การนิยามและการตรวจจับร่องรอยที่ไม่ดีของโปรแกรมเชิงแง่มุม

นายคมศัลล์ ศรีวิสุทธิ์

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

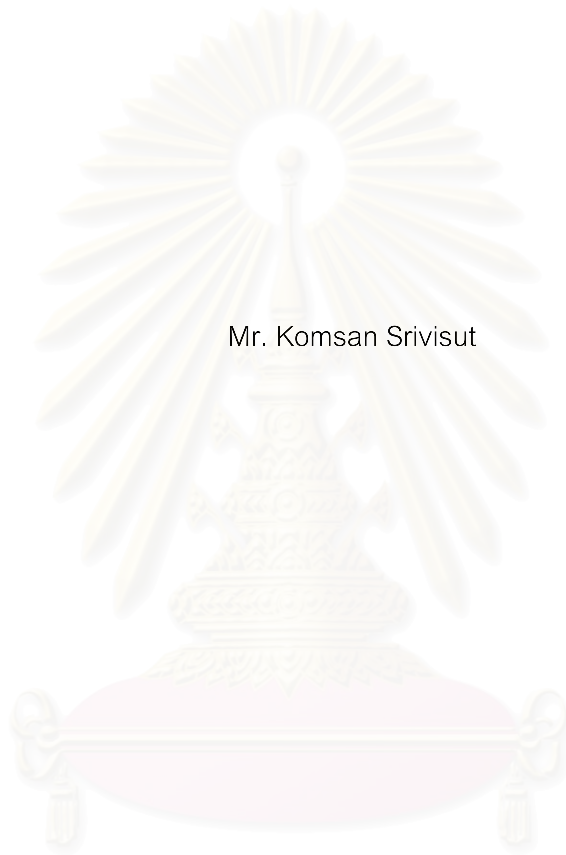
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2550

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

DEFINITION AND DETECTION OF BAD SMELLS OF ASPECT-ORIENTED PROGRAM



Mr. Komsan Srivisut

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2007

Copyright of Chulalongkorn University

Thesis Title	DEFINITION AND DETECTION OF BAD SMELLS OF ASPECT-ORIENTED PROGRAM
By	Mr. Komsan Srivisut
Field of Study	Computer Engineering
Thesis Advisor	Associate Professor Pornsiri Muenchaisri, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

.... Dean of the Faculty of Engineering

(Professor Direk Lavansiri, Ph.D.)

THESIS COMMITTEE

.... Chairman

(Assistant Professor Wiwat Vatanawood, Ph.D.)

.... Thesis Advisor

(Associate Professor Pornsiri Muenchaisri, Ph.D.)

.... Member

(Assistant Professor Twittie Senivongse, Ph.D.)

.... Member

(Nakornthip Prompoon)

.... Member

(Songsak Rongviriyapanich, Ph.D.)

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

คมศัลล์ ศรีวิสุทธิ์ : การนิยามและการตรวจจับร่องรอยที่ไม่ดีของโปรแกรมเชิงแง่มุม (DEFINITION AND DETECTION OF BAD SMELLS OF ASPECT-ORIENTED PROGRAM) อ. ที่ปรึกษา: รศ.ดร.พรศิริ หมื่นไชยศรี, 115 หน้า.

ร่องรอยที่ไม่ดีเป็นคำอุปมาอุปมัยเพื่ออธิบายแบบรูปของซอฟต์แวร์ที่เกี่ยวข้องกับการออกแบบที่ไม่ดีและการโปรแกรมที่ไม่ดี ร่องรอยที่ไม่ดีนี้สามารถกำจัดได้โดยการใช้เทคนิครีแฟคทอริง ซึ่งเป็นเทคนิคที่ใช้ในการปรับปรุงคุณภาพของซอฟต์แวร์ การพัฒนาซอฟต์แวร์เชิงแง่มุมได้รวมเอาแนวคิดใหม่และวิธีคิดในการพัฒนาซอฟต์แวร์ที่แตกต่างไปจากเดิม และแก้ปัญหาการตัดขวาง ซึ่งอาจนำมาสู่ข้อผิดพลาดในการออกแบบที่ต่างไปจากเดิมได้ ดังนั้นการนิยามชนิดของร่องรอยที่ไม่ดี เพื่อบ่งชี้การออกแบบที่ไม่ดีและการโปรแกรมที่ไม่ดีที่ซ่อนอยู่ในซอฟต์แวร์เชิงแง่มุมจึงเป็นสิ่งจำเป็น

งานวิจัยนี้เสนอนิยามของร่องรอยที่ไม่ดีเชิงแง่มุม 5 ชนิดใหม่ที่มีผลกระทบต่อคลับปลิงของซอฟต์แวร์ นอกจากนี้ยังแสดงวิธีแก้ปัญหาเพื่อกำจัดชนิดของร่องรอยที่ไม่ดีในรูปของขบวนการรีแฟคทอริง ทั้งยังออกแบบมาตรวัดและกำหนดช่วงของค่ามาตรวัด เพื่อช่วยในการตรวจจับชนิดของร่องรอยที่ไม่ดีเหล่านั้น และเครื่องตรวจจับร่องรอยที่ไม่ดียังถูกพัฒนาขึ้น เพื่อเป็นเครื่องมือช่วยในการตรวจจับร่องรอยที่ไม่ดีอัตโนมัติ ผลการประเมินร่องรอยที่ไม่ดีที่เสนอแสดงให้เห็นว่า หลังจากกำจัดชนิดของร่องรอยที่ไม่ดีเหล่านั้นแล้ว คลับปลิงของซอฟต์แวร์ลดลง

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

ภาควิชา.....วิศวกรรมคอมพิวเตอร์.... ลายมือชื่อนิสิต ....
สาขาวิชา....วิศวกรรมคอมพิวเตอร์... ลายมือชื่ออาจารย์ที่ปรึกษา..
ปีการศึกษา..........2550...............

4670659821 : MAJOR COMPUTER ENGINEERING

KEY WORD: ASPECT-ORIENTED PROGRAMMING / BAD SMELLS / SOFTWARE METRICS / REFACTORING

KOMSAN SRIVISUT : DEFINITION AND DETECTION OF BAD SMELLS OF ASPECT-ORIENTED PROGRAM. THESIS ADVISOR : ASSOC.PROF. PORNSIRI MUENCHAISRI, Ph.D., 115 pp.

"Bad smell" is a metaphor describing software patterns that are generally associated with bad designs and bad programmings. It can be removed by using the refactoring technique which improves the quality of the software. Aspect-Oriented (AO) software development, which involves new notions and different ways of thinking for developing software and solving the crosscutting problem, possibly introduces different kinds of design flaws. Defining bad-smell kinds hidden in AO software, in order to point out bad designs and bad programmings, is then necessary.

This research proposes the definition of five new kinds of AO bad smells affecting coupling of software. Moreover, appropriate solution to eliminate each kind of bad smell is presented in terms of refactoring procedure. Also, metrics are designed and thresholds are determined to support for detecting such kinds of bad smells. Bad-smell detector is further developed as a tool to support for automatic bad-smell detection. The results of bad-smell validation show that after removing the bad-smell kinds, software coupling is decreased.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Department ....Computer Engineering... Student's Signature ....

Field of study ..Computer Engineering... Advisor's Signature ....

Academic year ....2007.................

ACKNOWLEDGEMENTS

None of this work would have been possible without my thesis advisor, Associate Professor Dr. Pornsiri Muenchaisri. Since I started working with her, I have started to understand and enjoy the peculiarities of research. For all your persistent attention, your constant guidance, and your skill in prodding me to work hard, “Ajarn Pornsiri”, thank you.

I additionally would like to sincerely thank the rest of my thesis committee: Assistant Professor Dr. Wiwat Vatanawood, Assistant Professor Dr. Twittie Senivongse, Ajarn Nakornthip Prompoon, and Dr. Songsak Rongviriyapanich, who reviewed my work and gave insightful comments.

I also would like to thank all my colleagues and friends at the Software Engineering Laboratory who have amused, supported, and loved me. Special thanks to my sisterliness, Matinee Kiewkanya, who helps and discusses on my thesis.

Most of all, I would like to express my sincere gratitude to my parents and family members for their eternal love, supports and hearty encouragements throughout my life.

TABLE OF CONTENTS

Page

ABSTRACT (THAI) iv

ABSTRACT (ENGLISH) v

ACKNOWLEDGEMENTS vi

TABLE OF CONTENTS vii

LIST OF TABLES x

LIST OF FIGURES xii

Chapter

I INTRODUCTION 1

1.1 Motivation 1

1.2 Objective 2

1.3 Scope 2

1.4 Contribution 3

1.5 Publications 3

1.6 Research Methodology 3

1.7 Organization of the Thesis 4

II BACKGROUND AND LITERATURE REVIEW 5

2.1 Background 5

2.1.1 Aspect-Oriented Programming (AOP) 5

2.1.2 AspectJ 8

2.1.3 Refactoring 11

2.1.4 Bad Smells 12

2.1.5 Software Measurement 13

2.2 Literature Review 14

2.2.1 Refactoring 14

2.2.2 Bad Smells 15

2.2.2.1 Anonymous Pointcut Definition 16

2.2.2.2 Feature Envy 17

2.2.2.3 Abstract Method Introduction 18

Chapter Page

2.2.3 Metrics 19

III ASPECT-ORIENTED BAD SMELLS 20

3.1 Our Approach 20

3.1.1 Bad-Smell Definition 21

3.1.2 Bad-Smell Validation 21

3.2 Definitions of Bad Smells 21

3.2.1 Borrowed Pointcut 22

3.2.2 Duplicated Pointcut 24

3.2.3 Various Concerns 25

3.2.4 Identical Role 27

3.2.5 Junk Material 29

IV BAD-SMELL METRICS 32

4.1 Borrowed Pointcut 32

4.2 Duplicated Pointcut 33

4.3 Various Concerns 33

4.4 Identical Role 34

4.5 Junk Material 35

4.6 Anonymous Pointcut Definition 36

4.7 Feature Envy 37

4.8 Abstract Method Introduction 37

V BAD-SMELL VALIDATION 38

5.1 Bad-Smell Metric and Threshold Validation 38

5.1.1 Applying Bad-Smell Metrics and Detecting Bad Smells before Eliminating Bad Smells 39

5.1.2 Applying Bad-Smell Metrics and Detecting Bad Smells after Eliminating Bad Smells 40

5.2 Bad-Smell Validation 40

5.3 Disscussion 42

VI CONCLUSION AND FUTURE WORK 46

6.1 Conclusion 46

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Chapter Page

6.2 Limitation…….. 47

6.3 Future Work… 47

REFERENCES 49

APPENDICES 52

APPENDIX A. PUBLICATIONS 53

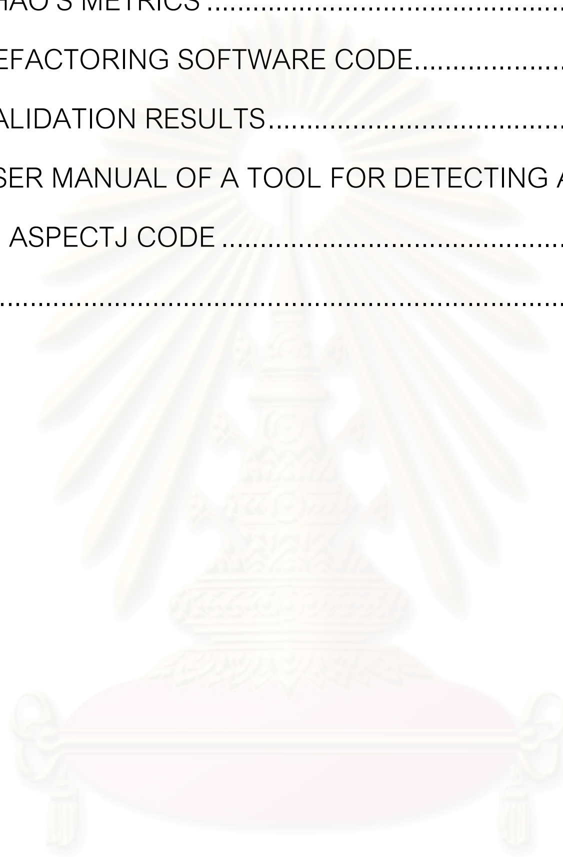
APPENDIX B. ZHAO'S METRICS 70

APPENDIX C. REFACTORING SOFTWARE CODE 75

APPENDIX D. VALIDATION RESULTS 79

APPENDIX E. USER MANUAL OF A TOOL FOR DETECTING AO BAD SMELLS IN ASPECTJ CODE 110

BIOGRAPHY 115



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

LIST OF TABLES

Table Page

5.1 Bad smells in four software samples before eliminating bad smells 40
5.2 Bad smells in four software samples after eliminating bad smells..........................41
5.3 Measured values of Zhao's metrics in Telecom 43
5.4 Measured values of Zhao's metrics in Spacewar..43
5.5 Measured values of Zhao's metrics in AspectTetris..44
5.6 Measured values of Zhao's metrics in AJHotDraw..45
D.1 Measured values of pointcut metrics in Telecom before refactoring 80
D.2 Measured values of pointcut metrics in Telecom after refactoring 80
D.3 Measured values of pointcut metrics in Spacewar before refactoring...................81
D.4 Measured values of pointcut metrics in Spacewar before refactoring (continued) 82
D.5 Measured values of pointcut metrics in Spacewar before refactoring (continued) 83
D.6 Measured values of pointcut metrics in Spacewar after refactoring 84
D.7 Measured values of pointcut metrics in Spacewar after refactoring (continued) ..85
D.8 Measured values of pointcut metrics in Spacewar after refactoring (continued) ..86
D.9 Measured values of pointcut metrics in AspectTetris before refactoring...............87
D.10 Measured values of pointcut metrics in AspectTetris before refactoring (continued) 88
D.11 Measured values of pointcut metrics in AspectTetris before refactoring (continued) 89
D.12 Measured values of pointcut metrics in AspectTetris before refactoring (continued) 90
D.13 Measured values of pointcut metrics in AspectTetris after refactoring 91
D.14 Measured values of pointcut metrics in AspectTetris after refactoring (continued) 92
D.15 Measured values of pointcut metrics in AspectTetris after refactoring (continued) 93

Table Page

D.16 Measured values of pointcut metrics in AspectTetris after refactoring (continued) 94
D.17 Measured values of pointcut metrics in AJHotDraw before refactoring 95
D.18 Measured values of pointcut metrics in AJHotDraw before refactoring (continued) 96
D.19 Measured values of pointcut metrics in AJHotDraw after refactoring 97
D.20 Measured values of pointcut metrics in AJHotDraw after refactoring (continued) 98
D.21 Measured values of aspect metrics in Telecom before refactoring 99
D.22 Measured values of aspect metrics in Telecom after refactoring 99
D.23 Measured values of aspect metrics in Spacewar before refactoring 100
D.24 Measured values of aspect metrics in Spacewar after refactoring 101
D.25 Measured values of aspect metrics in AspectTetris before refactoring 102
D.26 Measured values of aspect metrics in AspectTetris after refactoring 103
D.27 Measured values of aspect metrics in AJHotDraw before refactoring 104
D.28 Measured values of aspect metrics in AJHotDraw before refactoring (continued) 105
D.29 Measured values of aspect metrics in AJHotDraw before refactoring (continued) 106
D.30 Measured values of aspect metrics in AJHotDraw after refactoring 107
D.31 Measured values of aspect metrics in AJHotDraw after refactoring (continued) 108
D.32 Measured values of aspect metrics in AJHotDraw after refactoring (continued) 109

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

LIST OF FIGURES

Figure Page

2.1 Logging concern crosscuts other modules 5
2.2 Code scattering into other modules 6
2.3 Code tangling in a module 7
2.4 Logging concern is encapsulated in its own module 7
2.5 An example of join point 8
2.6 An example of pointcut 8
2.7 An example of advice 9
2.8 An example of introduction 9
2.9 An example of aspect 10
2.10 An example of AspectJ program 10
2.11 Refactoring process 11
2.12 The characteristic of the anonymous pointcut definition bad smell 16
2.13 After eliminating the anonymous pointcut definition bad smell 16
2.14 The characteristic of the feature envy bad smell 17
2.15 After eliminating the feature envy bad smell 18
2.16 The characteristic of the abstract method introduction bad smell 19
3.1 Our approach 20
3.2 The characteristic of the borrowed pointcut bad smell 23
3.3 After eliminating the borrowed pointcut bad smell 23
3.4 The characteristic of the duplicated pointcut bad smell 24
3.5 After eliminating the duplicated pointcut bad smell 25
3.6 The characteristic of the various concerns bad smell 26
3.7 After eliminating the various concerns bad smell 27
3.8 The characteristic of the identical role bad smell 28
3.9 After eliminating the identical role bad smell 29
3.10 The characteristic of the junk material bad smell 30
3.11 After eliminating the junk material bad smell 31
C.1 Aspect Timing of Telecom software before applying refactoring procedures 75

Figure Page

C.2 Aspect TimerLog of Telecom software before applying refactoring procedures ...76
C.3 Aspect Billing of Telecom software before applying refactoring procedures 76
C.4 Aspect Timing of Telecom software after applying refactoring procedures 77
C.5 Aspect TimerLog of Telecom software after applying refactoring procedures.......77
C.6 Aspect Billing of Telecom software after applying refactoring procedures 78
C.7 Aspect XPI of Telecom software taken place from the eliminating borrowed pointcut solution 78
E.1 Main preference page of Eclipse after extracting the plug-in's zip file 110
E.2 AO Bad Smells View 111
E.3 Opening AO Bad Smells View 112
E.4 Show View dialog 112
E.5 Bad Smell Detector menu 113
E.6 An example of detection results 114

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER I

INTRODUCTION

1.1 Motivation

As the traditional Object-Oriented (OO) programming (OOP) unintently introduces the problem of code scattering and code tangling in software development which is called crosscutting concern, Aspect-Oriented (AO) programming (AOP) [1] is emerging as the new programming paradigm to solve such problem by separating the crosscutting concerns into their own modules called aspects.

Coupling [2] is an internal quality attribute of software that can be used to indicate the degree of interdependence among the components of a software system. It has been recognized that good software design should obey the principle of low coupling. A system that has strong coupling is difficult to understand, change, and correct highly interrelated components in the system. There are two types of software coupling i.e. interaction coupling and inheritance coupling. In the event of developer cannot avoid coupling in software, inheritance coupling is more desirable than interaction coupling.

Bad smells [3] are design flaws in existing software that should be removed through refactoring. Having bad smells do not always suggest that a refactoring is needed. It rather suggests something may be wrong in the design or code. Decisions for removing bad smells thus depend on the specific aims of a programmer and the specific state and structure of the code on which he is working. Refactoring [3] is a technique for improving the design of an existing software by changing the internal structure of the software, while the behavior of the original software is preserved.

Since new notions and the different ways of thinking are introduced in order to support for identification, modularization, representation, and composition of crosscutting concerns, they perhaps introduce different kinds of design flaws. Interaction coupling between aspects and classes is also introduced. Therefore, defining the bad-smell kinds hidden in AO program is required as a means to identify

possibly anomalies. This research proposes the definition of new kinds of AO bad smells affecting coupling of software. Existing AO refactoring methods, which correspond to the solution for eliminating each kind of bad smell, are further presented. Also, AO software metrics are designed and thresholds are determined as indicators to identify bad smells hidden in AO program. Automatic tool is also developed to support for bad-smell detection.

1.2 Objective

The objectives of this research are as follows:

1. To define specific kinds of bad smells hidden in AO program.
2. To design AO software metrics and determine thresholds for supporting the bad-smell detection phase.
3. To suggest appropriate AO refactoring methods, which correspond to solution for eliminating each kind of bad smell.
4. To develop supporting tool for detecting bad smells in AspectJ code.

1.3 Scope

1. There are several techniques used to support bad-smell detection. This research focuses on software metrics.
2. The bad-smell metrics and their thresholds are designed and determined to support for detecting our kinds of bad smells and Piveta's bad smells.
3. The solution for eliminating each kind of AO bad smell can be used for a particular fraction of code which conforms to our problem examples. For more complex structure of code, our solution can partially be applied.
4. The defined bad-smell kinds are validated by comparing coupling before and after eliminating these bad smells.
5. The supporting tool covers only bad-smell detection and appropriate refactoring suggestions.
6. Sample program codes used for testing supporting tool must be developed based on AspectJ version 1.2 and priorly compiled.

7. At least two sample programs are used to test the supporting tool. Also, all samples should include at least ten classes and five aspects.

1.4 Contribution

The outcomes of this research are the followings:

1. The defined bad-smell kinds can be used to identify possibly anomalies in AO software. Then, to improve the quality of the software, appropriate refactoring methods, which we suggest, could be applied.
2. The designed bad-smell metrics and determined thresholds can be used to support for bad-smell detection and be applied to refactoring application phase for automation.
3. An automated supporting tool can be used for automatically detecting bad smells in AspectJ code.

1.5 Publications

Several parts of our research have been selected to be presented in both national and international conferences and published in the corresponding proceedings detailed in Appendix A.

1.6 Research Methodology

1. Study all topics related to the researches including AOP, AspectJ programming language, refactoring technique, bad smells, and software measurement.
2. Review and study the research papers in both paradigms i.e. object orientation and aspect orientation which are related to refactoring, bad smells, and metrics.
3. Define the kinds of AO bad smells and their metrics. Besides, determine threshold for each bad-smell metric.

4. Suggest the proper solution in order to eliminate these kinds of bad smells. After that, match appropriate refactoring methods to each bad-smell eliminating solution.
5. Develop a supporting tool for detecting the bad smells in AspectJ code.
6. Validate the bad-smell kinds with quality metrics by comparing the measured values of coupling metrics before and after eliminating them.
7. Analyze the results and make conclusions.
8. Write thesis.

1.7 Organization of the Thesis

The remainder of the thesis is organized into six chapters as follows.

Chapter II presents theoretical background including introduction of AOP, AspectJ, refactoring, bad smells, and software measurement. This chapter also reviews several researches related to refactoring, bad smells, and metrics in the light of aspect orientation.

Chapter III describes the approach of this research and the definitions of five AO bad smell kinds.

Chapter IV presents metrics and thresholds, which are used to support for detecting AO bad smells including our five kinds of bad smells and three kinds of Piveta's bad smells.

Chapter V shows the results from validating the AO bad smells, bad-smell metrics and their thresholds.

Finally, chapter VI concludes research work and presents some directions for the future work. Limitations of our work are also detailed.

CHAPTER II

BACKGROUND AND LITERATURE REVIEW

2.1 Background

This section reviews the theoretical background used in this thesis including AOP, AspectJ, refactoring, bad smells, and software measurement.

2.1.1 Aspect-Oriented Programming (AOP)

Separation of concerns [4] is about breaking down software into distinct parts that overlap in functionality as little as possible. All programming methodologies – including procedural programming and OOP – support some separations and encapsulations of concerns into single entities. For example, procedures, packages, classes, and methods all help programmers encapsulate concerns into single entities.

Unfortunately, there are some concerns defy these forms of encapsulation namely “crosscutting concerns”. For example, a logging strategy necessarily affects every single logged part of the system e.g. accounting, ATM, and database as shown in Figure 2.1. Logging thereby crosscuts all logged classes and methods.

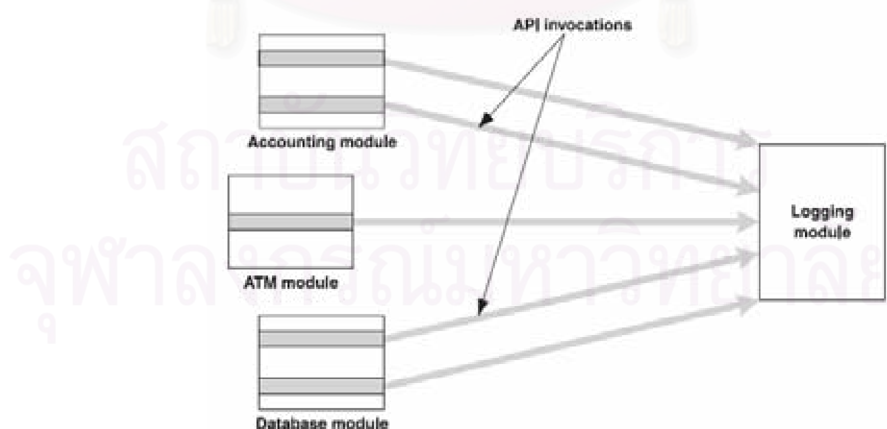


Figure 2.1: Logging concern crosscuts other modules [5].

