CHAPTER II

BACKGROUND AND LITERATURE REVIEWS

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*.

2.1 Aspect-Oriented Software Development and AspectJ

Aspect-oriented programming (AOP) paradigm, also called aspect-oriented software development (AOSD), attempts to solve code tangling and scattering problems by modularizing and encapsulating crosscutting concerns [6]. To encapsulate various types of concerns, AOP introduces a new unit called *aspect*. An aspect is a part of program that crosscuts the core concern, the base code (the non-aspect part of program), by applying advices over a quantification of join points, called *pointcut*. *Join points* are well-defined points in program flows which can be captured by the aspect along the program execution. Join points include method execution, instantiations of objects, and throwing exceptions. A *pointcut* is a set of join points. Whenever the program execution reaches one of join points described in the pointcut, a piece of code associated with the pointcut (called *advice*) is executed. This allows a programmer to describe where and when the additional code would be executed. An *Advice* is a method-like module that provides a way to express crosscutting actions when the corresponding join points are captured [12]. There are 3 kinds of advices: before, after, and around. A *before advice* is executed prior to join point execution. An *after advice* is executed after the execution of join point. An *around advice* surrounds the join point. It has abilities to bypass join point execution, to continue an original context, or to alter the context.

```
ce0  public class Point {
 s1    protected int x, y;
me2    public Point(int _x, int _y) {
 s3      x = _x;
 s4      y = _y;
       }
me5    public int getX() {
 s6      return x;
       }
me7    public int getY() {
 s8      return y;
       }
me9    public void setX(int _x) {
s10      x = _x;
       }
me11   public void setY(int _y) {
s12      y = _y;
       }
me13   public void printPosition() {
s14      System.out.println("Point at("+x+","+y+")");
       }
me15   public static void main(String[] args) {
s16      Point p = new Point(1,1);
s17      p.setX(2);
s18      p.setY(2);
       }
     }

ce19 class Shadow {
 s20   public static final int offset = 10;
 s21   public int x, y;

me22   Shadow(int x, int y) {
 s23     this.x = x;
 s24     this.y = y;
me25   public void printPosition() {
 s26     System.outprintln("Shadow at
                ("+x+","+y+")");
       }
     }
```

```
ase27 aspect PointShadowProtocol {
 s28   private int shadowCount = 0;
me29   public static int getShadowCount() {
 s30     return PointShodowProtocol.
                aspectOf().shadowCount;
       }
 s31   private Shadow Point.shadow;
me32   public static void associate(Point p, Shadow s){
 s33     p.shadow = s;
       }
me34   public static Shadow getShadow(Point p) {
 s35     return p.shadow;
       }
pe36   pointcut setting(int x, int y, Point p):
         args(x,y) && call(Point.new(int,int));
pe37   pointcut settingX(Point p): target(p) &&
         call(void Point.setX(int));
pe38   pointcut settingY(Point p): target(p) &&
         call(void Point.setY(int));
ae39   after(int x, int y) returning(Point p):
           setting(Point p) {
 s40     Shadow s = new Shadow(x,y);
 s41     associate(p,s);
 s42     shadowCount++;
       }
ae43   after(Point p): settingX(Point p) {
 s44     Shadow s = new getShadow(p);
 s45     s.x = p.getX() + Shadow.offset;
 s46     p.printPosition();
 s47     s.printPosition();
       }
ae48   after(Point p): settingY(Point p) {
 s49     Sahdow s = getShadow(p);
 s50     s.y = p.getY() + Shadow.offset;
 s51     p.printPosition();
 s52     s.printPosition();
     }
   }
```

Figure 2.1: An example of AspectJ program [13].

AspectJ is the first and the most popular aspect-oriented programming language [6]. It interacts with the base program using pointcuts, advices, and inter-type declarations. Join points of AspectJ include *method calls*, *method or constructor execution*, *field read or write access*, *exception handler execution*, *class or object initializations*, *object pre-initializations*, and *advice execution* [12]. An *inter-type declaration*, also called *introduction*, is a mechanism that allows an aspect to crosscut others in static way. Six possible changes through the inter-type declarations are *adding members* (methods, constructors, and fields) to types (classes and aspects), *adding concrete implementations* to interfaces, *declaring extend or implement* to types, *declaring precedence* to aspects, *declaring custom compilation errors or warnings*, and *converting exception types* from check to uncheck. Figure 2.1 shows an AspectJ program which contains two classes (Point and Shadow) and one aspect

(PointShadowProtocol). Aspect PointShadowProtocol stores Shadow objects in every Point objects.

2.2 Object-Oriented Design Heuristics, Bad Smells, Bug Patterns, and Design Patterns

Heuristic 2.9: Keep related data and behavior in one place

A violation of this heuristic will cause a developer to program by convention. That is, to accomplish some atomic system requirement, he or she will need to affect the state of the system in two or more areas. The two areas are actually of the same key abstraction and therefore should have captured in the same class. The designer should watch for objects that dig data out of other objects via some "get" operation. That type of activity implies that this heuristic is being violated. Consider a user of a stove class trying to preheat an oven for cooking. The user should only send the stove an are_you_preheated?() message. The oven can test if the actual temperature has reached the desired temperature, along with any other constraints concerning the preheating of ovens. A user who decides if the oven is preheated by asking the oven for its actual temperature, its desired temperature, the status of its gas valve, the status o fits pilot light, etc., is violating this heuristic. The oven owns the information of temperature and gas cooking apparatus; it should decide if the object is preheated. It is important to note the need for "get" methods (e.g., get_actualtemp(), get_desiredtemp(), get_valvestatus(), etc.) in order to implement the incorrect preheat method.

