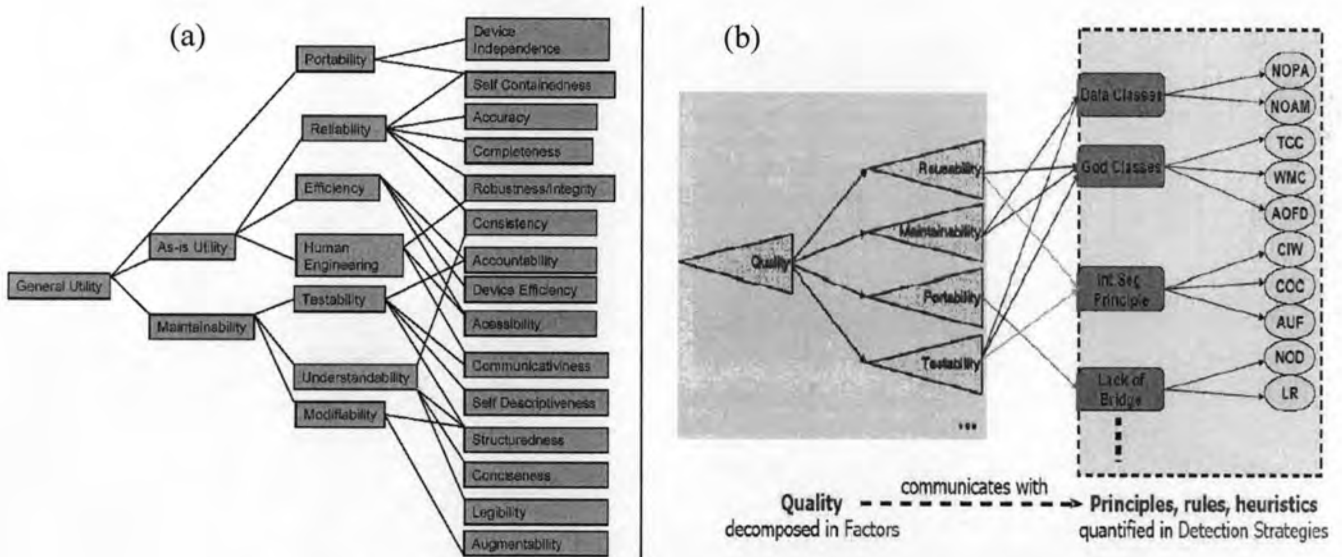

Figure 1.1: FCM (a) and FS (b) quality models [3, 4]

To support controlling the software quality, suitable measurement models or metrics are required to quantitatively evaluate the software. *Factor-Criteria-Metric (FCM)* quality models of McCall [5] and Boehm [3] are most popular models that are widely used to extract the relevant attributes from an expected quality factor. FCMs are constructed in tree-like structures (figure 1.1 (a)), which their upper branches hold high-level *quality factors*, such as reliability and maintainability. Each quality factor is then

broken down into low-level *criteria*; for instance, completeness and consistency. Finally, these criteria are linked to the correlated *metrics*.

Even though FCMs are well-known quality models, there are two main drawbacks of their usability that are mapping obscure from quality criteria to metrics and poor capacity to link quality flaws to their causes. To solve these limitations, Marinescu and Ratiu define an approach that has major improvements over the traditional approach, called *Factor-Strategy (FS)* quality model [4]. Like FCMs, FS model employs a decompositional approach (figure 1.1 (b)). Nevertheless, it associates quality factors with *strategies*: *principles*, *rules*, and *heuristics*, instead of criteria before linking these strategies to metrics.

Aspect-oriented programming (AOP) is a programming paradigm that firstly emerges out as an augmentation of object-oriented programming and aims to promote good design. It attempts to solve code tangling and code scattering problems by modularizing and encapsulating crosscutting concerns [6]. To encapsulate diverse types of concerns, AOP introduces *aspects*.

Measuring object-oriented software maintainability has been gotten enough support from the amount of researches. Jindasawat and her colleagues propose a maintainability model for object-oriented software from investigating the correlation between object-oriented design metrics and the exam score [7]. Genero et al. use the controlled experiment to choose class diagram structural complexity and size metrics for evaluating understandability and modifiability of object-oriented software [8]. Sheldon and his colleagues define a set of metrics for assessing object-oriented software maintainability based on class inheritance hierarchies [9]. Sant' Anna et al. propose a framework for assessing aspect-oriented software reusability and maintainability based on empirical and quantitative analysis [10]. Bartsch and Harrison explore the effect of aspect-oriented programming on maintainability. They compare understandability and modifiability of an object-oriented program with an aspect-oriented program using software professionals and a set of questionnaires [11]. Only few researches are done to support measuring aspect-oriented software maintainability. In addition, measuring maintainability in these studies is analyzed and is combined from two sub-characteristics: *understandability* and *modifiability* of FCM quality models [3].