The crosscutting concerns introduce the problem of code scattering and code tangling in software development as follows [5]:

- **Code scattering** occurs when a single concern is implemented in multiple modules. Since crosscutting concern is spread over many modules, related implementations are also scattered over all those modules. For example, many modules on the system must embed the code to ensure that only authorized users access the services as shown in Figure 2.2.

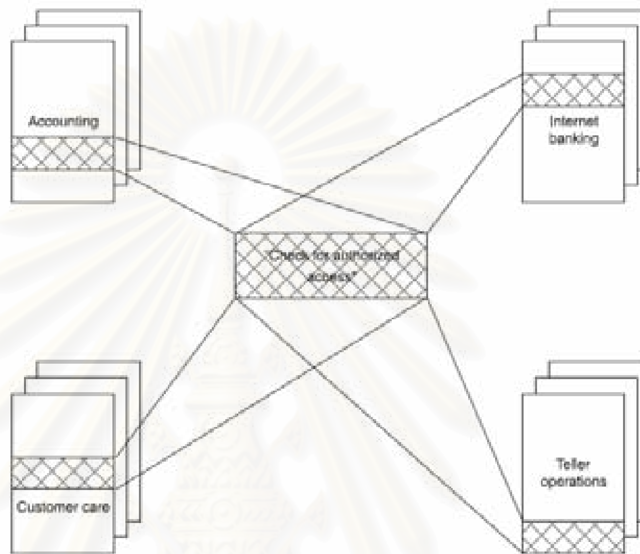


Figure 2.2: Code scattering into other modules [5].

- **Code tangling** taken place when a module is implemented in a way that it handles multiple concerns simultaneously. A programmer often considers concerns such as business logic, performance, synchronization, logging, security, and so forth while implementing a module. This leads to the simultaneous presence of elements from each concern's implementation and results in the code tangling. For example, the module in the system manages a part of multiple concerns as shown in Figure 2.3.

Both of code scattering and code tangling affect software design and development in many ways: poor traceability, lower productivity, lower code reuse, and harder evolution, resulting in poor software quality.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

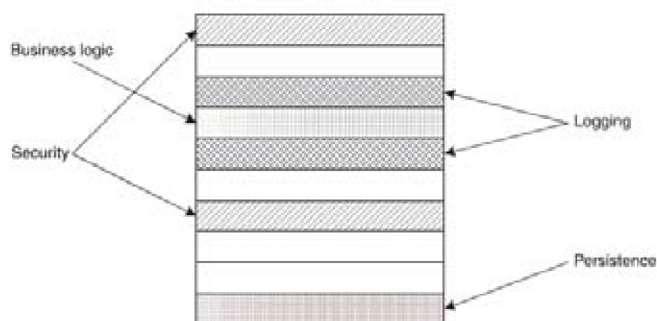


Figure 2.3: Code tangling in a module [5].

AOP is the new programming paradigm which attempts to aid programmers in the separation of concerns, specifically crosscutting concerns, as an advance in modularization. For example, logging concern, which previously spread over many modules, is encapsulated in its own module i.e. Logging aspect as shown in Figure 2.4.

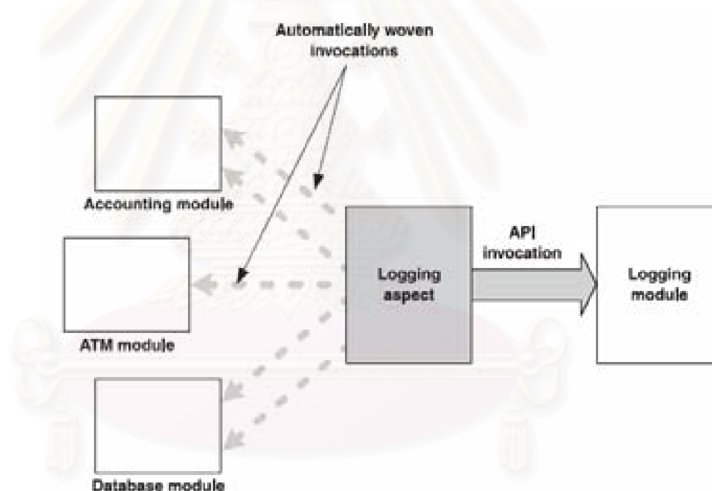


Figure 2.4: Logging concern is encapsulated in its own module [5].

Benefits of AOP are cleaner responsibilities of the individual module, higher modularization, easier system evolution, late binding of design decisions, more code reuse, improved time-to-market, and reduced costs of feature implementation.

The AspectJ language [5], which is the most popular one and already has a large community, is detailed in the next section.

2.1.2 AspectJ

AspectJ [5] is an AO extension to the Java programming language. It uses Java-like syntax, all valid Java programs are also valid AspectJ programs, but AspectJ also allows programmers to define special constructs called *aspects*. Aspects can contain several entities unavailable to standard classes. There are join point, pointcut, advice, and introduction. Each entity and also the aspect will be discussed in depth in the following sections.

2.1.2.1 Join Point

Join point is an identifiable point in the execution of a program. It could be a call to a method or an assignment to a member of an object. For example as shown in Figure 2.5, the join points in the `Account` class include the execution of the `credit()` method and the access to the `_balance` instance member.

```
public class Account {
	…
	void credit(float amount) {
		_balance += amount;
	}
}
```

Figure 2.5: An example of join point [5].

2.1.2.2 Pointcut

Pointcut is a program construct that selects join points and collected context at those points. For example as shown in Figure 2.6, this pointcut captures the execution of the `credit()` method in the `Account` class.

```
execution(void Account.credit(float))
```

Figure 2.6: An example of pointcut [5].

2.1.2.3 Advice

Advice is the code to be executed at a join point that has been selected by a pointcut. Advice can execute before, after, or around the join point. The body of advice is much like a method body-it encapsulates the logic to be executed upon reaching a join point. Using the earlier pointcut, we can write advice that will print a message before the execution of the `credit()` method in the `Account` class as shown in Figure 2.7.

```
before() : execution(void Account.credit(float)) {
      System.out.println("About to perform credit operation");
}
```

Figure 2.7: An example of advice [5].

2.1.2.4 Introduction

Introduction or intertype declaration is an instruction that introduces changes to the classes, interfaces, and aspects of the system. It makes static changes to the modules that do not directly affect their behavior. Figure 2.8 shows an introduction which declares the `Account` class to implement the `BankingEntity` interface.

```
declare parents: Account implements BankingEntity;
```

Figure 2.8: An example of introduction [5].

2.1.2.5 Aspect

Aspect is the central unit of AspectJ, in the same way that a class is the central unit in Java. It contains the code that expresses the weaving rules for both dynamic and static crosscutting. Pointcuts, advices, introductions, and declarations are combined in an aspect. In addition to the AspectJ elements, aspects can contain data, methods, and nested class members, just like a normal Java class. All the code examples from section 2.1.2 can be merged together in an aspect as shown in Figure 2.9.

```
public aspect ExampleAspect {
      before() : execution(void Account.credit(float)) {
            System.out.println("About to perform credit operation");
      }
      declare parents: Account implements BankingEntity;
}
```

Figure 2.9: An example of aspect [5].

```
public class Home {
      public void enter() {
            System.out.println("Entering");
      }
      public void exit() {
            System.out.println("Exiting");
      }
}

public aspect HomeSecurityAspect {
      before() : call(void Home.exit()) {
            System.out.println("Engaging");
      }
      after() : call(void Home.enter()) {
            System.out.println("Disengaging");
      }
}

public aspect SaveEnergyAspect {
      before() : call(void Home.exit()) {
            System.out.println("Switching off lights");
      }
      after() : call(void Home.enter()) {
            System.out.println("Switching on lights");
      }
}

public aspect HomeSystemCoordinationAspect {
      declare precedence: HomeSecurityAspect, SaveEnergyAspect;
}

public class TestHome {
      public static void main(String[] args) {
            Home home = new Home();
            home.exit();
            System.out.println();
            home.enter();
      }
}
```

Figure 2.10: An example of AspectJ program [5].

The simple example of AspectJ program is shown in Figure 2.10. This example represents the management of security system and conserving energy system in the home. Whenever home's owner leave, the security system is opened and the lights are switched off to conserve the energy. Otherwise, the lights are switched on and the security system is closed when home's owner come back.

2.1.3 Refactoring

Opdyke [6], who defined the refactoring technique, stated in his PhD dissertation that *refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written*. There are seventy-two refactoring methods used to restructure OO codes.

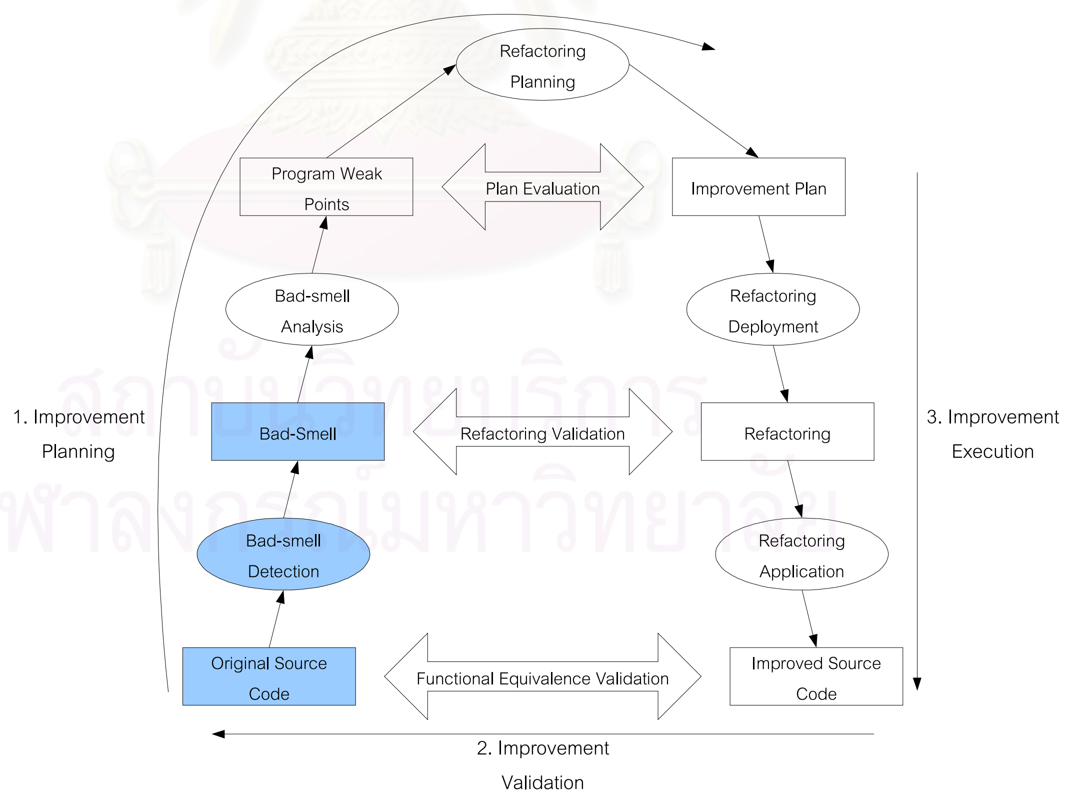


Figure 2.11: Refactoring process [7].

The refactoring process [7] consists of three major subprocesses, which are improvement planning, improvement validation, and improvement execution illustrated in Figure 2.11.

1. Improvement planning

The goal of this subprocess is to identify refactoring candidates. Starting with the identification of program points to be refactored, it includes organization of refactorings and selection of refactoring to be applied.

2. Improvement validation

This subprocess consists of three different validations. Each validation has its own objective. At the developer level, mainly the functional equivalence before and after the refactoring should be validated. At the analyst level, the intended effect should be validated. At the manager level, the cost-effect trade-off should be substantiated.

3. Improvement execution

The major objective of the subprocess is to apply refactoring to the target program. It includes the ordering of each refactoring according to the priority in terms of cost-effect trade-off by the analyst and the actual code modification by the developer.

In this research, we only focus on the bad-smell detection phase which is the first phase of the improvement planning subprocess.

2.1.4 Bad Smells

According to Beck [3], bad smells are "*structures in the code that suggest (sometimes scream for) the possibility of refactoring*". In other words, bad smells are design flaws in existing code that should be removed through refactorings. Having bad smells do not always indicate that a refactoring must be performed. Instead, it suggests symptoms indicating something might be wrong in design or code. Programmers are required to develop their own sense of when a symptom indeed warrants a change. Decisions also depend on the specific aims of the programmers and the specific state and structure of the code on which they are working.

For Object Orientation, there are twenty-two kinds of bad smell. For example, *Feature Envy* is a method that is more interested in a class other than the one it's actually in. In general, try to put a method in the class that contains most of the data the

method needs. *Move method* refactoring method can be applied to remove such kind of bad smell.

Another example of bad smell is *large class*. A class that is trying to do too much can usually be identified as *large class* by looking at how many instance variables it has. When a class has too many instance variables, duplicated code can not be far behind. To remove this kind of bad smell, *extract class* refactoring method can be applied.

There are several techniques used to detect bad smells in code for example, clone analysis tool, logic meta programming, ac hoc approach, visualization mechanism, and OO metrics [8]. In this research, we use software metrics to detect AO bad-smell kinds.

2.1.5 Software Measurement

Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules [9]. In the assessment process prescribed by ISO-9126 [10], the goals of measurement must first be defined, then the measurement itself is specified, the means of measurement are implemented and the measurement is carried out. In a final step, the measurement results are evaluated.

Software metrics have been classified by Fenton [9] into three classes.

- **Process metrics** are used to measure characteristics of software processes such as the development process, the maintenance process or the testing process. Typical process characteristics are effort involved, costs occurred, tasks accomplished and elapsed time.
- **Product metrics** are used to measure characteristics of software products such as programs, components, system and databases. Typical product characteristics are size, complexity and various qualities.
- **Resource metrics** are used to measure characteristics of software resources which may be hardware, software or people. Typical resource characteristics are performance, availability, reliability and productivity.

Fenton distinguishes further between internal and external attributes.

- **Internal attributes** of a product, process or resource are those which can be measured purely in terms of the product, process or resource itself. Internal attributes of software products are, for example, complexity, modularity, testability and reusability. They can be measured by examining the source code itself.
- **External attributes** are those which can only be measured with respect to how the product, process or resource relates to its environment. External attributes of software products are, for example, reliability, security, usability and performance. They can only be measured by testing the product in a particular environment.

Coupling [2] is one of internal quality attributes that can be used to indicate the degree of interdependence among the components of software system. Coupling is thought to be a desirable goal in software construction, leading to better values for external attributes such as maintainability, reusability, and reliability. In this research, we use Zhao's metrics suite [11] with regard to the coupling to validate our kinds of bad smells. The definitions of Zhao's metrics are detailed in Appendix B.

2.2 Literature Review

Several research works related to refactoring, bad smells, and metrics in the light of aspect orientation are reviewed in this section.

Bad smells and refactorings are closely related, since bad smells can be removed by using the refactoring technique. However, the prior researches in the light of aspect orientation focused mostly on the refactoring technique i.e. very few are related to bad smells. Several researchers proposed refactoring methods to support the refactoring process.

2.2.1 Refactoring

Refactorings which are related to AOP can be classified into two major groups. The first group covers extracting the crosscutting concerns, which are embedded in

base code, into aspect modules. The second group covers restructuring the aspect code in order to improve the design of AO software.

Runa [12] proposed thirty new fundamental AOP-specific refactorings and recasted the existing (OO) refactorings to preserve program behavior in AO code. Composite refactorings, which are built from their fundamental refactorings, are additionally presented to aid in the extraction of crosscutting concerns by deploying AOP techniques in existing programs. In order to guarantee behavior preservation in AspectJ, preconditions are further introduced to Runa's refactoring methods.

Hanenberg et al. [13] introduced a number of new AO refactorings which help to migrate from OO to AO software and to restructure existing AO code. There are three refactorings to restructure existing AO code such as *Extract Advice*, *Extract Introduction*, and *Separate Pointcut*. Likewise, Monteiro and Fernades [14] proposed a collection of twenty-eight AO refactorings cover both the extraction of aspects from OO legacy code and subsequent tidying up of the resulting aspects. They also reviewed the traditional OO code smells in the light of aspect orientation and proposed some new smells for the detection of crosscutting concerns. In addition, they firstly proposed a new code smell that is specific to aspect named *Aspect Laziness* – an aspect that does not carry the full weight of its responsibilities and instead pass the burden to classes.

Runa's refactorings and Monteiro and Fernades's refactoings are selected to match with solution for eliminating each kind of AO bad smells in this research.

2.2.2 Bad Smells

Piveta et al. [15] defined five kinds of bad smells that occur in AO systems i.e. *anonymous pointcut definition*, *large aspect*, *lazy aspect*, *feature envy*, and *abstract method introduction*. They also complemented their work with algorithms to automatically detect their five defined bad-smell kinds, more specifically those written using AspectJ language [16]. Characteristics of some Piveta's bad smells resemble to our bad smells, but the main difference is the technique which is used for detecting bad smells. Details of some Piveta's bad smells are described next. We additionally map refactoring method to some of bad-smell eliminating solutions.

2.2.2.1 Anonymous Pointcut Definition

Definition: A pointcut is unnamed [15].

Impact: In AspectJ, as pieces of advice are not named, sometimes it is necessary to rely on the pointcut definition to remark on the affected points in base code. The reusability of common pointcuts is also reduced.

Figure 2.12 illustrates an example of *anonymous pointcut definition*. Advice *adviceA1* of aspect *aspectA* includes a pointcut which is unnamed. Hence, such pointcut is considered to be the *anonymous pointcut definition*.

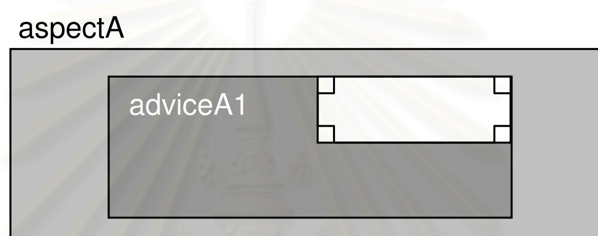


Figure 2.12: The characteristic of the *anonymous pointcut definition* bad smell.

Solution: A name, that clearly defines the pointcut intention, could be defined and referenced by any advice and declare-construction [16]. *Anonymous pointcut definition* is removed by applying refactoring procedure as follows:

1. Create a new pointcut and give it the name by using *Create Named Pointcut* [12].

As pointcut of advice *adviceA1* could be the candidate of the *anonymous pointcut definition* kind, the above refactoring procedure is applied and the result is illustrated in Figure 2.13.

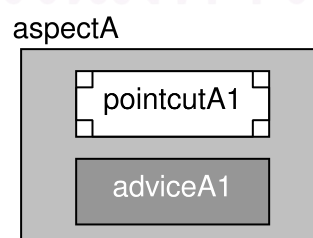


Figure 2.13: After eliminating the *anonymous pointcut definition* bad smell.

2.2.2.2 Feature Envy

Definition: A single aspect uses a class-defined pointcut [15].

Impact: In AspectJ, pointcuts could be defined in aspects and also in classes. If a single aspect uses a class-defined pointcut, the interaction coupling between aspect and class unnecessarily takes place. The same problem might occur also among classes. It happens when a class extensively refers to members of another class instead of referring to that of its own.

This bad smell resembles to our *borrowed pointcut* bad smell presented next but the pointcut, which is within the scope of *borrowed pointcut*, is defined in an aspect.

Figure 2.14 illustrates an example of *feature envy*. Advice *adviceB2* refer to pointcut *pointcutA1* defined in class *classA*. Hence, aspect *aspectB* is considered to be the *feature envy*.

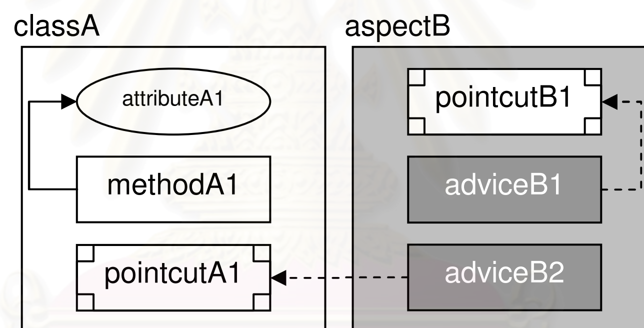


Figure 2.14: The characteristic of the *feature envy* bad smell.

Solution: The suspected pointcut should be moved from a class to the aspect using it by applying refactoring procedure as follows:

1. Move suspected pointcut to the referring aspect by using *Move Named Pointcut* [12].

As aspect *aspectB* could be the candidate of the *feature envy* kind, the above refactoring procedure is applied resulting as illustrated in Figure 2.15.

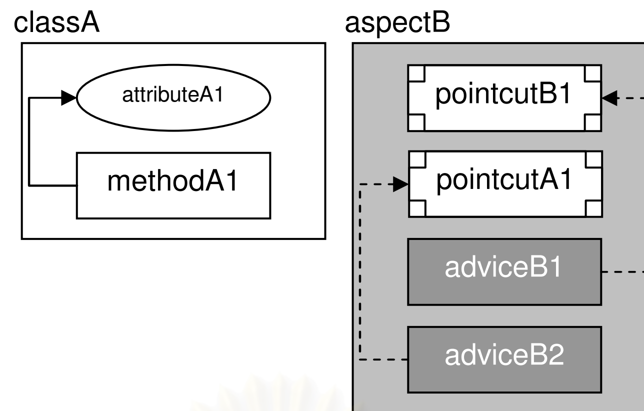


Figure 2.15: After eliminating the *feature envy* bad smell.

2.2.2.3 Abstract Method Introduction

Definition: An abstract method is inserted in application class through the inter-type declaration mechanism [15].

Impact: Aspect could be used to add state and behavior into existing classes. This is made through the inter-type declaration mechanism. This mechanism allows methods and/or attributes to be inserted in classes. However, the use of this functionality may cause problems when abstract methods are inserted in application classes. This introduction forces the programmer to provide concrete implementations to the introduced methods in every affected class and subclass. This dependency unnecessarily increases the coupling between the aspect and the affected classes [16].

Figure 2.16 illustrates an example of *abstract method introduction*. Aspect *aspectA* introducing method *methodD1* to class *classB* and class *classC*, cause method *methodD1* is introduced to class *classD*. Hence, this aspect is considered to be the *abstract method introduction.*

Solution: This kind of bad smell is not harmful and unnecessary to be eliminated from software code, because this kind of dependency occurs according to the generalization of classes. For instance, common introduced methods of subclasses are pulled up into their superclass.

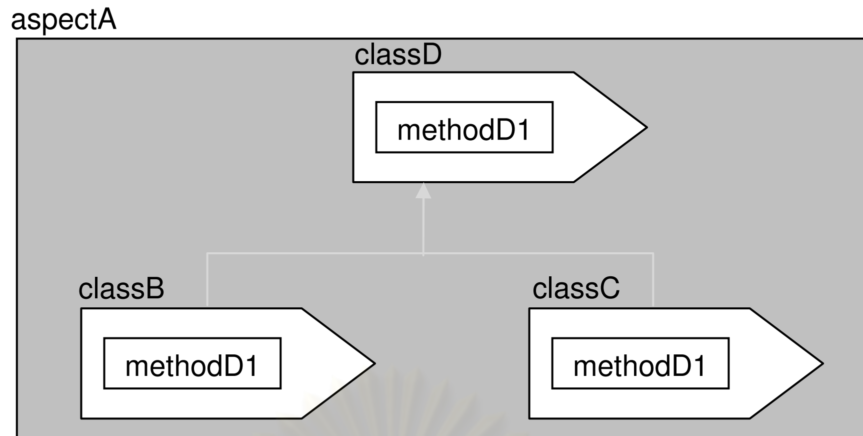


Figure 2.16: The characteristic of the *abstract method introduction* bad smell.

To reconcile with our work, we additionally propose bad-smell metrics and their thresholds to some Piveta's bad smells in Chapter IV.