Figure 2.2: An example of the object-oriented design heuristic.

Design heuristics are guidelines to help developers make proper decisions; to identify and to encapsulate experience of experts; to imprecisely and informally guide good or bad practices; to provision knowledge and judgement [14]. Riel defines 61 objected-oriented design heuristics which are grouped into 8 topics: *heuristics for class and objects*, *heuristics for topologies of action-oriented versus object-oriented applications*, *heuristics for the relationships between classes and objects*, *heuristics for the inheritance relationship*, *heuristics for the multiple inheritance*, *heuristic for the association relationship*, *heuristic for class-specific data and behavior*, and *heuristics for physical object-oriented design*. All heuristics are listed in Appendix A. A design heuristic example is shown in Figure 2.2.

Refactorings are processes of changing the software structure to improve its quality. *Bad smells* are signs or warnings which are used to identify the candidate classes to be restructured [15]. Fowler introduces twenty-two bad smells for object-oriented software refactorings. The aspect-oriented refactorings and bad smells are also proposed by Monteiro and Fernandes [16], Piveta and his colleagues [17], Srivisut and Muenchaisri [18].

Bug patterns are erroneous code idioms or bad coding practices that have been proved fail time and time again. They are proposed by Zhang and Zhao to support testing and debugging AspectJ programs. Six bug patterns with their symptoms, cause root, cures and preventions are defined [19].

Design patterns are general repeatable solutions to common problems in software design. They are not ready-made designs that can be transformed directly into code, but they can be applied as templates to solve the similar problems in different situations. They are classified in 3 main categories: creational patterns, structural patterns, and behavioral patterns [20].

2.3 Maintainability

Some studies about measuring the software maintainability are described as follows:

Jindasawat et al. investigate the correlation between object-oriented design metrics and two maintainability sub-characteristics: *understandability* and *modifiability* using the controlled experiment [7]. Twelve structural design metrics for class diagram and six design metrics for sequence diagram are collected. Understandability and modifiability sub-characteristics are evaluated by the exam score. After performing some statistical tests, 14 of 18 metrics are passed to be maintainability indicators.

Genero and her colleagues try to find metrics for evaluating understandability and modifiability of UML class diagrams in design phase. They use controlled experiment to estimate understanding and modifying time, and then explore the relationships between class diagram structural complexity and size metrics and the measured time using multivariate linear model. The results reveal that 6 of 11 metrics affect the effort of maintaining UML class diagrams [8].

Sheldon and his colleagues extend the works of Chidamber and Kemerer [21] and Li [22] to define metrics for *understandability* and *modifiability* of class inheritance hierarchies. Class diagrams along with their inheritance relationships are translated into graphs. Then, two simple metrics are calculated based on the graphs. The proposed metrics are heuristically validated using a comparison of two designs. They claim that using their metrics adds an insight gain about trading off requirement conflicts: to

promote reuse via deep hierarchy or to ease the software maintenance via shallow hierarchy [23].

Sant' Anna et al. present a framework, which is based on a suite of metrics and a quality model, to assist the assessment of aspect-oriented software in terms of reusability and maintainability. Most of metrics rely on traditional metrics and extensions of OO metrics. The quality model is constructed based on empirical and quantitative analysis [10].

Bartsch and Harrison describe an exploratory assessment of the effect of aspect-oriented programming on software maintainability. An experiment was conducted in which 11 software professionals were asked to carry out maintenance tasks on one of two programs: written in Java and in AspectJ. A number of statistical hypotheses were tested. The results did seem to suggest a slight advantage for the subjects using the object-oriented system since in general it took the subjects less time to answer the questions on this system. However, the results did not show a statistically significant influence of aspect-oriented programming at the 5% level [11].

2.4 Aspect-Oriented Design Principles and Metrics

Wampler describes 6 aspect-oriented design principles. These principles are adapted from the traditional object-oriented design principles, but they are considered in an aspect-oriented perspective [24]. These principles are not aimed to support specific quality factors.

Zhao proposes a metric suite for assessing couplings, cohesion, and complexity in AO systems. For *couplings*, he presents a coupling framework for AO systems which is designed for counting dependencies between aspects and classes. Formal definitions of coupling metrics in terms of dependencies are also defined [25]. For *cohesion*, Zhao and Xu propose an approach to measure cohesion based on the dependence analysis. They discuss the tightness of aspects in 3 facets: inter-attribute, module-attribute, and inter-module. These facets can be applied to measure the aspect cohesion independently or can be integrated as a whole [26]. For *complexity*, Zhao proposes a set of metrics for measuring aspect-oriented software complexity. Based on

aspect-oriented software dependence models, metrics are defined from different system viewpoints and levels [27].

2.5 Weight Values for Class Diagram Relationships

Weight values for 10 types of class diagram's relationships, i.e., *dependency*, *common association*, *qualified association*, *association class*, *aggregation association*, *composition association*, *generalization (parent class is concrete)*, *binding*, *generalization (parent class is abstract)*, and *realization*, are defined in [28]. Kang and his colleagues use weighted class dependence graphs to analyze the given class diagram, and then measure a structural complexity of that diagram based on its entropy distance. They consider the complexity of both the classes and relationships between classes and present rules for transforming complexity value of the class and different kinds of relationships into a weighted dependence graph. These processes help the structural complexity of class diagram can be objectively measured.