This work aims to apply a set of aspect-oriented design principles and object-oriented design principles to evaluate aspect-oriented software maintainability. Following the FS quality model, object-oriented design heuristics, bad smells for object-oriented refactorings, and bad smells for aspect-oriented refactorings are filtered and are combined with a set of proposed aspect-oriented design guidelines to construct aspect-oriented software maintainability metrics. The object-oriented design principles are brought because the main structure of aspect-oriented system is the object-oriented structure. Design principle violation check definitions are defined in the form of Boolean expressions to facilitate measuring maintainability metrics. These definitions are also helpful in detecting specified design flaws independently. The aspect-oriented software maintainability metrics can be used to evaluate maintainability both systems written in AspectJ and Java.

1.2 Objectives

Objectives of this work are as follows:

- To extract or to define aspect-oriented design guidelines.
- To define design principle violation check definitions for guidelines, design heuristics, and bad smells.
- To define a set of metrics for evaluating aspect-oriented software maintainability.

1.3 Scope and Limitations

1. This work will extract or will define at least eight aspect-oriented design guidelines for maintainability classified in four groups that are *the design guidelines for pointcuts and advices*, *the design guidelines for aspects*, *the design guidelines for the relationships between aspects and other components*, and *the design guidelines for inheritance relationships*.
2. This work will propose aspect-oriented software maintainability metrics in three measurement levels: the *unit* level, the *system* level, and the *system*

comparison level based on the Factor-Strategy (FS) quality model and the Factor-Criteria-Metric (FCM) quality models.

3. Design principles are collected from four sources which are *object-oriented design heuristics*, *bad smells for object-oriented system refactorings*, *bad smells for aspect-oriented system refactorings*, and *proposed aspect-oriented design guidelines*.
4. The violation check definitions and the aspect-oriented software maintainability metrics are validated using a set of aspect-oriented software samples and object-oriented software samples.

1.4 Contributions

The outcomes of this work are definitions for detecting design principle violations, supportive guidelines for aspect-oriented software design, and the metric suite for evaluating aspect-oriented software maintainability.

1.5 Research Methodology

1. Survey literature and review related researches about aspect-oriented programming, software measurement, and maintainability.
2. Study aspect-oriented refactorings and bad smells, object-oriented design heuristics, good or bad practices in aspect-oriented programming, and AspectJ.
3. Gather aspect-oriented software samples and object-oriented software samples.
4. Extract or define aspect-oriented design guidelines.
5. Filter design principles (object-oriented design heuristics, bad smells for refactoring object-oriented software, bad smells for refactoring aspect-oriented software) and gather them with aspect-oriented design guidelines.
6. Define violation check definitions for the filtered design principles and aspect-oriented design guidelines.
7. Explore the relationships between remaining principles and maintainability

8. Define maintainability metrics for aspect-oriented software.
9. Estimate the average range of maintainability using fifty software samples.
10. Apply the maintainability metrics with case studies.
11. Make conclusion and write thesis.

1.6 Organization of the Thesis

The remainder of the thesis is organized as follows:

Chapter II describes theoretical backgrounds and literature reviews including *aspect-oriented software development*, *AspectJ*, *object-oriented design principles*, *aspect-oriented design principles*, *maintainability*, *aspect-oriented design principles*, *aspect-oriented metrics*, and *weight values for object-oriented relations*.

Chapter III illustrates an overview of the thesis. It describes all collaborating components and links those components into the big picture. This chapter also presents the aspect-oriented guidelines which are extracted and are defined to fulfill the aspect-oriented software maintainability measurement. They support both *high level design* (e.g. logical functions) and *detailed design* (e.g. pseudocode). Fifteen guidelines are classified into 4 categories: *the design guidelines for pointcuts and advices*, *the design guidelines for aspects*, *the design guidelines for the relationships between aspects and other components*, and *the design guidelines for inheritance relationships*. In addition, this chapter describes a set of collected and filtered design principles.

Chapter IV defines design principle violation check definitions and their validations. These concrete definitions can be applied to construct an automatic tool for detecting each design flaw separately or can be combined to construct aspect-oriented software maintainability metrics in Chapter VII.

Chapter V validates design principle violation check definitions. All violation check definitions are confirmed their correctness by using them to detect design principle violations in code samples written from the original examples and the quotations of the principle sources.

Chapter VI applies the design principle violation check definitions to a set of aspect-oriented systems. Chi-Square test for estimating the percentage of repeated

หอสมุดกลาง ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

design principle violations is conducted as well. The test reveals that design flaws occur repeatedly in each system. This chapter also describes points of concern for violative-system adjustments that are *variations in metric values*, *side-effects of alterations*, and *changes in the system behavior*.

Chapter VII describes processes of defining aspect-oriented software maintainability metrics. Firstly, assumptions about the effects of design principles on a set of maintainability-related metrics are set. Secondly, those metrics are measured automatically from two sets of systems (systems violating design principles and systems obeying design principles) and are evaluated their changes in metric results. Finally, the qualified design principles with positive results are selected to construct aspect-oriented software maintainability metrics. The proposed metrics are specified in three levels that are the *unit level*, the *system level*, and the *system comparison* level.

Chapter VIII shows applications of the metrics to evaluate fifty systems and to compare maintainability between systems written in AspectJ and in Java. The results show that the metrics can be applied to evaluate and to compare maintainability of both systems implemented by an aspect-oriented approach and an object-oriented approach.

Lastly, Chapter IX concludes the research work and presents some directions for the future work.