2.2.3 Metrics

Metrics for AOP can be classified into two groups. One is revised from traditional OO metrics and the other one is AO specific metrics.

There are several AO metrics revised from traditional OO metrics. For example, Ceccato and Tonella [17] proposed AO metrics which were revised from the Chidamber and Kemerer's metrics suite [18]. Some of their metrics were adapted or extended in order to make them applicable to the AOP software. They also proposed other metrics which measure specifically the novel kinds of coupling introduced by AOP.

Zhao [11] proposed metrics suite to measure coupling in AO system thoroughly. In AO systems, coupling is mainly about the degree of interdependence among aspects and/or classes. They formally defined various coupling metrics in term of different types of dependencies between aspects and classes.

In this research, we use Zhao's metrics to validate our defined bad-smell kinds in Chapter V.

CHAPTER III

ASPECT-ORIENTED BAD SMELLS

This chapter is divided into two parts. The first part describes the approach of this research. The second part details the definitions of five defined bad-smell kinds.

3.1 Our Approach

The approach of this research consists of two main processes i.e. defining bad smells and validating bad smells. Figure 3.1 shows activity diagram of our approach.

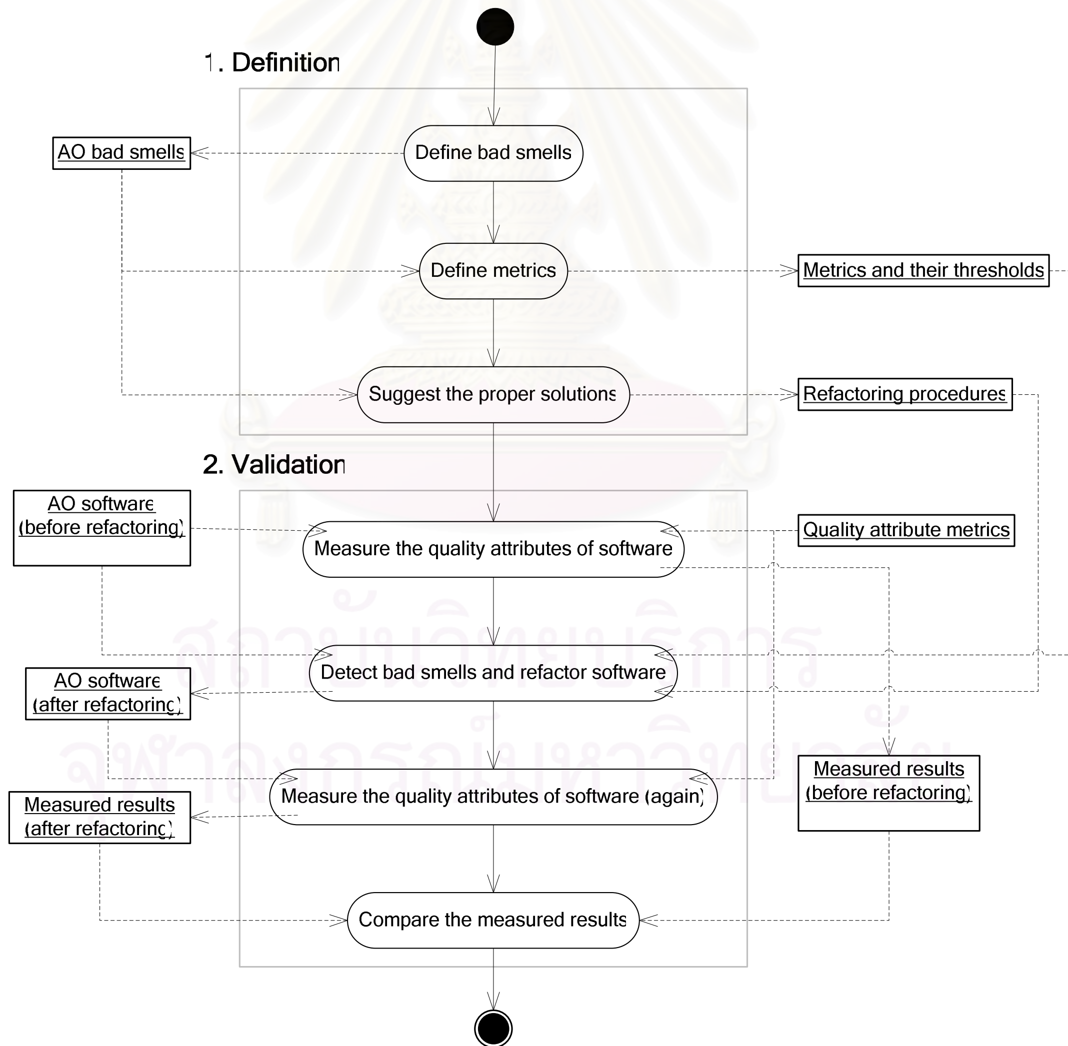


Figure 3.1: Our approach.

3.1.1 Bad-Smell Definition

In bad-smell definition process, we first define kinds of AO bad smells by considering programming patterns that affect coupling of software. Coupling is crucially considered in this thesis because having low coupling is thought to be a desirable goal in software construction, leading to better values for external attributes such as maintainability, reusability, and reliability. In addition, AOP introduces interaction coupling between aspects and classes. We would like to study on the trade-off between the advantages obtained from AOP and disadvantages caused by the coupling introduced by aspects.

After that, the metrics, which correspond to the characteristics of these kinds of bad smells, are designed. Also, the ranges of measured values of the designed metrics are determined in order to indicate suspected entities in program as the bad smells. For the purpose of eliminating the bad smells, the proper solutions, which improve quality of software affected by AO bad smells, are suggested. Existing refactoring methods are then mapped to these solutions. The details of each defined bad smell are described in section 3.2. As for bad-smell metrics and thresholds are presented in Chapter IV.

3.1.2 Bad-Smell Validation

In order to validate the defined bad-smell kinds, we compare the values from quality attribute metrics applied before and after removing these bad smells from AO programs. If the results are in the way that coupling is improved, then the defined bad-smell kinds can be used to indicate design flaws in AO software. Chapter V shows the results from the validation.

3.2 Definitions of Bad Smells

In this section, the definition of each bad-smell kind is summarized according to the following template:

Definition: The explanation of a bad-smell kind characteristic.

Impact: The description on how the bad-smell kind affects software.

Solution: The way to improve software quality. In this part, the appropriate refactoring procedure, which maps to the solution, is presented.

There are five kinds of bad smells detailed here, including *borrowed pointcut*, *duplicated pointcut*, *various concerns*, *identical role*, and *junk material*. In order to consider our defined bad-smell kinds, all considered pointcuts should be named according to the characteristics of these bad smells.

3.2.1 Borrowed Pointcut

Definition: A pointcut is referred to by advices of the aspects of which are not subaspects[1].

Impact: In OO design, there are two types of coupling between classes which are interaction coupling and inheritance coupling [2]. Interaction coupling is the interconnection between classes through message passing. Inheritance coupling is the interconnection between classes through inheritance. Since aspect is a conceptual unit likes object in OOP, similarly coupling between aspects are interaction coupling and inheritance coupling. AOP also introduces the interaction coupling between classes and aspects. Coad and Yourdon [19] suggest that high inheritance coupling is desirable. As opposed to inheritance coupling, low interaction coupling is desirable in OO software systems.

In the case of several advices of other aspects refer to a pointcut of the aspect of which is not a superaspect; it possibly creates the interaction coupling between unrelated aspects. Although this kind of reference reduces the interaction coupling between classes and aspects. Such pointcut is considered to be *borrowed pointcut*.

Figure 3.2 illustrates an example of *borrowed pointcut*. The dashed arrows show a reference from advice to pointcut. Advice *adviceB2* and advice *adviceC1* refer to pointcut *pointcutA1* in aspect *aspectA* but aspect *aspectA* is not the superaspect of the other aspects. Hence, pointcut *pointcutA1* is considered to be the *borrowed pointcut*.

---

[1] A subaspect is the concrete extension of an abstract aspect, the concept being similar to subclasses in OO language [20].

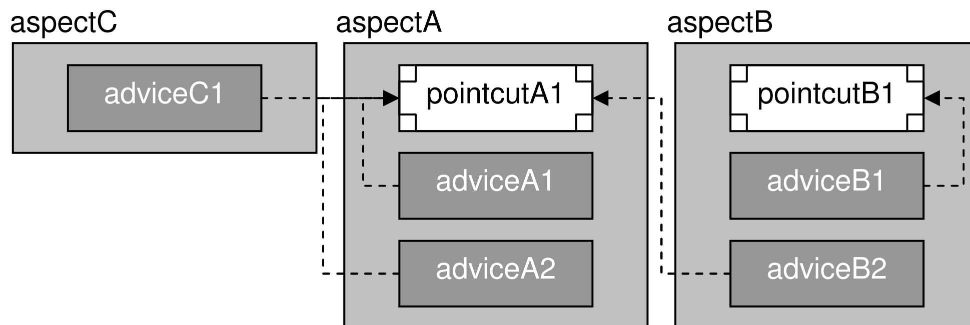


Figure 3.2: The characteristic of the *borrowed pointcut* bad smell.

Solution: Since the *borrowed pointcut* introduces the interaction coupling between unrelated aspects, crosscutting programming interface[2] (XPI) [21, 22] is used to reduce such kind of coupling by changing it to be inheritance coupling. Refactoring procedure should be applied as follows:

1. Create a new aspect as XPI to collect unrelated aspect pointcuts by using *Create Empty Aspect* [12] and specify this aspect to be public.
2. Move suspected pointcuts to the created aspect by using *Move Named Pointcut* [12] and specify these pointcuts to be public.

As pointcut *pointcutA1* could be the candidate of the *borrowed pointcut* kind, the above refactoring procedure is applied and the result is illustrated in Figure 3.3. The interaction coupling is decreased.

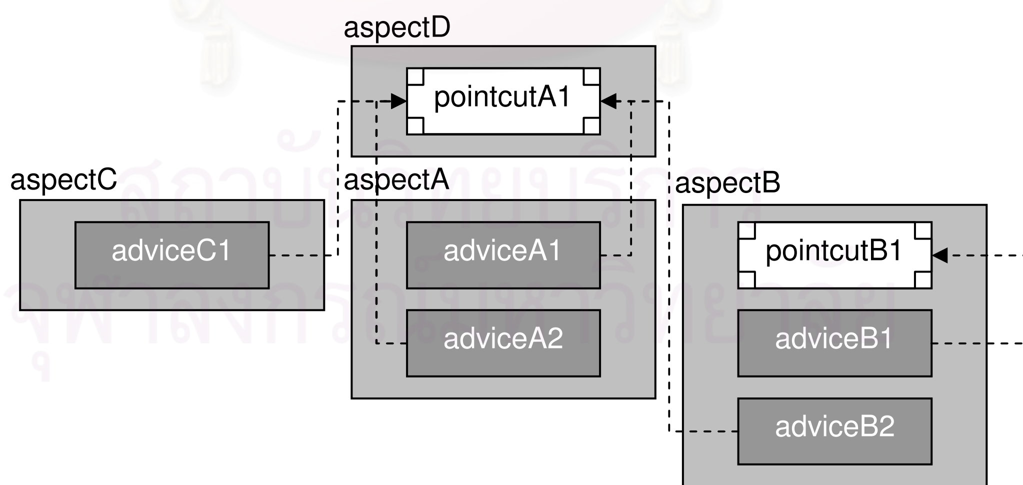


Figure 3.3: After eliminating the *borrowed pointcut* bad smell.

---

[2] XPIs are explicit, abstract interfaces that decouple aspects from details of advised code.

3.2.2 Duplicated Pointcut

Definition: Pointcuts, which are of the same type, collecting the same set of join points in base code.

Impact: Many pointcuts, which might differently be defined and are of the same type, collect the same set of join points in base code. This is a kind of duplicate codes that affects the size of code. In general, the larger the system size, the more difficult it is to understand the system. The interaction coupling also occurs among aspects and classes, since aspect intercepts the execution of classes. Pointcuts, which collect the same set of join points and are of the same type, are the *duplicated pointcut*s.

Figure 3.4 illustrates an example of *duplicated pointcut*. The dotted arrows show a crosscutting from aspect to classes. Pointcut *pointcutA1* and pointcut *pointcutB1*, which are differently defined and are of the same type, are intercepting to the same set of join points in both classes. Hence, pointcut *pointcutA1* and pointcut *pointcutB1* are considered to be the *duplicated pointcut*.

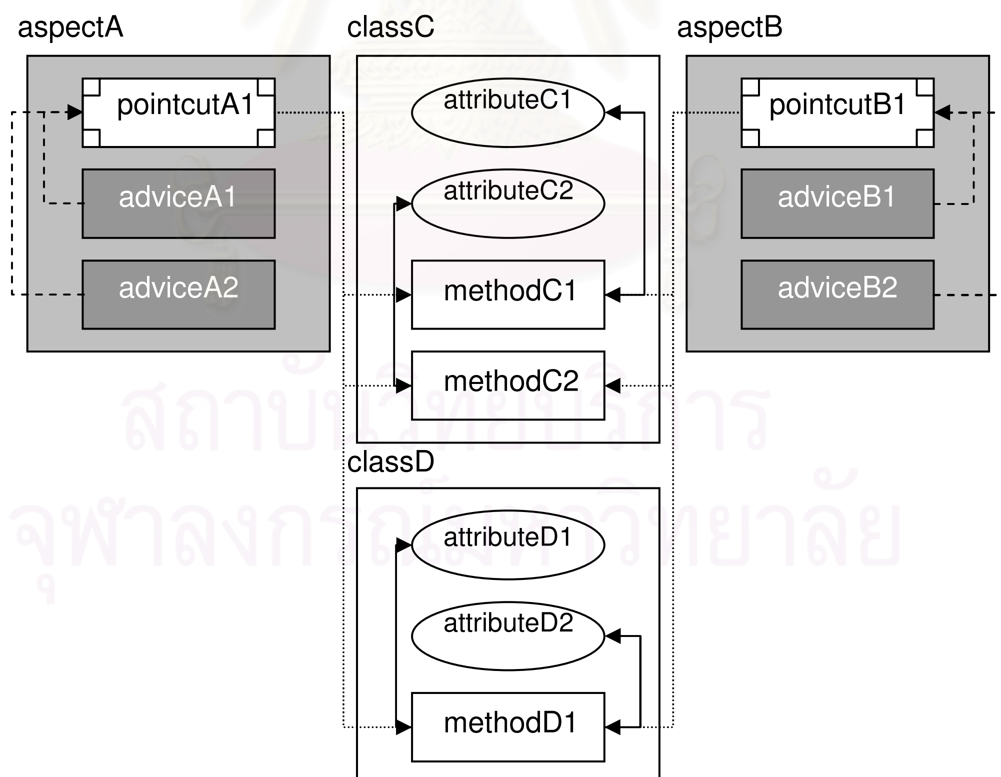


Figure 3.4: The characteristic of the *duplicated pointcut* bad smell.

Solution: Since the *duplicated pointcut* introduces the interaction coupling among aspects and classes, XPI [21, 22] is used to reduce such kind of coupling and duplication of codes by applying refactoring procedure as follows:

1. Create a new aspect as XPI to collect duplicated pointcuts by using *Create Empty Aspect* [12] and specify this aspect to be public.
2. Move suspected pointcuts to the created aspect by using *Move Named Pointcut* [12] and specify this pointcut to be public.
3. Delete the redundant pointcuts by using *Delete Named Pointcut* [12].

As pointcut *pointcutA1* and pointcut *pointcutB1* could be the candidates of the *duplicated pointcut* kind, the above refactoring procedure is applied and the result is illustrated in Figure 3.5. The interaction coupling and also duplication of codes are decreased.

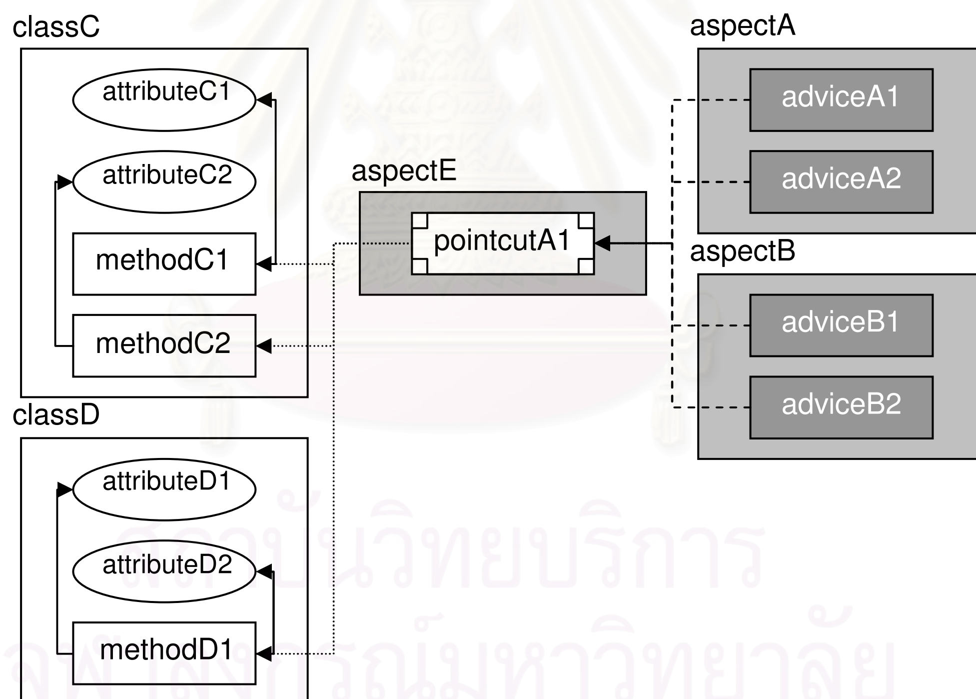


Figure 3.5: After eliminating the *duplicated pointcut* bad smell.

3.2.3 Various Concerns

Definition: A pointcut is referred to by more than one advice, which are the same kind (either before advice or after advice), in an aspect.

Impact: An aspect performs too many functions which often shows up as too many pointcuts. In general, an aspect modularizes a unique concern. When an aspect has too many pointcuts and advices, it implicitly indicates that there may be more than one unrelated concern in such aspect.

This bad smell is similar to *large aspect* bad smell proposed by Piveta et al. [15]. The difference between *various concerns* and *large aspect* is on how to consider the number of concerns in an aspect. For instance, in *various concerns* bad smell, the relationship between pointcuts and advices in an aspect is considered, but in *large aspect*, the number of members in an aspect is considered. *Large aspect* threshold is defined by the user of the function or given as a constant. In [16], the threshold is defined according to the number of crosscutting members of its aspect. The range is defined by analyzing the data from AJHotDraw [23] with the negative binomial statistical distribution. The results from an analysis determine that an aspect with ten or more crosscutting members is marked as a *large aspect*.

Figure 3.6 illustrates an example of *various concerns*. Pointcut *pointcutA2* is referred to by advice *adviceA2* and advice *adviceA3* which are the same kind. Hence, aspect *aspectA* is considered to be the *various concerns.*

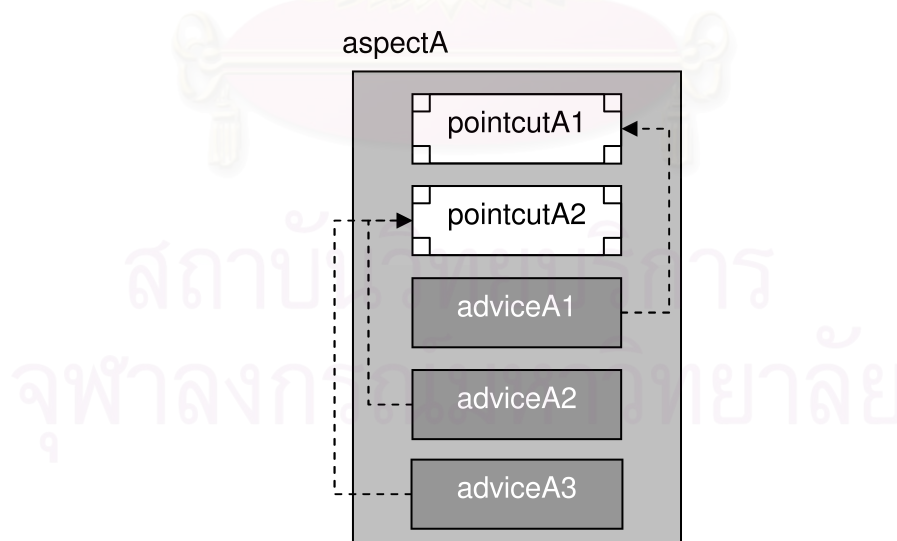


Figure 3.6: The characteristic of the *various concerns* bad smell.

Solution: Since the *various concerns* consist of at least two unrelated concerns which are not correspond to separation of concerns, unrelated concerns should be extracted and moved into their own aspects by applying refactoring procedure as follows:

1. Create a new aspect in order to include another concern of the old one by using *Create Empty Aspect* [12]. The amount of new created aspects is up to the number of advices which refer to the same pointcut.
2. Since all concerns in an aspect share the same pointcut, it is reasonable to apply XPI in order to collect the shared pointcut. Thus, a new aspect as XPI is created by using *Create Empty Aspect* [12].
3. Move a shared pointcut into XPI by using *Move Named Pointcut* [12].
4. Move another concern such as its advice into the new aspect by using *Move Advice* [12].

As aspect *aspectA* could be the candidate of the *various concerns* kind, the above refactoring procedure is applied and the result is illustrated in Figure 3.7. Concerns are more clearly separated.

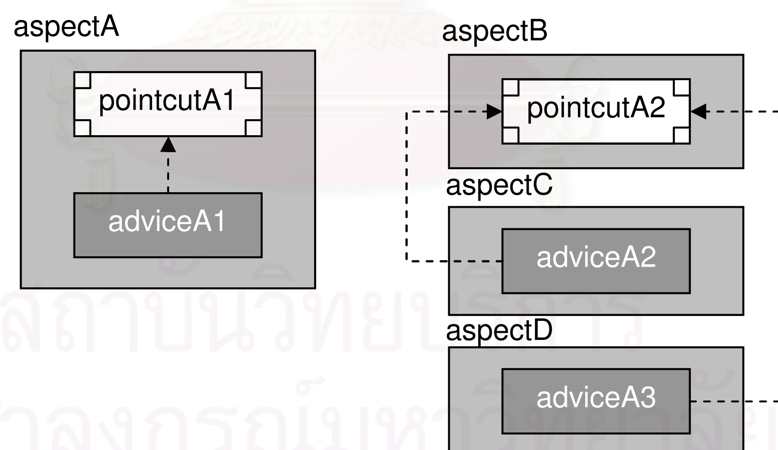


Figure 3.7: After eliminating the *various concerns* bad smell.

3.2.4 Identical Role

Definition: Members of intercepted classes, which inherit from the same class or interface through the inter-type declaration mechanism, are introduced.

Impact: An aspect intercepts control-flow of the base code sometimes requires members of base code to use in execution of the aspect's function. In some cases, there are members of classes, which are newly related by ancestor through the inter-type declaration mechanism, introduced into an aspect. Resulting in the duplication of code of introduced members and preventing them from being reused. Duplication of code further occurs when those introduced members are called or accessed in an aspect. Coupling between an aspect and the affected classes are also unnecessarily increased.

Figure 3.8 illustrates an example of *identical role*. Aspect *aspectA* declares class *classD* to be a parent of class *classB* and class *classC*. Also, attribute *attributeD1* and method *methodD1*, which are members of class *classD*, are introduced to both of class *classB* and class *classC* in aspect *aspectA*. Hence, aspect *aspectA* is considered to be the *identical role.*

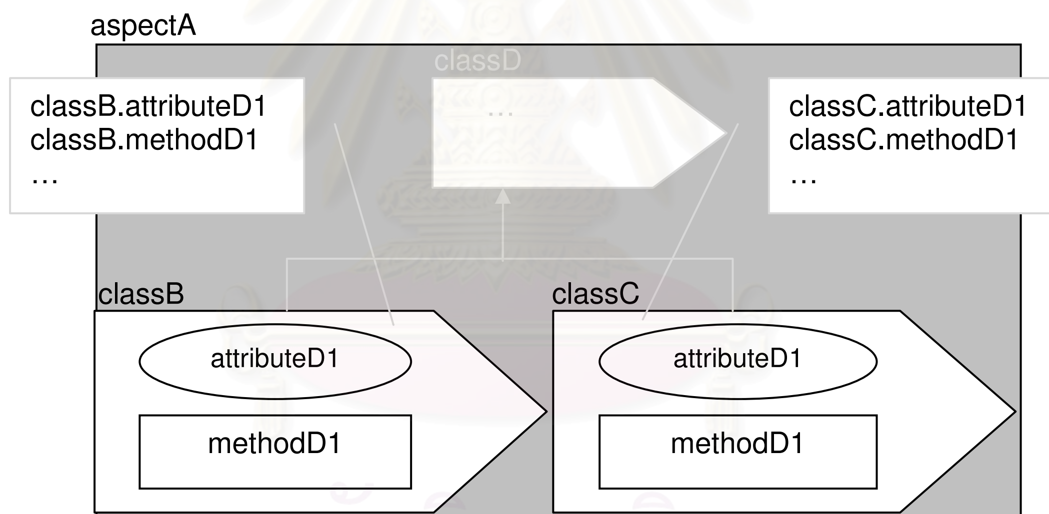


Figure 3.8: The characteristic of the *identical role* bad smell.

Solution: Since the *identical role* introduces unnecessary interaction coupling between an aspect and classes and duplication of codes, all classes, which are introduced the same members in an aspect, should be formed with a representative, and then all references should be changed from those classes to the representative by applying refactoring procedure as follows:

1. Create an inner marker interface to represent all suspected classes by using *Generalize Target Type with Marker Interface* [14].

2. Since the members of suspected classes are not declared by the marker interface, *Extend Marker Interface with Signature* [14] is used to extend them with that signature.

As aspect *aspectA* could be the candidate of the *identical role* kind, the above refactoring procedure is applied and the result is illustrated in Figure 3.9. The interaction coupling and also duplication of codes are decreased.

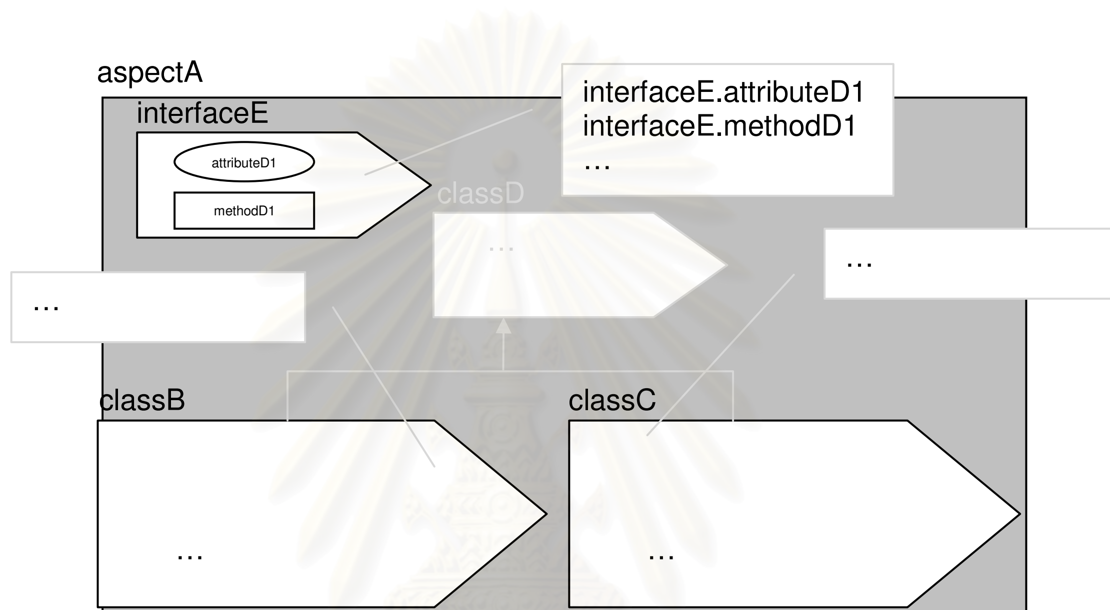


Figure 3.9: After eliminating the *identical role* bad smell.

3.2.5 Junk Material

Definition: An aspect or pointcut is not used (except abstract aspect and abstract pointcut).

Impact: Unused aspect and unused pointcut are unnecessary code in program. It might be created by any reasons and results in increasing the needless entities and size in program.

Figure 3.10 illustrates an example of *junk material*. Aspect *aspectA* consists of pointcut *pointcutA1* and pointcut *pointcutA2*. Pointcut *pointcutA2* is referred to by advice *adviceB2*. Hence, this aspect is considered to be the *junk material.*

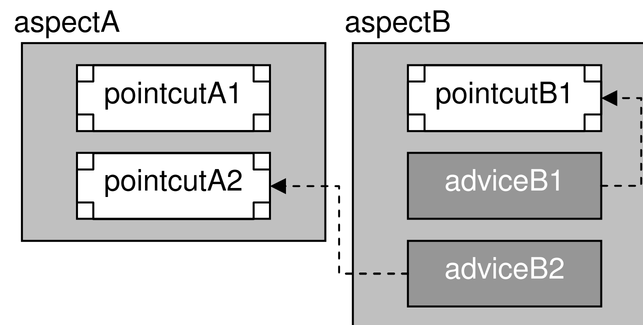


Figure 3.10: The characteristic of the *junk material* bad smell.

Solutions: Since the junk material is unnecessary code in program, this code should be deleted by applying refactoring procedure as follows:

Aspect:

In the case of an empty aspect or an aspect which consists of unreferred pointcuts:

1. Delete the aspect by using *Delete Unreferenced and Empty Aspect* [12].

In the case of pointcuts, which are defined in the aspect, are referred to by advices of other aspects, we can apply refactoring methods in two ways:

- If there is a pointcut which is referred to by advices of another aspect:
    1. Move such pointcut to the aspect which refers to it by using *Move Named Pointcut* [12].
    2. Delete the rest of pointcuts which are not referred to by other aspect by using *Delete Unreferenced Named Pointcut* [12].
    3. Delete empty aspect by using *Delete Unreferenced and Empty Aspect* [12].
- If there is a pointcut which is referred to by advices of many aspects:
    1. Keep such pointcut and change the aspect to be XPI.
    2. Delete the rest of pointcuts which are not referred to by other aspect by using *Delete Unreferenced Named Pointcut* [12].

Pointcut:

1. Delete unreferred pointcut by using *Delete Unreferenced Named Pointcut* [12].

As aspect *aspectA* could be the candidate of the *junk material* kind, the above refactoring procedure is applied and the result is illustrated in Figure 3.11. Unnecessary code is eliminated.

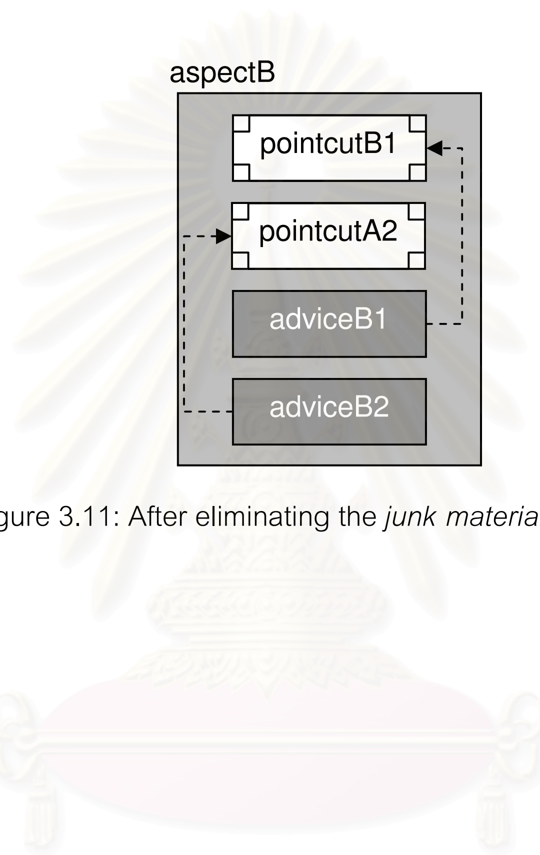


Figure 3.11: After eliminating the *junk material* bad smell.

CHAPTER IV

BAD-SMELL METRICS

This chapter presents metrics and thresholds, which are used to support for detecting AO bad smells. AO bad smells stated here cover both of our bad smells and Piveta's bad smells. For each AO bad smell, metrics and thresholds are presented according to the following template:

Metric: A name of a metric for detecting the bad-smell kind.

Definition: A definition of the metric.

Threshold: The range of measured value, which is suspected to reveal the bad-smell kind.

4.1 Borrowed Pointcut

Metric:

- Number of Non-Subaspect Advices refers to a Pointcut of an aspect (*NNSAdP*)

Definition:

- *NNSAdP* is the total number of advices of the aspects of which are not subaspects refer to a given pointcut.

Threshold: $NNSAdP_{P_i} > 0$

Given:

P are all pointcuts in an aspect.

P_i is a given pointcut.

i equals to 1,…,n, where n is the total number of P.

In order to emphasize that pointcut *pointcutA1* in Figure 3.2 of Chapter III (p.23) contains the *borrowed pointcut*, *NNSAdP* is applied to pointcut *pointcutA1*. The *NNSAdP* of pointcut *pointcutA1* is equal to 2 which is on the threshold. Consequently, pointcut *pointcutA1* is of the kind *borrowed pointcut*.

4.2 Duplicated Pointcut

Metric:

- Set of the corresponding Join points of a Pointcut (*SJP*)

Definition:

- *SJP* is the set of join points, which is attached with pointcut type, corresponding to a given pointcut.

Threshold: $SJP_{P_i} \cap SJP_{P_j} = SJP_{P_i}$

$SJP_{P_i} \cap SJP_{P_j} = SJP_{P_j}$, which $P_i, P_j \in P$ and $P_i \neq P_j$

Given:

P are all pointcuts in the software.

P_i is a given pointcut.

P_j are other pointcuts in the software.

i, j equal to 1,…,n, where n is the total number of P.

In order to emphasize that pointcut *pointcutA1* and pointcut *pointcutB1* in Figure 3.4 of Chapter III (p.24) contain the *duplicated pointcut*, *SJP* is applied to pointcut *pointcutA1* and pointcut *pointcutB1*. The *SJP* of pointcut *pointcutA1* and pointcut *pointcutB1* are shown below:

$$SJP_{pointcutA1} = \begin{Bmatrix} execution(classC.methodC1), \\ execution(classC.methodC2), \\ execution(classD.methodD1) \end{Bmatrix}$$

$$SJP_{pointcutB1} = \begin{Bmatrix} execution(classC.methodC1), \\ execution(classC.methodC2), \\ execution(classD.methodD1) \end{Bmatrix}$$

Both pointcuts collect the same set of join points in base code and are the same type. It corresponds to the threshold. Consequently, pointcut *pointcutA1* and pointcut *pointcutB1* are of the kind *duplicated pointcut*s.

4.3 Various Concerns

Metric:

- Number of Advices of an Aspect refer to a Pointcut (*NAdAsP*)

Definition:

- *NAdAsP* is the total number of the same kind advices (either before or after) in an aspect referring to a given pointcut.

Threshold: $NAdAsP_{P_i} > 1$

Given:

P is all pointcuts in an aspect.

P_i is a given pointcut.

i equals to 1,…,n, where n is the total number of P.

In order to emphasize that pointcut *pointcutA2* in Figure 3.6 of Chapter III (p.26) contains the *various concerns*, *NAdAsP* is applied to pointcut *pointcutA2*. The *NAdAsP* of pointcut *pointcutA2* is equal to 2 which is on the threshold. Consequently, pointcut *pointcutA2* is of the kind *various concerns*.

4.4 Identical Role

Metric:

- Set of the introduced members of a Class in an Aspect (*SCAs*)

Definition:

- *SCAs* is the set of introduced members of a given class, which declared to be an inherited class of a class or an interface, in an aspect.

Threshold: $SCAs_{C_i} \cap SCAs_{C_j} = SCAs_{C_i}$

$SCAs_{C_i} \cap SCAs_{C_j} = SCAs_{C_j}$, which $C_i, C_j \in C$ and $C_i \neq C_j$

Given:

C are all classes, which declared to be an inherited class of a class or an interface, in an aspect.

C_i is a given class.

C_j are other classes, which declared to be an inherited class of a class or an interface, in an aspect.

i, j equal to 1,…,n, where n is the total number of C.

In order to emphasize that aspect *aspectA* in Figure 3.8 of Chapter III (p.28) contains the *identical role*, *SCAs* is applied to class *classB* and class *classC*. The *SCAs* of class *classB* and class *classC* are shown below:

$$SCAs_{classB} = \{attributeD1, methodD1\}$$
$$SCAs_{classC} = \{attributeD1, methodD1\}$$

Both classes are introduced the same members and thus corresponds to the threshold. Consequently, aspect *aspectA* is of the kind *identical role*.

4.5 Junk Material

Metric:

- Number of Pointcuts defined in Aspect (*NPAs*)
- Number of Advices in Aspect (*NAdAs*)
- Number of Introductions in Aspect (*NIAs*)
- Number of Members in Aspect (*NMAs*)
- Number of Other Aspects refer to a Pointcut (*NOAsP*)
- Sum of NOAsP (*SNOAsP*)
- Number of Advices refer to a Pointcut (*NAdP*)

Definition:

- *NPAs* is the total number of pointcuts defined in an aspect.
- *NAdAs* is the total number of advices in a given aspect.
- *NIAs* is the total number of introductions in a given aspect.
- *NMAs* is the total number of attributes and methods in a given aspect.
- *NOAsP* is the total number of other aspects referring to pointcut defined in a given aspect.
- *SNOAsP* is the sum of *NOAsP* of all pointcuts defined in a given aspect.
- *NAdP* is the total number of advices referring to a given pointcut.

Threshold:

Aspect:

- $NPAs > 0$ and $NAdAs = 0$ and $NIAs = 0$ and $NMAs = 0$ and $SNOAsP \geq 0$ or
- $NPAs = 0$ and $NAdAs = 0$ and $NIAs = 0$ and $NMAs = 0$

Pointcut:

$$NAdP = 0$$

Description:

1. If $NPAs = 0$ and $NAdAs = 0$ and $NIAs = 0$ and $NMAs = 0$, an empty aspect is found.
2. If $NPAs > 0$ and $NAdAs = 0$ and $NIAs = 0$ and $NMAs = 0$ and $SNOAsP = 0$, an aspect which consists of unreferred pointcuts is found.
3. If $NPAs > 0$ and $NAdAs = 0$ and $NIAs = 0$ and $NMAs = 0$ and $SNOAsP > 0$, an aspect consists of pointcuts which are referred to by advices of other aspects is found.

To better understand these thresholds, solution's details of *junk material* bad smell in Chapter III (p.30) describes possible cases of the *junk material*.

In order to emphasize that aspect *aspectA* in Figure 3.10 of Chapter III (p.29) contains the *junk material*, *NPAs, NAdAs, NIAs, and SNOAsP* are applied to aspect *aspectA*. The *NPAs, NAdAs, NIAs, NMAs, and SNOAsP* of aspect *aspectA* are equal to 2, 0, 0, 0, and 1, respectively. Those measured values are on the threshold. Consequently, aspect *aspectA* is of the kind *junk material*.

4.6 Anonymous Pointcut Definition

Metric:

- Set of Unnamed Pointcuts defined in an Aspect (*SUPAs*)

Definition:

- *SUPAs* is the set of unnamed pointcuts defined in an aspect.

Threshold: $SUPAs \neq \emptyset$

In order to emphasize that the pointcut of advice *adviceA1* in Figure 2.12 of Chapter II (p.16) contains the *anonymous pointcut definition*, *SUPAs* is applied to aspect *aspectA*. The *SUPAs* of aspect *aspectA* is shown below:

$$SUPAs = \{anonymous_pointcut_of_adviceA1\}.$$

Such measured value is on the threshold. Thus, such pointcut of advice *adviceA1* is of the kind *anonymous pointcut definition*.

4.7 Feature Envy

Metric:

- Number of Class-defined Pointcuts in an Aspect (*NPCAs*)

Definition:

- *NPCAs* is the total number of class-defined pointcuts which are referred to by advices of a given aspect.

Threshold: $NCPAs > 0$

In order to emphasize that aspect *aspectB* in Figure 2.14 of Chapter II (p.17) contains the *feature envy*, *NPCAs* is applied to aspect *aspectB*. The *NPCAs* of aspect *aspectB* is equal to 1 which is on the threshold. Consequently, aspect *aspectB* is of the kind *feature envy*.

4.8 Abstract Method Introduction

Metric:

- Number of introduced Abstract Methods in an Aspect (*NAMA*)

Definition:

- *NAMA* is the total number of abstract methods introduced in an aspect.

Threshold: $NAMA > 0$

In order to emphasize that aspect *aspectA* in Figure 2.16 of Chapter II (p.19) contains the *abstract method introduction*, *NAMA* is applied to aspect *aspectA*. The *NAMA* of aspect *aspectA* is equal to 1 which is on the threshold. Consequently, aspect *aspectA* is of the kind *abstract method introduction*.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER V

BAD-SMELL VALIDATION

This chapter shows the results from validating the proposed bad-smell kinds and the metrics including their thresholds. There are three different kinds of software used for this validation, tutorial software (Telecom and Specewar), academic software (AspectTetris), and open-source software (AJHotDraw).

Telecom and Spacewar [24] was implemented by Xerox Corporation as exploring AspectJ examples in Eclipse [25]. Those AspectJ examples provide illustrative source code to teach the users on the development of AO programs using the language. There are ten classes and three aspects in Telecom simulation. Spacewar consists of twenty-two classes, five aspects, and four inner aspects in classes.

AspectTetris [26] is the game Tetris made in AspectJ. It is implemented by Evertsson as a part of the Advanced Software Engineering course at Blekinge Institute of Technology, Sweden. There are sixteen classes, one interface, and eight aspects in AspectTetris.

AJHotDraw [23] is an AO refactoring of JHotDraw, a relatively large and well-designed open-source Java framework for technical and structured 2D graphics. AJHotDraw composes of three hundred and fifty classes, fifty interfaces, and ten aspects.

This chapter is structured into three parts. The first part illustrates the results of validating the bad-smell metrics and their thresholds. The second part shows the results of validating our defined bad smells. The end of this chapter discusses the results of the research.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

5.1 Bad-Smell Metric and Threshold Validation

Bad-smell metrics and their thresholds are validated in order to ensure that they can be used to specify bad smells in AO programs. To validate them, we consider the measured values from bad-smell metrics applied after removing the bad smells from AO

programs. If the results are not within thresholds, then bad-smells metrics and their thresholds is utilizable.

To apply bad-smell metrics and their thresholds, Appendix C shows an example of software code before and after refactoring its structure.

5.1.1 Applying Bad-Smell Metrics and Detecting Bad Smells before Eliminating Bad Smells

The measured values of all metrics apply before removing bad smells in all samples are summarized in Appendix D.

In Telecom software (Table D.1, p.79), for example, metric threshold indicates that pointcut *endTiming* of aspect *Timing* is of the kind *borrowed pointcut*. Also, one pointcut of aspect *Billing*, two pointcuts of aspect *TimerLog*, and one pointcut of aspect *Timing* are suspected to be the *anonymous pointcut definition*. Besides, aspect *Billing* is indicated by the metric threshold that it contains the *abstract method introduction* bad smell.

In Spacewar software (Table D.3 – Table D.5, p.80 – p.82), aspect *EnsureShipAlive* is indicated by threshold to be the *feature envy* bad smell. Pointcut *guiInit*, pointcut *deleteLines*, and pointcut *newGame* in AspectTetris (Table D.9 – Table D.12, p.86 – p.89) software could be candidates of the *duplicated pointcut* bad smells. Aspect *FigureSelectionObserverRole* is of the kind *identical role* in AJHotDraw software (Table D.17 – Table D.18, p.94 – p.95).

The results presented in Table 5.1 are the number of existences of bad smells which are specified by our metric thresholds.

Various concerns bad smell and *junk material* bad smell however, are not found in the tested software samples. It could not be concluded whether the bad-smell metric and their threshold can indicate these bad smells in AO software, but all software samples possibly do not have an aspect which corresponds to the characteristics of *various concerns* and *junk material*.

Table 5.1: Bad smells in four software samples before eliminating bad smells.

Bad smell	Numbers of existences			
	Telecom	Spacewar	AspectTetris	AJHotDraw
Borrowed pointcut	1	0	0	0
Duplicated pointcut	0	0	3	0
Anonymous pointcut definition	4	7	2	2
Feature envy	0	1	0	0
Various concerns	0	0	0	0
Identical role	0	0	0	1
Abstract method introduction	1	0	0	0
Junk material	0	0	0	0

5.1.2 Applying Bad-Smell Metrics and Detecting Bad Smells after Eliminating Bad Smells

The measured values of all metrics apply after removing bad smells in all samples are also summarized in Appendix D.

With an exception of *abstract method introduction* bad smell, after removing all candidates of AO bad smells and applying bad-smell metrics, it can be observed that all measured values are not within thresholds. The reason of unchanging of measured values of *NAMA* is that we do not suggest removing kind of bad smell from code. Hence, our bad-smell metrics and thresholds are utilizable.

The results presented in Table 5.2 are the number of existences of bad smells which are specified by our metric thresholds after eliminating bad smells.

5.2 Bad-Smell Validation

Our bad smell kinds are validated for the purpose of examining that the defined bad-smell kinds precisely affect the quality of software. The means to validate our bad smells is to compare the results of coupling metrics before and after removing those bad smells from AO software samples. If the results are in the way that coupling is reduced, then the defined bad-smell kinds is able to indicate design flaws in AO

software. The quality attribute metrics, which are used in this thesis, are the coupling metrics proposed by Zhao [11]. The coupling metrics are detailed in Appendix B. Table 5.3 – Table 5.6 shows the measured values of Zhao's metrics before and after eliminating bad smells in four software samples.

Table 5.2: Bad smells in four software samples after eliminating bad smells.

Bad smell	Numbers of existences			
	Telecom	Spacewar	AspectTetris	AJHotDraw
Borrowed pointcut	0	0	0	0
Duplicated pointcut	0	0	0	0
Anonymous pointcut definition	0	0	0	0
Feature envy	0	0	0	0
Various concerns	0	0	0	0
Identical role	0	0	0	0
Abstract method introduction	1	0	0	0
Junk material	0	0	0	0

From the results in Table 5.3 – Table 5.6, it can be observed that only couplings particularly related to intertype-class dependence (*IC*), pointcut-class dependence (*PC*), and pointcut-method dependency (*PM*) are decreased. The reason is that all kinds of AO bad smells found in these software samples i.e. *borrowed pointcut*, *duplicated pointcut*, *anonymous pointcut definition*, *feature envy*, *identical role*, and *abstract method introduction* are related to intertype-declaration and pointcut. More specifically, *identical role* and *abstract method introduction* are related to intertype-declaration and the rest of found bad-smell kinds are related to pointcut. Therefore, *IC*, *PC*, and *PM* are decreased after eliminating those kinds of bad smells.

For example, the *IC* of aspect *FigureSelectionObserverRole* of AJHotDraw software is decreased from 5 to 1 that is because aspect *FigureSelectionObserverRole* is considered to be *identical role* bad smell. The *PC* and *PM* of all bad-smell candidates i.e. aspect *Billing*, aspect *Timing*, aspect *EnsureShipIsAlive*, aspect *Counter*, aspect *GameInfo*, aspect *Levels*, aspect *Menu*, and aspect *NextBlock* are decreased. However,

the *IC* of aspect *Billing* is not decreased even it is a candidate to the *abstract method introduction* bad smell. That is because this kind of bad smell is not harmful and unnecessary to be eliminated from software code.

As coupling are decreased after removing these kinds of AO bad smells, our defined bad-smell kinds can preliminarily indicate some kinds of design flaws in AO software.

5.3 Discussion

From the results of our research, developer can apply our bad-smell kinds to avoid these programming patterns in implementation phase of AO software development, since our kinds of bad smells exactly affect coupling of software. For instance, a named pointcut can possibly be reused by advices, which are not the same kind, in an aspect. Pointcuts, which collect the same set of join points and are of the same type, can be combined for the purpose of reusability by supporting of XPI to avoid interaction coupling among aspect. Also, an advice should refer to a pointcut defined in an aspect or a superaspect of the advice. An aspect should cover only one concern. Even developer errs from avoiding these bad-smell kinds, our bad-smell metrics can be used to support for detecting kinds of bad smells. In addition, although AOP supports developer for separation of concerns, coupling should be considered cautiously.

The Zhao's metrics used in this thesis mainly consider the degree of interdependence among aspects and classes. However, the interdependence among aspects should also be considered. Unfortunately, there is none of coupling metrics thoroughly details on such kind of interdependence at this moment.

จุฬาลงกรณ์มหาวิทยาลัย

Table 5.3: Measured values of Zhao's metrics in Telecom.

Software: Telecom																		
Aspect	*AtC*		*AC*		*IC*		*MC*		*PC*		*AM*		*IM*		*MM*		*PM*	
	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After
Billing	0	0	3	3	0	0	2	2	1	0	3	3	0	0	0	0	4	3
TimerLog	0	0	2	2	0	0	0	0	0	0	2	2	0	0	0	0	2	2
Timing	0	0	2	2	0	0	1	1	1	0	6	6	1	1	0	0	2	1

Table 5.4: Measured values of Zhao's metrics in Spacewar.

Software: Spacewar																		
Aspect	*AtC*		*AC*		*IC*		*MC*		*PC*		*AM*		*IM*		*MM*		*PM*	
	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After
Coordinator	0	0	0	0	0	0	11	11	0	0	4	4	0	0	39	39	0	0
Debug	0	0	9	9	0	0	0	0	0	0	37	37	0	0	0	0	58	58
EnsureShipIsAlive	0	0	1	1	0	0	0	0	1	0	1	1	0	0	0	0	3	0
GameSynchronization	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2
RegistrySynchronization	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table 5.5: Measured values of Zhao's metrics in AspectTetris.

Software: AspectTetris																		
Aspect	*AtC*		*AC*		*IC*		*MC*		*PC*		*AM*		*IM*		*MM*		*PM*	
	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After
Counter	1	1	0	0	0	0	0	0	0	0	5	5	0	0	0	0	5	3
DesignCheck	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	92	92
GameInfo	1	1	0	0	0	0	0	0	0	0	6	6	0	0	0	0	1	0
Levels	2	2	3	3	0	0	0	0	3	3	5	5	0	0	0	0	2	0
Menu	4	4	1	1	0	0	1	1	1	1	16	16	0	0	6	6	2	1
NewBlocks	0	0	0	0	0	0	0	0	0	0	3	3	0	0	0	0	4	4
NextBlock	1	1	0	0	0	0	0	0	0	0	9	9	0	0	0	0	2	1
TestAspect	0	0	0	0	0	0	0	0	0	0	2	2	0	0	0	0	1	1

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table 5.6: Measured values of Zhao's metrics in AJHotDraw.

Software: AJHotDraw																		
Aspect	*AtC*		*AC*		*IC*		*MC*		*PC*		*AM*		*IM*		*MM*		*PM*	
	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After
CmdCheckViewRef	0	0	1	1	0	0	0	0	0	0	4	4	0	0	0	0	10	10
FigureSelectionObserverRole	0	0	0	0	5	1	0	0	0	0	0	0	0	0	0	0	0	0
FigureSelectionSubjectRole	0	0	2	2	4	4	2	2	2	2	4	4	5	5	0	0	2	2
PersistentAttributeFigure	0	0	0	0	2	2	0	0	0	0	0	0	9	9	0	0	0	0
PersistentCompositeFigure	0	0	0	0	2	2	0	0	0	0	0	0	9	9	0	0	0	0
PersistentDrawing	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PersistentFigure	0	0	0	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0
PersistentImageFigure	0	0	0	0	2	2	0	0	0	0	0	0	14	14	0	0	0	0
PersistentTextFigure	0	0	0	0	2	2	0	0	0	0	0	0	22	22	0	0	0	0
SelectionChangedNotification	0	0	2	2	0	0	0	0	3	3	2	2	0	0	0	0	5	5

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER VI

CONCLUSION AND FUTURE WORK

This chapter concludes the research work and presents some directions for the future work. Limitations of our work are also detailed.

6.1 Conclusion

Since new notions and the different ways of thinking of aspect-orientation are emerging, they perhaps introduce different kinds of design flaws and introduce interaction coupling between aspects and classes. Hence, defining bad-smell kinds hidden in AO software as indicators to identify possibly anomalies are required. This thesis defines five kinds of specific AO bad smells affect software coupling namely, *borrowed pointcut*, *duplicated pointcut*, *various concerns*, *identical role* and *junk material*. Other three kinds of AO bad smells i.e. *anonymous pointcut definition*, *feature envy*, and *abstract method introduction*, which are proposed by Piveta [16], are further studied. The metrics which correspond to the bad-smell characteristics are designed to support for detecting our five kinds of bad smells and Piveta's bad smells in AO programs. In order to indicate which particular fraction of code contains the bad smell, the bad-smell thresholds are specified. The refactoring methods, which map to the bad-smell eliminating solutions, are also suggested.

The bad-smell kinds and their metrics are validated through four AO software samples i.e. Telecom, Spacewar, AspectTetris, and AJHotDraw. The bad-smell kinds are validated by comparing the results of applying coupling metrics before and after removing these kinds of bad smells from AO programs. Otherwise, the bad-smell metrics are validated by checking the measured values of these metrics after removing bad smells.

The designed bad-smell metrics indicate that there are *borrowed pointcut*, *duplicated pointc*ut, *anonymous pointcut definition*, *feature envy*, *abstract method introduction*, *identical role*, and *junk material* in software samples. After eliminating these bad smells, the coupling is decreased. We can conclude that the bad-smell metrics

could be used to indicate AO bad smells in software. Also, the defined bad-smell kinds precisely affect coupling of AO software.

6.2 Limitation

1. The *duplicated pointcut* threshold determined in this work is to show a simple case of this bad smell kind i.e. the equivalence of set. The duplicated pointcut thus neither covers the case of overlapping of set nor is-a-subset.
2. In general, a developer perhaps implements an advice to cover more than one concern, which is considered to be bad programming practice. To avoid this kind of bad practice, a developer should implement an advice to cover only one concern. The *various concerns* bad smell is based on the later practice, thus this bad smell kind can be found only when an advice covers a concern.
3. AOP introduces three types of coupling between modules e.g. coupling among aspects, coupling among classes, and coupling among aspects and classes. Zhao's metrics used in the bad smell validation covers only the degree of interdependence among aspects and classes.
4. There are no metrics, which thoroughly detail on other internal quality attributes such as cohesion and complexity, used to analyze possible impact of AO bad smells on those internal attributes deeply.
5. *Various concerns* bad smell and *junk material* bad smell are not found in the tested software samples. The reason is that all software samples possibly do not have an aspect which corresponds to the characteristics of these bad smell kinds.

6.3 Future Work

1. As mentioned many times that AOP involves new notions and the different ways of thinking. Besides our defined bad-smell kinds, AOP perhaps introduces further kinds of AO bad smells. For this reason, other kind of AO bad smells should be defined.

2. According to the software quality, there are several approaches to improve the software quality. Hence, other proper approaches should be considered in order to remove the bad smells.
3. Our AO bad smells, bad-smell metrics, and bad-smell metric thresholds should be validated with other software samples for more reliability and correctness.
4. In order to validate the bad-smell kinds, other quality attributes such as cohesion, complexity, separation of concerns, and size should be applied to analyze the impact of AO bad smells to those quality attributes.
5. All defined bad-smell kinds are specific to AspectJ programming language. In AOP, there are several programming languages to support the AOP paradigm. The bad-smell kinds which can generally be found in several programming languages, should be defined. Furthermore, the bad-smell kinds, which are specific to individual programming language, should be defined.
6. An automated tool for refactoring the codes could be constructed to support programmers in the improvement execution subprocess of the refactoring process.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

REFERENCES

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M. and Irwin, J. Aspect-Oriented Programming. Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). Finland, 1997.

[2] Eder, J., Kappel, G. and Schrefl, M. Coupling and Cohesion in Object-Oriented Systems. Proceedings of Conference on Information and Knowledge Management. Baltomore, USA, 1992.

[3] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[4] Dijkstra, E. W. A Discipline of Programming. Prentice Hall, 1976.

[5] Laddad, L. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., 2003.

[6] Opdyke, W. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Framworks. Ph.D. Thesis University of Illinois at Urbana-Champaign, 1992.

[7] Kataoka, Y., Imai, T., Andou, H. and Fukaya, T. A Quantitative Evaluation of Maintainability Enhancement by Refactoring. Proceedings of International Conference on Software Maintenance (ICSM'02). Montre'al, Canada, 2002.

[8] Mens, T. and Tourwe', T. A Survey of Software Refactoring. IEEE Transactions on Software Engineering. 30, 2(February 2004): 126-139.

[9] Fenton, N. E. and Pfleeger, S. L. Software Metrics: A Rigorous and Practical Approach. PWS Publishing, 1997.

[10] ISO/IEC: Standard 9126-Software Product Evaluation-Quality Characteristics and Guidelines for Their Use. Geneva, 1995.

[11] Zhao, J. Measuring Coupling in Aspect-Oriented Systems. Proceedings of the 10th International Software Metrics Symposium (Metrics 04). Chicago, USA, 2004.

[12] Runa, S. Refactoring Aspect-Oriented Software. Bachelor Thesis Williams College, 2003.

[13] Hanenberg, S., Oberschulte, C. and Unland, R. Refactoring of Aspect-Oriented Software. Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays). Erfurt, Germany, 2003.

[14] Monteiro, M. P. and Fernandes, J. M. Towards a Catalog of Aspect-Oriented Refactorings. Proceedings of AOSD 05. Chicago, Illinois, USA, 2005.

[15] Piveta, E. K., Hecht, M., Pimenta, M. S. and Price, R. T. Bad Smells em sistemas orientados a aspectos (in portuguese). Proceedings of Brazilian Symposium in Software Engineering (SBES 2005). Uberlandia, Brazil, 2005.

[16] Piveta, E. K., Hecht, M., Pimenta, M. S. and Price, R. T. Detecting Bad Smells in AspectJ. Journal of Universal Computer Science. 12, 7(July 2006): 811-827.

[17] Ceccato, M. and Tonella, P. Measuring the Effects of Software Aspectization. Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE) at Working Conference on Reverse Engineering (WCRE). Delft, Netherlands, 2004.

[18] Chidamber, S. R. and Kemerer A. Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering. 20, 6(June 1994): 476-493.

[19] Coad, P. and Yourdon, E. The Coad/Yourdon method: simplicity, brevity, and clarity—keys to successful analysis and design. Object development methods. SIGS Publications, Inc., 1994.

[20] Hannemann, J. and Kiczales, G. Design Pattern Implementation in Java and AspectJ. Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA2002). Washington, USA, 2002.

[21] Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N. and Rajan, H. Information Hiding Interfaces for Aspect-Oriented Design.

Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13). Lisbon, Protugal, 2005.

[22] Griswold, W. G., Shonle, M., Sullivan, K., Song, Y., Tewari, N., Cai, Y. and Rajan, H. Modular Software Design with Crosscutting Interfaces. IEEE Software, Special Issue on Aspect-Oriented Programming. 23, 1(Jan-Feb 2006): 51-60.

[23] Deursen, A. V., Marin, M. and Moonen, L. AJHotDraw: A showcase for refactoring to aspects. Proceedings of Linking Aspect Technology and Evolution Workshop (LATE), AOSD 2005. Chicago, USA, 2005.

[24] Eclpise Foundation. AJDT – AspectJ Development Tools Project [Computer software]. Available from: http://eclipse.org/ajdt/ [2007, August 31]

[25] Eclpise Foundation. Eclipse Project [Computer software]. Available from: http://eclipse.org/ [2007, August 31]

[26] Evertsson, G. Tetris in AspectJ [Computer program]. Available from: http://www.guzzzt.com/coding/aspecttetris.shtml [2007, August 31]

[27] Briand, L. C., Daly, J. and Wuest, J. A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering. 25, 1(January/February 1999): 91-121.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDICES

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX A

PUBLICATIONS

A.1 International Conferences

1) Srivisut, K. and Muenchaisri, P. Defining and Detecting Bad Smells of Aspect-Oriented Software. Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007). Beijing, China, July 23-27, 2007.
2) Srivisut, K. and Muenchaisri, P. Bad-Smell Metrics for Aspect-Oriented Software. Proceedings of the 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007). Melbourne, Australia, July 11-13, 2007.

A.2 National Conference

1) Srivisut, K. and Muenchaisri, P. Determining Threshold of Aspect-Oriented Software Metrics. Proceedings of the 3rd Joint Conference on Computer Science and Software Engineering (JCSSE 2006). Bangkok, Thailand, June 29-30, 2006.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Defining and Detecting Bad Smells of Aspect-Oriented Software

Komsan Srivisut
Center of Excellence in Software Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Bangkok, Thailand
Komsan.S@Student.chula.ac.th

Pornsiri Muenchaisri
Center of Excellence in Software Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Bangkok, Thailand
Pornsiri.Mu@chula.ac.th

Abstract

Bad Smells are software patterns that are generally associated with bad design and bad programming. They can be removed by using the refactoring technique which improves the quality of software. Aspect-Oriented (AO) software development, which involves new notions and the different ways of thinking for developing software and solving the crosscutting problem, possibly introduces different kinds of design flaws. Defining bad smells hidden in AO software in order to point out bad design and bad programming is then necessary. This paper proposes the definition of new AO bad smells. Moreover, appropriate existing AO refactoring methods for eliminating each bad smell are presented. The proposed bad smells are validated. The results show that after removing the bad smells by using appropriate refactoring methods, the software quality is increased.

1. Introduction

As the traditional Object-Oriented (OO) programming unintently introduces the problem of code scattering and code tangling in software development, AO programming [1] is emerging as the new programming paradigm to solve such problem by separating the crosscutting concerns into their own modules called aspect. Bad smells [2] are design flaws in existing software that should be removed through refactorings. The bad smells themselves do not intend to provide precise criteria for when refactoring should maturely be performed. They rather suggest symptoms indicating something may be wrong in design or code. Decisions for removing the bad smells thus depend on the specific aims of the programmer and the specific state and structure of the code on which he is working. Refactoring [2] is a technique for improving the design of an existing software by changing the internal structure of software, while the behavior of the original software is preserved.

Since new notions and the different ways of thinking are introduced in order to support for identification, modularization, representation, and composition of crosscutting concerns, they perhaps introduce different kinds of design flaws. Therefore, defining bad smells hidden in AO software as indicators to identify possibly anomalies is required. This paper proposes the definition of new AO bad smells. Appropriate existing AO refactoring methods are further presented. The quality attribute of software is also measured as a means to validate the proposed bad smells. The results show that the software quality attribute is increased after removing the bad smells.

The rest of this paper is structured as follows. Section 2 presents the definition of each bad smell, its metric, its threshold, and its appropriate refactoring methods. Section 3 illustrates the validation of bad smells with AO sample software. In Section 4, related works are discussed. Conclusions and future works are given in Section 5.

2. Bad smell definition

2.1. Borrowed pointcut

Definition: A pointcut is referred by advices of the aspects of which are not subaspects.

In Figure 1, the dashed arrows show a reference from advice to pointcut. Advice *adviceB2* and advice *adviceC1* refer to pointcut *pointcutA1* in aspect

aspectA but aspect *aspectA* is not the superaspect of the other aspects. Hence, pointcut *pointcutA1* is considered to be the *borrowed pointcut*.

Metric: Number of Non-Subaspect Advices refers to a Pointcut of an aspect (*NNSAdP*)

Threshold: $NNSAdP > 0$

Solution: Crosscutting programming interface (XPI) [3] is used to reduce this kind of interaction by applying refactoring procedure as follows:

1. Create a new aspect as XPI to collect unrelated aspect pointcuts by using *Create Empty Aspect* [4] and specify this aspect to be public.
2. Move suspected pointcuts to the created aspect by using *Move Named Pointcut* [4] and specify these pointcuts to be public.

As pointcut *pointcutA1* is suspected to be *borrowed pointcut* bad smell, the above refactoring procedure is applied and the result shows that the interaction coupling is reduced as illustrated in Figure 2. Code is flexible and easy to understand and reuse.

2.2. Duplicated pointcut

Definition: Pointcuts collect the same set of joinpoints in base code.

In Figure 3, the dotted arrows show a crosscutting from aspect to class. Pointcut *pointcutA1* and pointcut *pointcutB1*, which are differently defined, are intercepting to the same set of joinpoints in both classes. Hence, pointcut *pointcutA1* and pointcut *pointcutB1* are considered to be the *duplicated pointcut.*

Metric: Set of the corresponding Joinpoints of a Pointcut (*SJP*)

Threshold:

$SJP_{Pi} \cap SJP_{Pj} = SJP_{Pi}$

$SJP_{Pj} \cap SJP_{Pi} = SJP_{Pj}$, which $P_i, P_j \in P$ and $P_i \neq P_j$

Given:

P is all pointcuts in the software.

P_i is a given pointcut.

P_j is other pointcuts in the software.

i, j equal to 1,…,n, where n is the total number of P.

Solution: XPI is used to reduce the duplication of code by applying refactoring procedure as follows:

1. Create a new aspect as XPI to collect duplicated aspect pointcuts by using *Create Empty Aspect* [4] and specify this aspect to be public.
2. Move suspected pointcuts to the created aspect by using *Move Named Pointcut* [4] and specify this pointcut to be public.
3. Delete the redundant pointcuts by using *Delete Named Pointcut* [4].

As pointcut *pointcutA1* and pointcut *pointcutB1* are suspected to be *duplicated pointcut* bad smell, the above refactoring procedure is applied. The result shows that the size of code and interaction coupling are decreased and the reusability is increased as illustrated in Figure 4.

2.3. Various concerns

Definition: A pointcut is referred by more than one advices, which are the same kind (either before advice or after advice), in an aspect.

In Figure 5, pointcut *pointcutA2* is referred by advice *adviceA2* and advice *adviceA3*. Hence, aspect *aspectA* is considered to be the *various concerns.*

Metric: Number of Advices refer to a Pointcut (*NAdP*)

Threshold: $NAdP_{Pi} > 1$

Given:

P is all pointcuts in an aspect.

P_i is a given pointcut.

i equals to 1,…,n, where n is the total number of P.

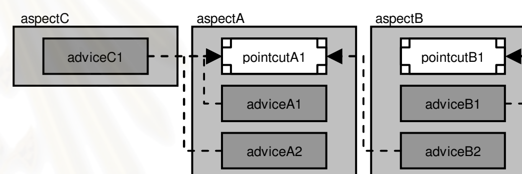


Figure 1. The characteristic of *borrowed pointcut*

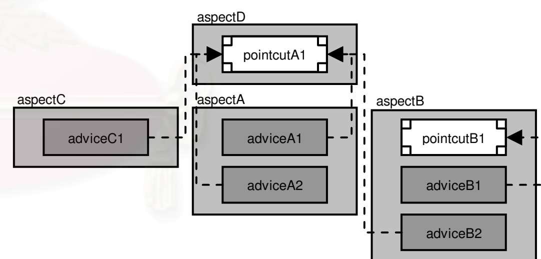


Figure 2. Eliminating *borrowed pointcut*

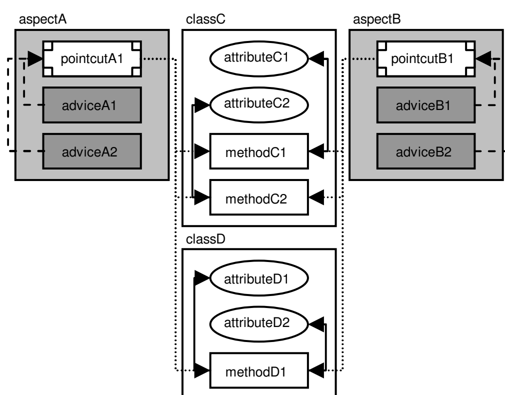


Figure 3. The characteristic of *duplicated pointcut*

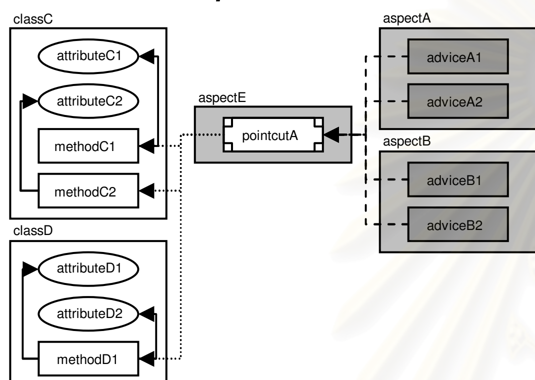


Figure 4. Eliminating *duplicated pointcut*

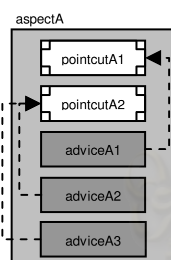


Figure 5. The characteristic of *various concerns*

Solution: Unrelated concerns are extracted and moved into their own aspect by applying refactoring procedure as follows:

1. Create a new aspect in order to include another concern of the old one by using *Create Empty Aspect* [4].
2. Since all concerns in an aspect share the same pointcut, it is reasonable to apply XPI in order to collect the shared pointcut. Thus, a new aspect as XPI is created by using *Create Empty Aspect* [4].
3. Move a shared pointcut into XPI by using *Move Named Pointcut* [4].
4. Move another concern such as their advices into the new aspect by using *Move Advice* [4].

As aspect *aspectA* is suspected to be *various concerns* bad smell, the above refactoring procedure is applied and the result shows that a concern is modularized in its own aspect and the complexity of the aspect is decreased as illustrated in Figure 6.

2.4. Identical role

Definition: Members of intercepted classes, which inherit from the same class or interface, are introduced.

In Figure 7, aspect *aspectA* includes the specific concrete classes: class *classB* and class *classC*. Both specific concrete classes are extended from class *classD*. The inherited members of both classes are duplicated. Hence, this aspect is considered to be the *various concerns.*

Metric: Set of the inherited Classes of a given Type (*SCT*)

Threshold: n(*SCT*) > 1

Solution: All related types are formed with a representative, and then all references are changed from related types to the representative by applying refactoring procedure as follows:

1. Create inner marker interface to represent all inherited classes of a given type by using *Generalize Target Type with Marker Interface* [5].
2. Since marker interface does not declare the members of inherited classes, *Extend Marker Interface with Signature* [5] is used to extend them with that signature.

As aspect *aspectA* is suspected to be *identical role* bad smell, the above refactoring procedure is applied and the result shows that the size of code is decreased and the fragment of code is reused as illustrated in Figure 8.

3. Validation

The proposed bad smells are validated with two sample software named Telecom [6] and AspectTetris [7]. Firstly, a quality attribute i.e., coupling of both samples is measured by using quality metric called Coupling between Modules (CBM). This metric was proposed by Ceccato and Tonella [8]. It revised the well known Chidamber and Kemerer's metric [9] named Coupling Between Objects (CBO) to make it applicable to AO software. Then, the bad smell metrics are used to detect the bad smells in sample software. One *borrowed pointcut* bad smell and three *duplicated pointcut* bad smells are discovered in Telecom and

AspectTetris, respectively. After detecting the bad smells, the appropriate refactoring methods are applied. The quality attribute of sample software after removing the bad smells is measured once again. The measured results of quality metric (both before and after refactoring) are compared to ensure that the quality attribute of sample software is improved as shown in Table 1.

4. Related works

In order to improve AO software, several authors [4, 5, 10-12] proposed refactoring techniques, but specific AO bad smells were still limited. Monteiro and Fernades [5] proposed a collection of AO refactoring cover both the extraction of aspects from OO legacy code and the subsequent tidying up of the resulting aspects. They also reviewed the traditional OO code smells in the light of aspect orientation and proposed some new smells for the detection of crosscutting concerns. In addition, they firstly proposed a new code smell that was specific to aspect named *Aspect Laziness*.

5. Conclusions and future works

This paper proposes four specific AO bad smells i.e., *borrowed pointcut*, *duplicated pointcut*, *various concerns*, and *identical role*. Appropriate AO refactoring methods are selected to eliminate the bad smells. The proposed bad smells are validated through two AO sample software. After eliminating the bad smells found in both samples, their quality is improved. We plan to validate the proposed bad smells with other AO sample software and with other quality attributes. Also we intend to define further AO bad smells, their metrics, and their appropriate refactoring methods. Tool-support for the detection is also our future works.

6. References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," presented at Eiropean Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1 ed: Addison-Wesley, 1999.

[3] W. G. Griswold, M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan, "Modular Software Design with Crosscutting Interfaces," *IEEE Software, Special Issue on Aspect-Oriented Programming*, vol. January/February 2006, 2006.

[4] S. Runa, "Refactoring Aspect-Oriented Software," in *Computer Science*. Williamstown, Massachusetts: WILLIAMS COLLEGE, 2003, pp. 82.

[5] M. P. Monteiro and J. M. Fernandes, "Towards a Catolog of Aspect-Oriented Refactorings," presented at AOSD 05, Chicago, Illinois, USA, 2005.

[6] http://www.eclipse.org/ajdt/.

[7] http://www.guzzzt.com/coding/aspecttetris.shtml.

[8] M. Ceccato and P. Tonella, "Measuring the Effects of Software Aspectization," presented at 1st Workshop on Aspect Reverse Engineering (WARE) at Working Conference on Reverse Engineering (WCRE), Delft, The Netherlands, 2004.

[9] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," presented at IEEE Transactions on Software Engineering, 1994.

[10] A. v. Deursen, M. Marin, and L. Moonen, "Aspect Mining and Refactoring," presented at First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE), University of Waterloo, 2003.

[11] M. Iwamoto and J. Zhao, "Refactoring Aspect-Oriented Programs," presented at 4th AOSD Modeling with UML Workshop, UML'2003, San Francisco, California, USA, 2003.

[12] S. Hanenberg, C. Oberschulte, and R. Unland, "Refactoring of Aspect-Oriented Software," presented at 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), Erfurt, Germany, 2003.

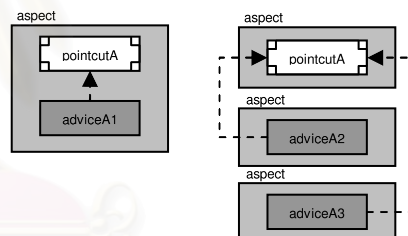


Figure 6. Eliminating *various concerns*

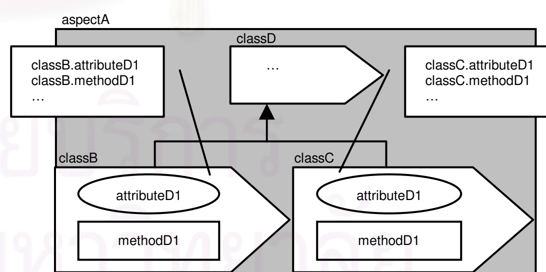


Figure 7. The characteristic of *identical role*

Bad-Smell Metrics for Aspect-Oriented Software

Komsan Srivisut
Software Engineering Laboratory
Center of Excellence in Software Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Bangkok, Thailand
Komsan.S@Student.chula.ac.th

Pornsiri Muenchaisri
Software Engineering Laboratory
Center of Excellence in Software Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Bangkok, Thailand
Pornsiri.Mu@chula.ac.th

Abstract

Aspect-Oriented Programming (AOP) is a new programming paradigm that improves separation of concerns by decomposing the crosscutting concerns in aspect modules. Bad smells are metaphors to describe software patterns that are generally associated with bad design and bad programming of Object-Oriented Programming (OOP). New notions and different ways of thinking for developing Aspect-Oriented (AO) software inevitably introduce bad smells which are specific bad design and bad programming in AO software called AO bad smells. Software metrics have been used to measure software artifact for a better understanding of its attributes and to assess its quality. Bad-smell metrics should be used as indicators for determining whether a particular fraction of AO code contains bad smells or not. Therefore, this paper proposes definition of metrics corresponding to the characteristic of each AO bad smell as a means to detecting them. The proposed bad-smell metrics are validated and the results show that the proposed bad-smell metrics can preliminarily indicate bad smells hidden in AO software.

1. Introduction

One goal of software metrics is to identify and measure the essential parameters that affect software development. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality [1, 2].

Separation of concerns [3] entails breaking down software into distinct parts that overlap in functionality as little as possible. All programming methodologies – including procedural programming and OOP – support some separation and encapsulation of concerns into single entities. For example, procedures, packages, classes and methods all help programmers encapsulate concerns into single entities. Unfortunately, there are some concerns defy these forms of encapsulation calls "crosscutting concerns". For example, a logging strategy necessarily affects every single logged part of the system. Logging thereby crosscuts all logged classes and methods. AOP [4] is the new programming paradigm which attempts to aid programmers in the separation of concerns, specifically crosscutting concerns, as an advance in modularization.

Bad smells [5] are proposed by Beck and Fowler as flaws in existing code that should be removed through refactorings. Such bad smells do not aim to provide precise criteria for when refactorings should maturely be performed. Instead, it suggests symptoms indicating something may be wrong in design or code. Programmers are required to develop their own sense of when a symptom indeed warrants a change. Decisions also depend on the specific aims of the programmer and the specific state and structure of the code on which he is working. AO bad smells, the specific design flaws in AO software, are possibly emerging according to the new notions and the different ways of thinking in developing software. Measuring the characteristics of AO bad-smells in software by using software metrics is essentially applied to determine whether a particular fraction of AO code contains bad smells or not.

This paper proposes AO software metrics suite as indicators to identify bad smells hidden in software. The proposed bad-smell metrics are validated by using them to detect the bad smells in four AO sample software. The results show that the proposed metrics can suggest bad smells in sample software.

The rest of this paper is structured as follows. Section 2 presents the definition of each bad smell detected by the proposed metrics. The metrics along with bad-smell metric thresholds are presented in Section 3. Section 4 illustrates validation of bad-smell metrics with sample AO software and the results. In Section 5, related works are discussed. Conclusions and future works are given in Section 6.

2. AO bad smells

In this section, eight bad smells are detailed, including our five AO bad smells and other three AO bad smells [6]. All bad smells presented here are classified into four pointcut bad smells (*borrowed pointcut*, *duplicated pointcut*, *anonymous pointcut definition*, and *feature envy*), three aspect bad smells (*various concerns*, *identical role*, and *abstract method introduction*) and one multi-entity bad smell (*junk material*). In order to consider the pointcut bad smells (except *anonymous pointcut definition*), *various concerns* bad smell, and *junk material* bad smell, all considered pointcuts should be named according to the characteristics of the bad smells. Hence, unnamed pointcuts considered in the bad smells are assumed as the named pointcuts which are referred by their own advice. AspectJ [7], an extension of the programming language Java, is used as the current primary representative AOP language to define the metrics in this work. AspectJ is the most popular one and already has a large community.

2.1. Borrowed pointcut

Borrowed pointcut bad smell occurs whenever a pointcut is referred by advices of other aspects that are not subaspect of the one it is actually defined. Although this kind of reference reduces the coupling between classes and aspects, it creates the coupling between unrelated aspects. The desired design characteristic of software is to have low interaction coupling and high inheritance coupling [8].

2.2. Duplicated pointcut

Pointcuts that collect the same set of joinpoints in base code are defined as *duplicated pointcut* bad smell. This is a kind of duplicate code that affects the size of code. In general, the bigger size, the more difficult it is to understand the system. The interaction coupling also occurs between class and aspect, since aspect intercepts the execution of class.

2.3. Anonymous pointcut definition [6]

In AspectJ, as pieces of advice are not named, sometimes it is necessary to rely on the pointcut definition to have an idea of the affected points in base code. Also a named pointcut is possibly reused by other advices which affect to the same points in base code. Unnamed pointcut is defined as *anonymous pointcut definition* bad smell.

2.4. Feature envy [6]

Feature envy bad smell is always found in an aspect that uses a class-defined pointcut. In AspectJ, although pointcuts could be defined in aspects and also in classes, it is possible to move the pointcut from the class to the aspect that uses it. This kind of reference increases unnecessarily coupling between modules.

This bad smell resembles to our *borrowed pointcut* bad smell presented above but their pointcut is defined in a class.

2.5. Various concerns

An aspect is marked as *various concerns* bad smell when there are a non 1-to-1 relationship between pointcut and advice. In other words, many advices, which are the same kind (either before advice or after advice), refer to the same pointcut. In general, an aspect modularizes a unique concern. When an aspect has too many pointcuts and advices, it implicitly indicates that there may be more than one unrelated concern in such aspect.

This bad smell is similar to *large aspect* bad smell proposed by Piveta et al. [6]. The difference between *various concerns* bad smell and *large aspect* bad smell is on how to consider the number of concerns in an aspect. *Large aspect* bad smell considers the number of members in an aspect. Piveta et al. determined in [9] that an aspect with ten or more crosscutting members is marked as *large aspect* bad smell.

2.6. Identical role

Specific concrete types, which are closely related in inheritance relationship, result in duplicated code and prevent them from being reused. Also coupling arising

between an aspect and the affected classes are unnecessarily increased. Those concrete types are called *identical role* bad smell.

2.7. Abstract method introduction [6]

Abstract method introduction bad smell occurs when abstract method is added into existing class through the inter-type declaration mechanism of aspect. This introduction forces the programmer to provide concrete implementations to the introduced methods in every affected class and subclasses. This dependency unnecessarily increases the coupling between the aspect and the affected classes.

This kind of bad smell is not harmful and unnecessary to be eliminated from software code, because this kind of dependency occurs according to the generalization of classes. For instance, common introduced methods of subclasses are pulled up into their superclass.

2.8. Junk material

Unused aspect and unused pointcut are unnecessary code in program and are considered to be *junk material* bad smell. They might be created by any reasons and results in increasing the needless entities and size in program.

3. Bad-smell metrics

Fifteen metrics proposed in this paper are six pointcut-level metrics, eight aspect-level metrics and one class-level metric, and are summarized in Table 1.

In order to detect the bad smells in code, the range of measured value of the metric should be specified to reveal the bad smell. The range is called threshold. Table 2 illustrates the threshold of each bad smell.

4. Validation

The proposed bad-smell metrics are validated by using them to detect the bad smell hidden in three different kinds of software, including tutorial software (Telecom and Spacewar [10]), academic software (AspectTetris [11]), and open-source software (AJHotDraw [12]). For clear understanding on measurement and bad-smell detection, Section 4.1 shows an example of measured values of those metrics collected from sample software code. After obtaining the measured values, bad-smell metric thresholds are used to examine whether a piece of AO code contains the bad smells or not. The bad smells found in each sample are shown in Section 4.2.

4.1. Measurement and bad-smell detection

In order to measure the values of all metrics, collecting each metric value is presented through a fragment of code named aspect *Billing* of Telecom software [10] illustrated in Listing 1.

According to Listing 1, the measured value of each metric from aspect *Billing* is summarized in Table 3. All metrics are used in aspect and class level. The rest of the proposed metrics are pointcut-level metrics. As mentioned above, unnamed pointcut is assumed as a named pointcut which is referred by its own advice. The measured value of each metric from unnamed pointcut of aspect *Billing* is summarized in Table 4.

Another pointcut of aspect *Billing* is pointcut *endTiming* which is defined in aspect *Timing*. Thus, point of view for collecting pointcut level metrics is moved to pointcut *endTiming* in aspect *Timing*. The fragment of code of aspect *Timing* [10] is shown in Listing 2. The measured value of each metric from pointcut *endTiming* of aspect *Timing* is summarized in Table 5.

After examining the measured values of metrics in Table 3 with the threshold in Table 2, one *anonymous pointcut definition* bad smell, and one *abstract method introduction* bad smell in aspect *Billing* are discovered, since *NUPAs* and *NAMA* of aspect *Billing* are greater than zero. The number of existences of each bad smell in aspect *Billing* are summarized in Table 6.

The results of bad-smell detection by using the proposed metrics through four sample software are summarized in Section 4.2.

Table 1. The proposed metrics

Level	Metrics
Pointcut	Number of Advices refer to a Pointcut (*NAdP*)
	Number of Advices in Aspect refer to a Pointcut (*NAdAsP*)
	Number of Subaspect Advices refer to an aspect Pointcut (*NSAdP*)
	Number of Non-Subaspect Advices refer to an aspect Pointcut (*NNSAdP*)
	Set of the corresponding Joinpoints of a Pointcut (*SJP*)
	Number of Other Aspects refer to a Pointcut (*NOAsP*)
Aspect	Number of Pointcuts defined in Aspect (*NPAs*)
	Number of Named Pointcuts defined in Aspect (*NNPAs*)
	Number of Unnamed Pointcuts defined in Aspect (*NUPAs*)
	Set of the inherited Classes of a given Type (*SCT*)
	Number of introduced Abstract Methods in an Aspect (*NAMA*)
	Number of Advices in Aspect (*NAdAs*)
	Number of Introductions in Aspect (*NIAs*)
	Sum of NOAsP (*SNOAsP*)
Class	Number of Pointcuts defined in a Class (*NPC*)

Table 2. Threshold of each bad smell

Bad smell	Threshold
Borrowed pointcut	$NNSAdP > 0$
Duplicated pointcut	$SJP_{Pi} \cap SJP_{Pj} = SJP_{Pi}$ $SJP_{Pj} \cap SJP_{Pi} = SJP_{Pj}$, which P_i, $P_j \in P$ and $P_i \neq P_j$ *Given:* P is all pointcuts in the software. P_i is a given pointcut. P_j is other pointcuts in the software. i, j equal to 1,…,n, where n is the total number of P.
Anonymous pointcut definition	$NUPAs > 0$
Feature envy	$NPC > 0$
Various concerns	$NAdP_{Pi} > 1$ *Given:* P is all pointcuts in an aspect. Pi is a given pointcut. i equals to 1,…,n, where n is the total number of P.
Identical role	$n(SCT) > 1$
Abstract method introduction	$NAMA > 0$
Junk material	*Aspect*: $NPAs > 0$ and $NAdAs = 0$ and $NIAs = 0$ and $SNOAsP \geq 0$ $NPAs = 0$ and $NAdAs =$ and $NIAs \geq 0$ *Pointcut*: $NAdP = 0$

```
01   public aspect Billing {
02      declare precedence: Billing, Timing;
03
04      public static final long LOCAL_RATE = 3;
05      public static final long LONG_DISTANCE_RATE = 10;
06
07      public Customer Connection.payer;
08      public Customer getPayer(Connection conn) { return
09   conn.payer; }
10
11      after(Customer cust) returning (Connection conn):
12         args(cust, ..) && call(Connection+.new(..)) {
13         conn.payer = cust;
14      }
15
16      public abstract long Connection.callRate();
17
18      public long LongDistance.callRate() { return
19   LONG_DISTANCE_RATE; }
20      public long Local.callRate() { return LOCAL_RATE; }
21
22      after(Connection conn): Timing.endTiming(conn) {
23         long time = Timing.aspectOf().getTimer(conn).getTime();
24         long rate = conn.callRate();
25         long cost = rate * time;
26         getPayer(conn).addCharge(cost);
27      }
28
29      public long Customer.totalCharge = 0;
30      public long getTotalCharge(Customer cust) { return
31   cust.totalCharge; }
32
33      public void Customer.addCharge(long charge){
34         totalCharge += charge;
35      }
36   }
```

Listing 1. Aspect *Billing* of Telecom software [10]

4.2. Validation with four sample code

The limited space at our disposal in this paper does not allow us to rigorously discuss all bad smells found in all sample code in order to verify that the proposed metrics can accurately be used to detect bad smells hidden in software. Therefore, all bad smells found only in Telecom software are thoroughly discussed. The remaining samples show exclusively the results of detecting bad smell using the proposed metrics.

Table 3. The measured values from aspect *Billing* of Telecom software

Bad-smell metric	Measured value
NPAs	1
NNPAs	0
NUPAs	1
NPC	N/A
SCT	N/A
NAMA	1
NAdAs	2
NIAs	7
SNOAsP	0

Table 4. The measured values from unnamed pointcut of aspect *Billing* of Telecom software

Bad-smell metric	Measured value
NAdP	1
NAdAsP	1
NSAdP	0
NNSAdP	0
SJP	{Connection.new(..), Local.new(..), LongDistance.new(..)}
NOAsP	0

```
01   public aspect Timing {
02
03   …
04      after (Connection c): target(c) && call(void
05   Connection.complete()) {
06         getTimer(c).start();
07      }
08
09      pointcut endTiming(Connection c): target(c) &&
10         call(void Connection.drop());
11
12      after(Connection c): endTiming(c) {
13         getTimer(c).stop();
14         c.getCaller().totalConnectTime += getTimer(c).getTime();
15         c.getReceiver().totalConnectTime += getTimer(c).getTime();
16      }
17   }
```

Listing 2. Aspect *Timing* of Telecom software [10]

Table 5. The measured values from pointcut *endTiming* of aspect *Timing* of Telecom software

Bad-smell metric	Measured value
NAdP	2
NAdAsP	1
NSAdP	0
NNSAdP	1
SJP	{void Connection.drop()}
NOAsP	1

Table 6. Bad smells in aspect *Billing* of Telecom software

Bad smell	Number of existences
Borrowed pointcut	0
Duplicated pointcut	0
Anonymous pointcut definition	1
Feature envy	0
Various concerns	0
Identical role	0
Abstract method introduction	1
Junk material	0

After using our proposed metrics to identify bad smells hidden in Telecom software, it indicates that *borrowed pointcut* bad smell, *anonymous pointcut definition* bad smell, and *abstract method introduction* bad smell exist in Telecom software. The number of existences of each bad smell in Telecom software is summarized in Table 7. *Borrowed pointcut* bad smell is found in aspect *Timimg*. *Anonymous pointcut definition* bad smells are found in all aspects of Telecom software. *Abstract method introduction* bad smell is found in aspect *Billing*.

After that, the proposed metrics are verified by investigating bad smells in program code. After considering Telecom software at the code level, observing that all suspected pointcuts and suspected aspect indicated by the proposed metrics conform to the characteristics of the bad smells mentioned in Section 2. First, aspect *Timing* has a pointcut named *endTiming* (Listing 2, lines 9-10), which is exactly referred by an advice of aspect *Billing* (Listing 1, line 22). Pointcut *endTiming* conforms to the characteristic of *borrowed pointcut* bad smell. Although this reference reduces the coupling between class *Connection* and both of the aspects: aspect *Timing* and aspect *Billing*, it creates the coupling between both aspects. It is interesting to use crosscutting interface (XPI) [13] as an interface to collect such unrelated aspect pointcut.

There are one, two, and one unnamed pointcuts in code of aspect *Billing* (Listing 1, line 12), aspect *TimerLog* (Listing 3, lines 3 and 7), and aspect *Timing* (Listing 2, lines 4-5), respectively. All unnamed pointcuts conform to the characteristic of *anonymous pointcut definition* bad smell. It is possible to give a name to all unnamed pointcuts. In aspect *Billing*, there is certainly an abstract method *callRate* introduced to class *Connection* (Listing 1, line 16). This method conforms to the characteristic of *abstract method introduction* bad smell. It is not necessary to eliminate this kind of bad smell according to the generalization of the members of classes introduced in the aspect.

After investigating those suspects in program code and conforming them with the characteristics of the bad smells, it is confirming that our proposed metrics can be used to detect bad smells in AO software. The rest of samples are verified in the same way and the results are similar. The number of existences of each bad smell in Spacewar, AspectTetris, and AJHotDraw software are summarized in Table 8.

5. Related works

Bad smells and refactorings are closely related, since bad smells can be removed by using the refactoring techniques in order to improve the qualities of software. The prior researches in the light of aspect orientation focused on refactoring techniques. Several authors [14-17] propose refactoring techniques. Iwamoto and Zhao [14] investigate the impact of existing OO refactorings on AO program such as those proposed by Fowler [5]. Their intention is to build a catalog of AOP refactorings, but the information provided about them is limited to the names of the twenty four refactorings. Rura [15] proposes thirty new fundamental AOP-specific refactorings and recasts the existing (OO) refactorings to preserve program behavior in AO code. Composite refactorings, which are built from their fundamental refactorings, are additionally presented in order to aid in the extraction of crosscutting concerns by deploying AOP techniques in existing programs.

Hanenberg et al. [16] introduce a number of new AO refactorings which help to migrate from OO to AO software and to restructure existing AO code. There are three refactorings in order to restructure existing AO code such as, *Extract Advice*, *Extract Introduction* and *Separate Pointcut*. Monteiro and Fernades [17] propose a collection of twenty eight AO refactorings cover both the extraction of aspects from OO legacy code and the subsequent tidying up of the resulting aspects. They also review the traditional OO code smells in the light of aspect orientation and propose some new smells for the detection of crosscutting concerns. In addition, they firstly propose a new code smell that is specific to aspect named *Aspect Laziness* – an aspect that does not carry the full weight of their responsibilities and instead pass the burden to classes.

Piveta et al. [6] defined five bad smells that occur in AO systems i.e. *anonymous pointcut definition*, *large aspect*, *lazy aspect*, *feature envy*, and *abstract method introduction*. They complement their work with algorithms [9] to automatically detect their five proposed bad smells, more specifically those written using AspectJ language.

6. Conclusions and future works

As aspect orientation requires its new notions and the different ways of thinking, it perhaps introduces AO bad smells, the specific design flaws in AO software. In order to detect AO bad smell in software, software metrics corresponded to the characteristic of each bad-smell are possibly used to determine whether a particular fraction of code contains bad smells or not. This paper proposes fifteen AO software metrics for detecting eight bad smells hidden in AO software. There are our five AO bad smells and other three AO bad smells proposed by Piveta et al [6].

The fifteen proposed bad-smell metrics are validated through four AO sample software. The proposed bad-smell metrics preliminarily indicate that there are AO bad smells in all sample software. After investigating all suspected pointcuts and suspected aspects in software code with the proposed metrics, all suspected entities are conforming to the characteristics of the bad smells. It is confirming that the proposed metrics can be used to detect bad smells in AO software. We plan to validate the proposed bad-smell metrics with other AO sample software and also intend to define further AO bad smells, their metrics, and their appropriate refactoring methods.

7. References

[1] E. E. Mills, "Software Metrics," Software Engineering Institute, Pittsburg, PA, USA, SEI Curriculum Module SEI-CM-12-1.1 December 1988.

[2] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: PWS Publishing Company, 1997.

[3] E. W. Dijkstra, *A Discipline of Programming*, 1st ed: Prentice Hall, Inc., 1976.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," presented at Eiropean Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1 ed: Addison-Wesley, 1999.

[6] E. K. Piveta, M. Hecht, M. S. Pimenta, and R. T. Price, "Bad Smells em sistemas orientados a aspectos (in portuguese)," presented at Brazilian Symposium in Software Engineering, SBES 2005, Uberlandia, Brazil, 2005.

[7] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*: Manning Publications Co., 2003.

[8] A. J. Riel, *Object-Oriented Design Heuristics*: Addison-Wesley Professional, 1996.

[9] E. K. Piveta, M. Hecht, M. S. Pimenta, and R. T. Price, "Detecting Bad Smells in AspectJ," *Journal of Universal Computer Science*, vol. 12, 2006.

[10] http://www.eclipse.org/ajdt/.

[11] http://www.guzzzt.com/coding/aspecttetris.shtml.

[12] A. v. Deursen, M. Marin, and L. Moonen, "AJHotDraw: A showcase for refactoring to aspects," presented at Linking Aspect Technology and Evolution Workshop (LATE) AOSD 2005, Chicago, USA, 2005.

[13] W. G. Griswold, M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan, "Modular Software Design with Crosscutting Interfaces," *IEEE Software, Special Issue on Aspect-Oriented Programming*, vol. January/February 2006, 2006.

[14] M. Iwamoto and J. Zhao, "Refactoring Aspect-Oriented Programs," presented at 4th AOSD Modeling with UML Workshop, UML'2003, San Francisco, California, USA, 2003.

[15] S. Runa, "Refactoring Aspect-Oriented Software," in *Computer Science*. Williamstown, Massachusetts: WILLIAMS COLLEGE, 2003, pp. 82.

[16] S. Hanenberg, C. Oberschulte, and R. Unland, "Refactoring of Aspect-Oriented Software," presented at 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), Erfurt, Germany, 2003.

[17] M. P. Monteiro and J. M. Fernandes, "Towards a Catalog of Aspect-Oriented Refactorings," presented at AOSD 05, Chicago, Illinois, USA, 2005.

Table 7. Bad smells in Telecom software

Bad smell	Number of existences
Borrowed pointcut	1
Duplicated pointcut	0
Anonymous pointcut definition	4
Feature envy	0
Various concerns	0
Identical role	0
Abstract method introduction	1
Junk material	0

```
01    public aspect TimerLog {
02
03      after(Timer t): target(t) && call(* Timer.start())  {
04        System.err.println("Timer started: " + t.startTime);
05      }
06
07      after(Timer t): target(t) && call(* Timer.stop()) {
08        System.err.println("Timer stopped: " + t.stopTime);
09      }
10    }
```

Listing 3. Aspect *TimerLog* of Telecom software [10]

Table 8. Bad smells in the rest of sample software

Bad smell	Number of existences		
	Spacewar	AspectTetris	AJHotDraw
Borrowed pointcut	0	0	0
Duplicated pointcut	1	3	0
Anonymous pointcut definition	15	2	2
Feature envy	1	0	0
Various concerns	0	0	0
Abstract method introduction	0	0	1
Identical role	2	0	0
Junk material	0	0	1

Determining Threshold of Aspect-Oriented Software Metrics

Komsan Srivisut[1] and Pornsiri Muenchaisri[2]

[1, 2]Software Engineering Laboratory
Center of Excellence in Software Engineering
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Bangkok, 10330, Thailand

E-mail: [1]Komsan.S@Student.chula.ac.th and [2]Pornsiri.Mu@chula.ac.th

Abstract

Threshold of software metrics can be used as indicators to identify possible anomalies in software. Aspect-Oriented (AO) Programming is a new programming paradigm that solved the crosscutting problem by decomposes the crosscutting concern in aspect module. Establishing the threshold of AO software metrics in order to identify anomalies in AO software is necessary. The Gang-of-Four (GoF) patterns are widely accepted as good design. Metrics extracted from the GoF patterns should be relevant information for preliminary obtaining the threshold. In this paper, we present some metric thresholds, which established from the 23 aspect-based GoF patterns. We also validate the thresholds through 2 AO software examples. The results show that it indicates anomalies in the software example.

Keywords: Threshold, Software Metrics, Software Quality, Aspect-Oriented Programming

บทคัดย่อ

ช่วงของค่ามาตรวัด สามารถใช้เป็นตัวชี้เพื่อระบุสิ่งผิดปกติในซอฟต์แวร์ โปรแกรมเชิงแง่มุมเป็นแบบอย่างการโปรแกรมใหม่ ซึ่งแก้ปัญหาการตัดขวางที่เกิดขึ้น โดยแยกส่วนของหน้าที่ที่ตัดขวางไว้ในแง่มุม การสร้างช่วงของค่ามาตรวัดซอฟต์แวร์เชิงแง่มุม เพื่อระบุสิ่งผิดปกติในซอฟต์แวร์เชิงแง่มุมจึงเป็นสิ่งจำเป็น แบบรูปแกงค์ออฟโฟร์เป็นที่ยอมรับกันอย่างแพร่หลายว่าเป็นการออกแบบที่ดี ค่ามาตรวัดที่ได้จากรูปแบบแกงค์ออฟโฟร์จึงเป็นข้อมูลที่มีความหมายเพื่อการได้มาเบื้องต้นของช่วงของค่ามาตรวัด บทความฉบับนี้ผู้วิจัยจึงนำเสนอช่วงของค่ามาตรวัด ซึ่งสร้างมาจาก 23 แบบรูปแกงค์ออฟโฟร์บนพื้นฐานเชิงแง่มุม นอกจากนี้ผู้วิจัยทำการประเมินช่วงของค่ามาตรวัดด้วย 2 ตัวอย่างซอฟต์แวร์เชิงแง่มุม ผลที่ได้พบว่าช่วงของค่ามาตรวัดสามารถระบุสิ่งผิดปกติในตัวอย่างซอฟต์แวร์ได้

คำสำคัญ: ช่วงของค่ามาตรวัด, มาตรวัดซอฟต์แวร์, คุณภาพซอฟต์แวร์, การโปรแกรมเชิงแง่มุม

1. Introduction

An appealing operational approach for quality management using OO software metrics is to develop thresholds. Thresholds are defined as [1] "heuristic values used to set ranges of desirable and undesirable metric values for measured software. These thresholds are used to identify anomalies, which may or may not be an actual problem." For example, we can say that a certain coupling metric has a threshold of seven. If the measured value for a particular class is larger than seven, then we could flag that class as high risk [2].

Thresholds have a practical, theoretical, and methodological significance. It is much easier for quality assurance personnel to use thresholds for identifying potentially high risk classes; they are more actionable than statistical models and equations that commonly resulted from validation studies [2].

AOP [3] is a new paradigm that addresses crosscutting concerns: behavior of a software system which is hard to decompose and isolate in existing paradigm specifically in object orientation. Such crosscutting concern requires its implementation to be spread across many different modules. AOP aims to improve evolvability and reusability of the

software system by capturing such crosscutting behavior in a new modularization unit calls "aspect".

However, since the AO paradigm is still in its infancy, it is important to determine the threshold metrics to identify anomalies in AO software.

The GoF design patterns [4] offer flexible solutions to common software development problems. Each pattern is comprised of a number of parts, including purpose/intent, applicability, solution structure, and sample implementations. It is accepted as good design. The extracted metrics from GoF patterns should be relevant the information for preliminary obtaining the threshold. As our research, we establish AO software metric thresholds with the 23 GoF patterns and validate it with an AO software example. The results show that it can indicates anomalies in software example.

The rest of this paper is structured as follows. In section 2, we review some existing AO software metrics which is used to establish their thresholds, design patterns and related research on software metric thresholds. Section 3 presents the threshold values of each metric as our main outcome. Section 4 dedicates to presenting the validation of such thresholds through 2 AO software examples. Finally, we conclude in section 5 along with the future works.

2. Literature Research

2.1 AO Software Metrics

In this paper, we focus on metrics suite proposed by Ceccato and Tonella [5], which revised the well known Chidamber and Kemerer's metrics suite. Some of the metrics are adapted or extended, in order to make them applicable to the AOP software. In this suite, *module* will be used as a common term for classes and aspects. Similarly, methods, advices and introductions will be indicated by the *operation* term. There are 10 metrics as following:

Weighted Operations in Module (WOM)

WOM counts number of operations in a given module [5].

Depth of Inheritance Tree (DIT)

DIT is a length of the longest path from a given module to the class/aspect hierarchy root [5]. Since aspects can alter the inheritance relationship by means of static crosscutting, such effects of aspectization must be taken into account when computing this metric [5].

Number Of Children (NOC)

NOC is a number of immediate subclasses or sub-aspects of a given module [5].

Crosscutting Degree of an Aspect (CDA)

CDA is a number of modules affected by the pointcuts and by the introductions in a given aspect [5]. This is a brand new metric, specific to AOP software. CDA measures all modules possibly affected by an aspect. This gives an idea of the overall impact an aspect has on the other modules.

Coupling on Advice Execution (CAE)

CAE is a number of aspects containing advices possibly triggered by the execution of operations in a given module [5]. If the behavior of an operation can be altered by an aspect advice, due to a pointcut intercepting it, there is an (implicit) dependence of the operation from the advice. Thus, the given module is coupled with the aspect containing the advice and a change of the latter might impact the former. Such kind of coupling is absent in OO systems [5].

Coupling on Method Call (CMC)

CMC is a number of modules or interfaces declaring methods that are possibly called by a given module [5]. Aspect introductions must be taken into account when the possibly invoked methods are determined.

Coupling on Field Access (CFA)

CFA is a number of modules or interfaces declaring fields that are accessed by a given module [5]. In OO systems this metric is usually close to zero, but in AOP, aspects might access class fields to perform their function, so observing the new value in aspectized software may be important to assess the coupling of an aspect with other classes/aspects [5].

Coupling between Modules (CBM)

CBM is a number of modules or interfaces declaring methods or fields that are possibly called or accessed by a given module [5].

Response For a Module (RFM)

RFM is number of methods and advices potentially executed in response to a message received by a given module [5]. The main adaptation necessary to apply it to AOP software is associated with the implicit responses that are triggered whenever a pointcut intercepts an operation of the given module [5].

Lack of Cohesion in Operations (LCO)

LCO is number of pairs of operations working on different class fields minus pairs of operations working on common fields (zero if negative) [5]. In [5], they also proposed another AO software metric on which we do not focus. Such metric is CIM (Coupling on Intercepted Modules).

We think that it is enough for using CDA metric because CIM considers only explicit named modules, while CDA measures all modules possibly affected by an aspect. Thus, CDA is covering overall CIM of each aspect.

The AO software metrics described above were collected using AOPMetrics tool. AOPMetrics [6] was a common metrics tool for the OO and AOP. It

was developed by Stochmialek as a master's thesis on Wroclaw University of Technology in Poland.

2.2 Design Patterns

The 23 GoF patterns illustrate a variety of design and structural issues that would be hard to find in a single code base (except in very large and complex systems). The GoF patterns effectively comprise a microcosm of many possible systems. They provided us with a rich source of insights, without the need to analyze large code based or learn domain-specific concepts [7].

Design pattern examples are presented by Hannemann and Kiczales [8]. For each of the 23 GoF patterns they developed a representative example that makes use of the pattern, and implemented the example in both Java and AspectJ. AspectJ [9], which is an extension of the programming language Java, is the most popular one and already has a large community.

Garcia et al. [10] complemented Hannemann and Kiczales' work [8] by performing quantitative assessments of Java and AspectJ implementations for the 23 GoF patterns. They have found that most aspect-oriented solutions improved the separation of pattern-related concerns. Monteiro and Fernandes [7] emphasized that "The implementations presented by Hannemann and Kiczales [8] are currently one of the nearest things to examples of good AOP style and design."

Observer pattern, known as Model-View is intented to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [11]. Object-oriented implementations of the Observer pattern, usually add a field to all potential Subjects that stores a list of Observers interested in that particular Subject. When a Subject wants to report a state change to its Observers, it calls its own *notify* method, which in turn calls an *update* method on all Observers in the list [8].

Figure 1 shows a concrete example of the Observer pattern [8] in the context of a simple figure package. In such a system the Observer pattern is used to cause mutating operations to figure elements to update the screen. As shown in the figure, code for implementing this pattern is spread across the classes. The underlined methods contain code necessary to implement this instance of such pattern.

All participants (i.e. *Point* and *Line*) have to know about their role in the pattern and consequently have pattern code in them. Adding or removing a role from a class requires changes in that class. Changing the notification mechanism (such as switching between push and pull models [4]) requires changes in all paticipanting classes [8].

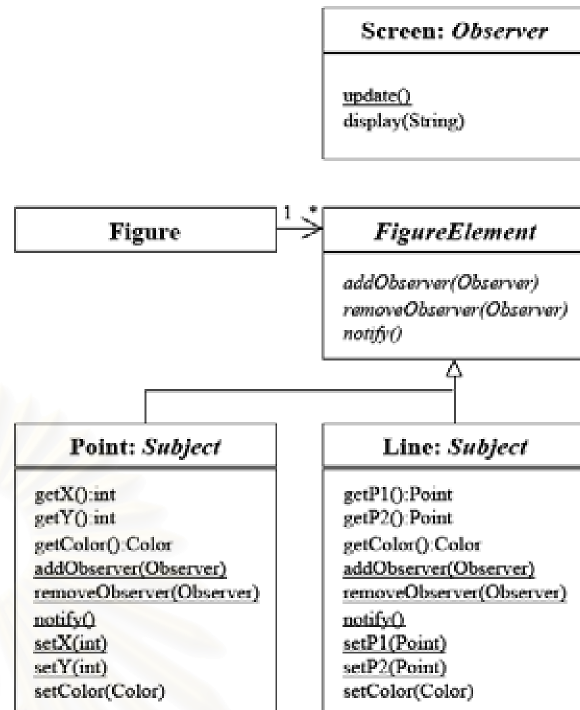


Figure 1. A simple Graphical Figure Element System that uses the Observer pattern in Java [8]

In the AspectJ version [8] all code pertaining to the relationship between Observers and Subjects is moved into an aspect, which changes the dependencies between the modules, as shown in Figure 2. Subject and Observer roles crosscut classes, and the changes of interest (the *subjectChange* pointcut) crosscuts methods in various classes.

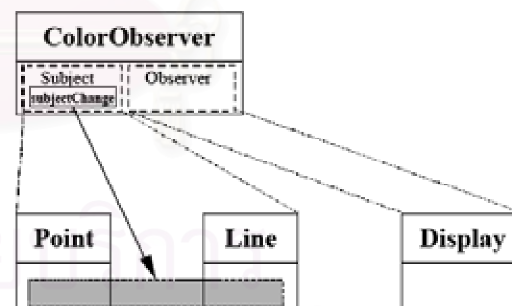


Figure 2. The structure of an instance of the Observer pattern in AspectJ [8]

2.3 Software Metric Thresholds

Henderson-Sellers [12] emphasized the practical utility of thresholds by stating that "An alarm would occur whenever the value of a specific internal metric exceeded some predetermined threshold."

Lorenz and Kidd [1] presented a number of thresholds for object-oriented metrics based on their experiences with Smalltalk and C++ projects.

Similarly, Rosenberg et al. [13] have developed thresholds for a number of popular object-oriented metrics that are used for quality management at NASA GSFC.

French [14] described a technique for deriving thresholds, and applied it to metrics collected from Ada95 and C++ programs.

Benlarbi et al. [2] tested for threshold effects in subset of the CK's metric suite. Their results indicated that there were no threshold effects for any of the metrics studied.

However, none for the above research established the software metric thresholds from Java programs. Meananet [15] presented a number of thresholds of object-oriented software metrics for detecting bad-smells in Java codes.

3. Threshold of Metrics

The approach for determining threshold of AO software metrics is shown in the activity diagram in Figure 3.

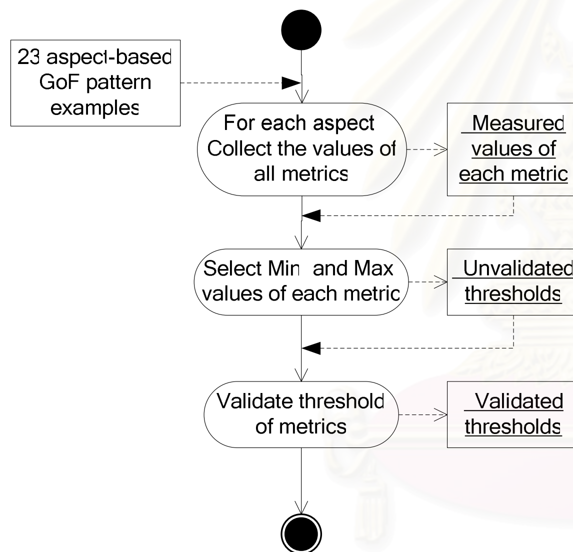


Figure 3. The approach

Assuming GoF patterns are good design. The values of all metrics are firstly collected from the 23 GoF pattern examples, implementing with AspectJ [8]. In order to collect the values of all metrics, we illustrate with an example of aspect belonged to the Observer pattern example [8]. Figure 4 shows aspect *ObserverProtocol* which defines the general behavior of the Observer design pattern. Aspect *ObserverProtocol* consists of 4 methods namely *getObservers*, *addObserver*, *removeObserver* and *updateObserver* and 1 after-advice. Thus, WOM of this aspect is equal to 5. There are 3 subaspects, which inherit from *ObserverProtocol*, namely *CoordinateObserver*, *ColorObserver* and *screenObserver*. As a result NOC is equal to 3 and DIT is equal to 0. CMC, CFA, and CBM are equal to 0 because *ObserverProtocol* is abstract and does not call to the other methods and attributes. RFM is equal to 4 since there are 4 methods in such aspect. The measured values of each AO software metric for aspect *ObserverProtocol* are summarized in Table 1.

Table 1. The measured values from aspect *ObserverProtocol*

AO software metric	Measured value
WOM	5
DIT	0
NOC	3
CDA	4
CAE	1
CMC	0
CFA	0
CBM	0
RFM	4
LCO	10

```
public abstract aspect ObserverProtocol {
  protected interface Subject  { }
  protected interface Observer { }
  private WeakHashMap perSubjectObservers;
  protected List getObservers(Subject subject) {
    if (perSubjectObservers == null) {
      perSubjectObservers = new WeakHashMap();
    }
    List observers =
      (List)perSubjectObservers.get(subject);
    if ( observers == null ) {
      observers = new LinkedList();
      perSubjectObservers.put(subject, observers);
    }
    return observers;
  }
  public void   addObserver(Subject subject,
    Observer observer) {
    getObservers(subject).add(observer);
  }
  public void removeObserver(Subject subject,
    Observer observer) {
    getObservers(subject).remove(observer);
  }
  protected abstract pointcut subjectChange(
    Subject s);
  after(Subject subject): subjectChange(subject) {
    Iterator iter = getObservers(subject).iterator();
    while ( iter.hasNext() ) {
      updateObserver(subject,
        ((Observer)iter.next()));
    }
  }
  protected abstract void updateObserver(
    Subject subject, Observer observer);
}
```

Figure 4. Aspect *ObserverProtocol*

Since AOP is a new paradigm that mainly affects to the implementation of software and may affect to the range of existing OO software metric thresholds. Therefore, we separately consider such thresholds both in class type and aspect type.

For each AO software metric, we collect the metric values through all 23 GoF pattern examples. The sample results are illustrated with WOM on a frequency distribution table based on class and aspect type in Table 2.

Table 2. WOM's results based on class and aspect type

Metric value	Number of classes	Number of aspects
0	3	3
1	70	15
2	54	8
3	45	4
4	26	2
5	7	1
6	2	4
7	3	-
8	-	1
9	-	2
10	1	1
Totals	211	41

After that, we select the minimum (Min.) and maximum (Max.) values of each AO software metric to be the range of thresholds. In Table 2, Min. and Max. of WOM for both class and aspect type are 0 and 10 respectively. The summary of AO software metric thresholds are shown in Table 3 based on class and aspect type.

Table 3. AO software metric thresholds based on class type

Software metric	Class		Aspect	
	Min.	Max.	Min.	Max.
WOM(WMC)	0	10	0	10
DIT	0	5	0	1
NOC	0	2	0	3
CDA	-	-	0	52
CAE	-	-	0	2
CMC	-	-	0	6
CFA	-	-	0	2
CBM(CBO)	0	7	0	6
RFM(RFC)	0	10	0	14
LCO(LCOM)	0	24	0	32

Note: The abbreviation of each metric in parentheses is the original abbreviation of CK's metrics suite.

In Table 3, any aspect may be an anomaly if the metric value is out of range. For example, if the value of RFM is less than 0 or greater than 14, that aspect is suspected to be an anomaly.

4. Validation of Metric Thresholds

We validate such metric thresholds with 2 AO software examples name AJHotDraw [16] and AspectTetris [17]. AJHotDraw is an AO refactoring of JHotDraw, a relatively large and well-designed open source Java framework for technical and structured 2D graphics. There are 350 classes, 50 interfaces and 10 aspects in AJHotDraw.

AspectTetris is the game Tetris made in AspectJ. It was implemented by Evertsson as a part of the course Advanced Software Engineering at Blekinge Institute of Technology. AspectTetris consists of 16 classes, 1 interface and 8 aspects.

After considering the measured value results of these examples, we found 63.14% of classes and 20% of aspects in AJHotDraw are suspected to be anomalies. In AspectTetris, 6.25% of classes and 12.5% of aspects are suspected to be anomalies. The limited space at our disposal in this paper dose not allow us to rigorously discuss all anomalies found in both examples. Therefore, we discuss all anomalies found only in aspect type. The metric results of all aspects in AJHotDraw and AspectTetris are shown in Table 4 and Table 5 respectively.

In Table 4, RFM metric values of aspects names *PersistentTextFigure* and *PersistentCompositeFigure* are greater than the range of its threshold. After consider both of them at code level, methods in these aspects invoked a large number of methods of other aspects. Such methods in these aspects are a kind of method introductions, which introduce to class *TextFigure* and class *CompositeFigure* that they cut across as a persistence concern.

In Table 5, CFA of aspect *NextBlock* is greater than the range of its threshold. After considering, at code level, *NextBlock* accesses to some fields of 3 classes to perform its function.

According to the validation of 2 software examples above, the metric thresholds can preliminary indicate the anomalies in them.

5. Conclusions and Future works

Thresholds are used to identify anomalies in software, which may or may not be an actual problem. Aspect-orientation is a newly programming paradigm that solves the crosscutting problem, which traditional object-orientation cannot solve. Therefore, to identify such anomalies in AO software, it is important to determine threshold for AO software metrics. There are 10 AO software metrics proposed by Ceccato and Tonella [5], which revise the well known CK's metrics suite. As it is widely accepted that the GoF patterns are good design, so in this research we establish such threshold for each AO software metric from 23 aspect-based GoF pattern examples. After that, we validate them through AJHotDraw and AspectTetris software examples. The results show that the established thresholds can be used to preliminarily indicate the anomalies in

both examples. We plan to improve the range with more design examples and with other techniques.

Acknowledgement

We would like to thanks Michal Stochmialek for supporting and helping us with a great tool.

6. References

[1] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*: Prentice Hall, Inc., 1994.
[2] S. Benlarbi, K. E. Emam, N. Goel, and S. N. Rai, "Thresholds for Object-Oriented Measures," presented at 11th International Symposium on Software Reliability Engineering (ISSRE'00), 2000.
[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," presented at Eiropean Conference on Object-Oriented Programming (ECOOP), Finland, 1997.
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1994.
[5] M. Ceccato and P. Tonella, "Measuring the Effects of Software Aspectization," presented at 1st Workshop on Aspect Reverse Engineering (WARE) at Working Conference on Reverse Engineering (WCRE), Delft, The Netherlands, 2004.
[6] http://aopmetrics.tigris.org/.
[7] M. P. Monteiro and J. M. Fernandes, "Towards a Catolog of Aspect-Oriented Refactorings," presented at AOSD 05, Chicago, Illinois, USA, 2005.
[8] J. Hannemann and G. Kiczales, "Design Pattern Implementation in Java and AspectJ," presented at OOPSLA'02, Seattle, Washington, USA, 2002.
[9] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*: Manning Publications Co., 2003.
[10] A. Garcia, C. Sant'Anna, and E. Figueiredo, "Modularizing Design Patterns with Aspects: A Quantitative Study," presented at AOSD'05, 2005.
[11] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," presented at IEEE Transactions on Software Engineering, 1994.
[12] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*: Prentice-Hall, 1996.
[13] L. Rosenberg, R. Stapko, and A. Gallo, "Object-Oriented Metrics for Reliability," presented at IEEE International Symposium on Software Metrics, 1999.
[14] V. French, "Estiblishing Software Metrics Thresholds," presented at the 9th International Workshop on Software Measurement, 1999.
[15] P. Meananet, "Threshold of Object-Oriented Software Metrics for Detecting Bad-Smells Code," in *Computer Engineering*. Bangkok: Chulalongkorn University, 2004.
[16] A. v. Deursen, M. Marin, and L. Moonen, "AJHotDraw: A showcase for refactoring to aspects," presented at Linking Aspect Technology and Evolution Workshop (LATE) AOSD 2005, Chicago, USA, 2005.
[17] http://www.guzzzt.com/coding/aspecttetris.shtml.

Table 4. The metric value results of all aspect types in AJHotDraw

Type name	Metric									
	WOM	DIT	NOC	CDA	CAE	CMC	CFA	CBM	RFM	LCO
PersistentImageFigure	2	0	0	1	0	4	1	5	10	0
PersistentFigure	2	0	0	2	0	0	0	0	2	0
PersistentDrawing	0	0	0	1	0	0	0	0	0	0
PersistentAttributeFigure	2	0	0	1	0	4	0	4	8	0
PersistentTextFigure	2	0	0	1	0	5	1	5	22	0
PersistentCompositeFigure	2	0	0	1	0	6	0	6	17	0
SelectionChangedNotification	2	0	0	1	0	1	0	1	0	0
FigureSelectionObserverRole	5	0	0	7	0	2	0	2	7	0
FigureSelectionSubjectRole	9	0	0	3	0	3	0	3	8	30
CmdCheckViewRef	1	0	0	18	0	2	0	2	0	0

Table 5. The metric value results of all aspect types in AspectTetris

Type name	Metric									
	WOM	DIT	NOC	CDA	CAE	CMC	CFA	CBM	RFM	LCO
GameInfo	1	0	0	1	0	2	0	2	0	0
Menu	3	0	0	2	0	3	1	3	2	0
NextBlock	2	0	0	2	1	2	3	4	0	1
NewBlocks	5	0	0	3	0	1	1	2	0	0
DesignCheck	0	0	0	0	0	0	0	0	0	0
TestAspect	1	0	0	1	0	1	0	1	0	0
Counter	6	0	0	2	1	0	1	1	0	0
Levels	4	0	0	4	0	1	1	2	0	0

APPENDIX B

ZHAO'S METRICS

Zhao [20] proposed a measure suite for assessing the coupling in AO system. The coupling in AO system is mainly about the degree of interdependence among aspects and/or classes.

In order to formally define his coupling measures in AO systems, he defined a terminology for an AO system, which is based on a similar terminology used in [27] for OO systems.

1. **System**

Definition 1 (AO system) *An AO system* S *consists of a set of aspects,* $A(S)$ *and a set of classes,* $C(S)$*.*

2. **Modules**

Definition 3 (Mudules of an Aspect or a Class) *Let* S *be an AO system. For each* $a \in A(S)$*, let* $\mathcal{A}(a)$ *be the set of advices of* a*,* $\mathcal{I}(a)$ *be the set of intertype declarations of* a*,* $\mathcal{P}(a)$ *be the set of poincuts of* a*, and* $\mathcal{M}(a)$ *be the set of methods of* a*, and* $\mathcal{M}_{all}(a)$ *be the set of all modules of* a*. For each* $c \in C(S)$*, let* $\mathcal{M}(c)$ *be the set of methods of* c*.*

In an AO system, advice, intertype declaration, pointcut, or method may have a set of parameters that may also influence coupling measurement. So he defines this issue as follows.

Definition 4 (Parameters) *Let* S *be an AO system. For each* $\alpha \in \mathcal{A}(S)$*,* $i \in \mathcal{I}(S)$*,* $p \in \mathcal{P}(S)$*, or* $m \in \mathcal{M}(S)$*, let* $Par(\alpha)$ *be the parameters of advice* α*,* $Par(i)$ *be the parameters of intertype declaration* i*,* $Par(p)$ *be the parameters of pointcut* p*, and* $Par(m)$ *be the parameters of method* m*.*

3. **Module Invocations**

In AO systems modules such as advices, intertype declarations, and methods in an aspect may invoke other modules of some classes. To measure coupling of an aspect, it is necessary to define the set of modules that a piece of

advice, an intertype declaration, or a method of the aspect invokes. Also, the frequency of these invocations should be defined.

Definition 5 (The Set of Invoked Methods) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S.*

- *For each piece of advice $\alpha \in \mathcal{A}(a)$, the set of invoked methods of α is denoted as $SIM(\alpha)$ such that if $\exists m \in \mathcal{M}(c)$ and the body of α has a method invocation where m is invoked for an object of c, then $m \in SIM(\alpha)$.*
- *For each intertype declaration $i \in I(a)$, the set of invoked methods of i is denoted as $SIM(i)$ such that if $\exists m \in \mathcal{M}(c)$ and the body of i has a method invocation where m is invoked for an object of c, then $m \in SIM(i)$.*
- *For each pointcut $p \in \mathcal{P}(a)$, the set of invoked methods of p is denoted as $SIM(p)$ such that if $\exists m \in \mathcal{M}(c)$ and the body of p has a method invocation where m is invoked for an object of c, then $m \in SIM(p)$.*
- *For each method $m \in \mathcal{M}(a)$, the set of invoked methods of m is denoted as $SIM(m)$ such that if $\exists m' \in \mathcal{M}(c)$ and the body of m has a method invocation where m' is invoked for an object of c, then $m' \in SIM(m)$.*

Definition 6 (The Number of Method Invocations) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S.*

- *For each piece of advice $\alpha \in \mathcal{A}(a)$, $NSI(\alpha, m)$ is the number of method invocations of m by α such that $m \in SIM(\alpha)$ and m is invoked for an object of c.*
- *For each intertype declaration $i \in I(a)$, $NSI(i, m)$ is the number of method invocations of m by i such that $m \in SIM(i)$ and m is invoked for an object of c.*
- *For each pointcut $p \in \mathcal{P}(a)$, $NSI(p, m)$ is the number of method invocations of m by p such that $m \in SIM(p)$ and m is invoked for an object of c.*

- *For each method $m' \in \mathcal{M}(a)$, $NSI(m', m)$ is the number of method invocations of m by m' such that $m \in SIM(m')$ and m is invoked for an object of c.*

4. Attributes

Definition 7 (Attributes of Aspects and Classes) *Let S be an AO system. For each $a \in A(S)$, let $\mathcal{A}^a(a)$ be the set of attributes of aspect a. For each $c \in C(S)$, let $\mathcal{A}^a(c)$ be the set of attributes of class c.*

5. Types

Attributes and parameters contain types that all can contribute to coupling measurement of AO systems.

Definition 8 (Available Types) *Let S be an AO system. The set T of available types in S is $T = T_{bi} \cup T_{ud} \cup T_c \cup T_a$ where T_{bi} is the set of build-in types provided by the programming language, T_{ud} is the set of user-defined types, T_c is the set of class types, and T_a is the set of aspect types.*

Definition 9 (Types of Attributes and Parameters) *Let S be an AO system, $x \in \mathcal{A}^a(a)$ be an attribute of aspect a, and $y \in \mathcal{A}^a(c)$ be an attribute of class c. The type of x is denoted by $T(x) \in T$ and the type of y is denoted by $T(y) \in T$.*

The coupling framework for AO systems is next described. The framework focuses on coupling caused by dependencies that occur between aspect and class in an AO system which are called aspect-class dependencies.

Definition 10 (Attribute-class dependence) *There is an attribute-class dependence between aspect a and class c, if c is the type of an attribute of a. The number of attribute-class dependencies from a to c can formally be represented as*

$$AtC(a,c) = \left|\{x \mid x \in \mathcal{A}^a(a) \wedge T(x) = c\}\right|$$

Definition 11 (Module-class Dependence) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S. There are four types of module-class dependencies that can be defined as follows:*

- *Advice-class dependence:*

  There is an advice-class dependence between a and c, if c is the type of a parameter of a piece of advice α of a, or c is the return type of

α. The number of advice-class dependencies from a to c can formally be represented as

$$AC(a,c) = \sum_{\alpha \in \mathcal{A}(a)} \left| \{x | x \in Par(\alpha) \wedge T(x) = c\} \right|$$

- *Intertype-class dependence:*

There is an intertype-class dependence between a and c, if c is the type of a parameter of an intertype declaration i of a, or c is the return type of i. The number of intertype-class dependencies from a to c can formally be represented as

$$IC(a,c) = \sum_{i \in I(a)} \left| \{x | x \in Par(i) \wedge T(x) = c\} \right|$$

- *Method-class dependence:*

There is an method-class dependence between a and c, if c is the type of a parameter of a method m of a, or c is the return type of m. The number of method-class dependencies from a to c can formally be represented as

$$MC(a,c) = \sum_{m \in \mathcal{M}(a)} \left| \{x | x \in Par(m) \wedge T(x) = c\} \right|$$

- *Pointcut-class dependence:*

Let p be a pointcut of aspect a. There is a pointcut-class dependence between a and c, if c is the type of a parameter of a pointcut p of a. The number of pointcut-class dependencies from a to c can formally be represented as

$$PC(a,c) = \sum_{p \in \mathcal{P}(a)} \left| \{x | x \in Par(p) \wedge T(x) = c\} \right|$$

Definition 12 (Module-method Dependence) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S. There are four types of module-method dependencies between a and c that can be defined as follows:*

- *Advice-method dependence:*

There is an advice-method dependence between a and c, if a piece of advice α of a directly invokes a method m of c. The number of advice-method dependencies from a to c can formally be represented as

$$AM(a,c) = \sum_{\alpha \in \mathcal{A}(a)} \sum_{m \in \mathcal{M}(c)} \left| (NSI(\alpha, m)) \right|$$

- *Intertype-method dependence:*

There is an intertype-method dependence between a and c, if an intertype i of a directly invokes a method m of c. The number of intertype-method dependencies from a to c can formally be represented as

$$IM(a,c) = \sum_{i \in I(a)} \sum_{m \in \mathcal{M}(c)} \left|(NSI(i,m))\right|$$

- *Method-method dependence:*

There is an method-method dependence between a and c, if a method m of a directly invokes a method m' of c. The number of method-method dependencies from a to c can formally be represented as

$$MM(a,c) = \sum_{m \in \mathcal{M}(a)} \sum_{m' \in \mathcal{M}(c)} \left|(NSI(m,m'))\right|$$

- *Pointcut-method dependence:*

There is an pointcut-method dependence between a and c, if a pointcut p of a contains at least one join point that is related to a method m of c. The number of pointcut-method dependencies from a to c can formally be represented as

$$PM(a,c) = \sum_{p \in \mathcal{P}(a)} \sum_{m \in \mathcal{M}(c)} \left|(NSI(p,m))\right|$$

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX C

REFACTORING SOFTWARE CODE

The software code before and after refactoring are presented here. Telecom software is selected to depict the restructuring of a software code. The sample software includes ten classes and three aspects before applying refactoring procedures. After applying refactoring procedures, the sample software includes ten classes and four aspects as a result of changing interaction coupling among aspects to be inheritance coupling among aspects. Figure C.1 – Figure C.3 show fractions of code of all aspects in Telecom software i.e. aspect *Timing*, aspect *Billing*, and aspect *TimerLog* before applying refactoring procedures.

```
01  public aspect Timing {
02      public long Customer.totalConnectTime = 0;
03
04      public long getTotalConnectTime(Customer cust) {
05          return cust.totalConnectTime;
06      }
07
08      private Timer Connection.timer = new Timer();
09      public Timer getTimer(Connection conn) { return conn.timer; }
10
11      after (Connection c): target(c) && call(void
12  Connection.complete()) {
13          getTimer(c).start();
14      }
15
16      pointcut endTiming(Connection c): target(c) &&
17          call(void Connection.drop());
18
19      after(Connection c): endTiming(c) {
20          getTimer(c).stop();
21          c.getCaller().totalConnectTime += getTimer(c).getTime();
22          c.getReceiver().totalConnectTime +=
23  getTimer(c).getTime();
24      }
25  }
```

Figure C.1: Aspect *Timing* of Telecom software before applying refactoring procedures.

```
01  public aspect TimerLog {
02
03      after(Timer t): target(t) && call(* Timer.start())  {
04        System.err.println("Timer started: " + t.startTime);
05      }
06
07      after(Timer t): target(t) && call(* Timer.stop()) {
08        System.err.println("Timer stopped: " + t.stopTime);
09      }
10  }
```

Figure C.2: Aspect *TimerLog* of Telecom software before applying refactoring procedures.

```
01  public aspect Billing {
02      declare precedence: Billing, Timing;
03
04      public static final long LOCAL_RATE = 3;
05      public static final long LONG_DISTANCE_RATE = 10;
06
07      public Customer Connection.payer;
08      public Customer getPayer(Connection conn) { return
09  conn.payer; }
10
11      after(Customer cust) returning (Connection conn):
12          args(cust, ..) && call(Connection+.new(..)) {
13          conn.payer = cust;
14      }
15
16      public abstract long Connection.callRate();
17
18      public long LongDistance.callRate() { return
19  LONG_DISTANCE_RATE; }
20      public long Local.callRate() { return LOCAL_RATE; }
21
22      after(Connection conn): Timing.endTiming(conn) {
23          long time = Timing.aspectOf().getTimer(conn).getTime();
24          long rate = conn.callRate();
25          long cost = rate * time;
26          getPayer(conn).addCharge(cost);
27      }
28
29      public long Customer.totalCharge = 0;
30      public long getTotalCharge(Customer cust) { return
31  cust.totalCharge; }
32
33      public void Customer.addCharge(long charge){
34          totalCharge += charge;
35      }
36  }
```

Figure C.3: Aspect *Billing* of Telecom software before applying refactoring procedures.

Figure C.4 – C.7 show fractions of code of all aspects in Telecom software i.e. aspect *Timing*, aspect *Billing*, and aspect *TimerLog* after applying refactoring procedures. Aspect *XPI* is also presented as the new aspect taken place from the eliminating borrowed pointcut solution.

```
01  public aspect Timing {
02      public long Customer.totalConnectTime = 0;
03
04      public long getTotalConnectTime(Customer cust) {
05          return cust.totalConnectTime;
06      }
07
08      private Timer Connection.timer = new Timer();
09      public Timer getTimer(Connection conn) { return conn.timer; }
10
11      pointcut completeMethod(): call(void Connection.complete());
12
13      after (Connection c): target(c) && completeMethod(){
14          getTimer(c).start();
15      }
16
17
18      after(Connection c): XPI.endTiming(c) {
19          getTimer(c).stop();
20          c.getCaller().totalConnectTime += getTimer(c).getTime();
21          c.getReceiver().totalConnectTime +=
22  getTimer(c).getTime();
23      }
24  }
```

Figure C.4: Aspect *Timing* of Telecom software after applying refactoring procedures.

```
01  public aspect TimerLog {
02
03      pointcut startMethod(): call(* Timer.start());
04      after(Timer t): target(t) && startMethod() {
05        System.err.println("Timer started: " + t.startTime);
06      }
07      pointcut stopMethod(): call(* Timer.stop());
08      after(Timer t): target(t) && stopMethod() {
09        System.err.println("Timer stopped: " + t.stopTime);
10      }
11  }
```

Figure C.5: Aspect *TimerLog* of Telecom software after applying refactoring procedures.

```
01  public aspect Billing {
02      declare precedence: Billing, Timing;
03
04      public static final long LOCAL_RATE = 3;
05      public static final long LONG_DISTANCE_RATE = 10;
06
07      public Customer Connection.payer;
08      public Customer getPayer(Connection conn) { return
09  conn.payer; }
10
11      pointcut connectionConstructor(): call(Connection+.new(..));
12
13      after(Customer cust) returning (Connection conn):
14          args(cust, ..) && connectionConstructor() {
15          conn.payer = cust;
16      }
17
18      public abstract long Connection.callRate();
19
20      public long LongDistance.callRate() { return
21  LONG_DISTANCE_RATE; }
22      public long Local.callRate() { return LOCAL_RATE; }
23
24      after(Connection conn): XPI.endTiming(conn) {
25          long time = Timing.aspectOf().getTimer(conn).getTime();
26          long rate = conn.callRate();
27          long cost = rate * time;
28          getPayer(conn).addCharge(cost);
29      }
30
31      public long Customer.totalCharge = 0;
32      public long getTotalCharge(Customer cust) { return
33  cust.totalCharge; }
34
35      public void Customer.addCharge(long charge){
36          totalCharge += charge;
37      }
38  }
```

Figure C.6: Aspect *Billing* of Telecom software after applying refactoring procedures.

```
01  public aspect XPI {
02
03      pointcut endTiming(Connection c): target(c) &&
04          call(void Connection.drop());
05
06  }
```

Figure C.7: Aspect *XPI* of Telecom software taken place from the eliminating borrowed pointcut solution.

APPENDIX D

VALIDATION RESULTS

The measured values of all bad-smell metrics, which are obtained from four sample software, are summarized here. The bad-smell metrics are separated into two groups: pointcut metrics and aspect metrics. Table D.1 - Table D.20 summarize the measured values of pointcut metrics and Table D.21 – Table D.32 summarize the measured values of aspect metrics.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.1: Measured values of pointcut metrics in Telecom before refactoring.

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
Billing	advice@line7	0	{Connection.new(), LongDistance.new(), Local.new()}	after	1	0	1
TimerLog	advice@line2	0	{Timer.start()}	after	1	0	1
	advice@line5	0	{Timer.stop()}	after	1	0	1
Timing	advice@line7	0	{Connection.complete()}	after	1	0	1
	endTiming	1	{Connection.drop()}	after	1	1	2

Table D.2: Measured values of pointcut metrics in Telecom after refactoring.

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
Billing	connection Constructor	0	{Connection.new(), LongDistance.new(), Local.new()}	after	1	0	1
TimerLog	startMethod	0	{Timer.start()}	after	1	0	1
	stopMethod	0	{Timer.stop()}	after	1	0	1
Timing	completeMethod	0	{Connection.complete()}	after	1	0	1
XPI	endTiming	0	{Connection.drop()}	0		2	2

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.3: Measured values of pointcut metrics in Spacewar before refactoring.

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
Coordinator	synchronizationPoint	0	∅	before	1	0	2
				after	1		
Debug	advice@line16	0	{SWFrame.new()}	after	1	0	1
	allConstructorsCut	0	{Bullet.new(), Display1.new(), Display2.new(), Display.new(), EnergyPacket.new(), EnergyPacketProducer.new(), Game.new(), GameSynchronization.new(), Pilot.new(), Palyer.new(), Registry.new(), RegistrySynchronization.new(), Robot.new(), Ship.new(), SpaceObject.new(), SWFrame.new(), Timer.new()}	before	1	0	2
				after	1		
	allInitializationsCut	0	{Bullet.new(), Display1.new(), Display2.new(), Display.new(), EnergyPacket.new(), EnergyPacketProducer.new(), Game.new(),	before	1	0	2

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.4: Measured values of pointcut metrics in Spacewar before refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
			GameSynchronization.new(), Pilot.new(), Palyer.new(), Registry.new(), RegistrySynchronization.new(), Robot.new(), Ship.new(), SpaceObject.new(), SWFrame.new(), Timer.new()}	after	1		
	allMethodsCut	0	{Bullet.new(), Display1.new(), Display2.new(), Display.new(), EnergyPacket.new(), EnergyPacketProducer.new(), Game.new(), GameSynchronization.new(), Pilot.new(), Palyer.new(), Registry.new(), RegistrySynchronization.new(), Robot.new(), Ship.new(), SpaceObject.new(), SWFrame.new(), Timer.new()}	before	1	0	2
				after	1		

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.5: Measured values of pointcut metrics in Spacewar before refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
	advice@line61	0	{clockTick()}	after	1	0	1
	advice@line69	0	{register(), unregister()}	after	1	0	1
	advice@line76	0	{Ship.fire()}	after	1	0	1
	advice@line80	0	{Ship.handleCollision()}	after	1	0	1
	advice@line85	0	{Ship.bounce()}	after	1	0	1
	advice@line90	0	{Ship.inflictDemage()}	before	1	0	1
EnsureShipIsAlive	-	-	-	-		-	-
GameSynchronization	synchronizationPoint	0	{Game.handleCollisions(), Game.newShip()}	0		0	0
RegistrySynchronizatio n	synchronizationPoint	0	{Registry.register(), Registry.unregister(), Registry.getObjects(), Registry.getShips()}	0		0	0

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.6: Measured values of pointcut metrics in Spacewar after refactoring.

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
Coordinator	synchronizationPoint	0	∅	before	1	0	2
				after	1		
Debug	SWFrameConstructor	0	{SWFrame.new()}	after	1	0	1
	allConstructorsCut	0	{Bullet.new(), Display1.new(), Display2.new(), Display.new(), EnergyPacket.new(), EnergyPacketProducer.new(), Game.new(), GameSynchronization.new(), Pilot.new(), Palyer.new(), Registry.new(), RegistrySynchronization.new(), Robot.new(), Ship.new(), SpaceObject.new(), SWFrame.new(), Timer.new()}	before	1	0	2
				after	1		
	allInitializationsCut	0	{Bullet.new(), Display1.new(), Display2.new(), Display.new(), EnergyPacket.new(), EnergyPacketProducer.new(), Game.new(),	before	1	0	2

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.7: Measured values of pointcut metrics in Spacewar after refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
			GameSynchronization.new(), Pilot.new(), Palyer.new(), Registry.new(), RegistrySynchronization.new(), Robot.new(), Ship.new(), SpaceObject.new(), SWFrame.new(), Timer.new()}	after	1		
	allMethodsCut	0	{Bullet.new(), Display1.new(), Display2.new(), Display.new(), EnergyPacket.new(), EnergyPacketProducer.new(), Game.new(), GameSynchronization.new(), Pilot.new(), Palyer.new(), Registry.new(), RegistrySynchronization.new(), Robot.new(), Ship.new(), SpaceObject.new(), SWFrame.new(), Timer.new()}	before	1	0	2
				after	1		

Table D.8: Measured values of pointcut metrics in Spacewar after refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
	clockTickMethod	0	{clockTick()}	after	1	0	1
	regisunregisMethod	0	{register(), unregister()}	after	1	0	1
	fireMethod	0	{Ship.fire()}	after	1	0	1
	handleCollision Method	0	{Ship.handleCollision()}	after	1	0	1
	bounceMethod	0	{Ship.bounce()}	after	1	0	1
	inflictDemageMethod	0	{Ship.inflictDemage()}	before	1	0	1
EnsureShipIsAlive	helmCommandsCut	0	{rotate(), thrust(), fire()}	around	1	0	1
GameSynchronization	synchronizationPoint	0	{Game.handleCollisions(), Game.newShip()}	0		0	0
RegistrySynchronizatio n	synchronizationPoint	0	{Registry.register(), Registry.unregister(), Registry.getObjects(), Registry.getShips()}	0		0	0

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.9: Measured values of pointcut metrics in AspectTetris before refactoring.

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
Counter	guiInit	0	{TetrisGUI.new()}	after	1	0	1
	deleteLines	0	{AspectTetris.deleteLines()}	before	1	0	2
				after	1		
	deleteLine	0	{Blocks.deleteLine()}	before	1	0	1
	Newgame	0	{AspectTetris.restartGame()}	before	1	0	1
	gameOver	0	{AspectTetris.gameOver()}	after	1	0	1
DesignCheck	declarewarning@line2	0	{BlockPanel.new(), IEventListener.incomingEvent(), Blocks.new(), Blocks.combineBlocks(), Blocks.deleteBlocks(), Blocks.checkCombineBlocks(), Blocks.turnBlock(), Blocks.deleteLine(), Blocks.getBlocks(), Blocks.typeToString(), Blocks.typeToColor(), Blocks.typeToImage(), TetrisImages.preLoad(), TetrisImages.new(), TetrisImages.setInstance(), TetrisImages.getImage(), TetrisImages.loadImage(), Timer.new(), Timer.setSleepTime(), Timer.start(), Timer.stop(),	-		0	1

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.10: Measured values of pointcut metrics in AspectTetris before refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*	*NOAsP*	*NAdP*
			Timer.run(), AspectTetris.startTetris(), AspectTetris.incomingEvent(), AspectTetris.newBlock(), AspectTetris.deleteLines(), AspectTetris.gameOver(), AspectTetris.restartGame(), AspectTetris.pauseGame, AspectTetris.getRandomBlock()}			
	declarewarning@line4	0	{ AspectTetris.startTetris(), AspectTetris.incomingEvent(), AspectTetris.newBlock(), AspectTetris.deleteLines(), AspectTetris.gameOver(), AspectTetris.restartGame(), AspectTetris.pauseGame, AspectTetris.getRandomBlock(), IEventListener.incomingEvent(), Blocks.new(), Blocks.combineBlocks(), Blocks.deleteBlocks(), Blocks.checkCombineBlocks(), Blocks.turnBlock(), Blocks.deleteLine(), Blocks.getBlocks(), Blocks.typeToString(), Blocks.typeToColor(), Blocks.typeToImage(), TetrisImages.preLoad(),	-	0	1

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.11: Measured values of pointcut metrics in AspectTetris before refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
			TetrisImages.new(), TetrisImages.setInstance(), TetrisImages.getImage(), TetrisImages.loadImage(), Timer.new(), Timer.setSleepTime(), Timer.start(), Timer.stop(), Timer.run(), BlockPanel.new(), BlockPanel.paintComponent(), BlockPanel.setBlocks(), BlockPanel.setBlock(), BlockPanel.getMiniminSize(), BlockPanel.getMaximunSize(), BlockPanel.getPreferredSize(), Driver.new(), Driver.setup(), Driver.run(), TetrisGUI.new()}				
GameInfo	guiInit	0	{TetrisGUI.new()}	before	1	0	1
Levels	guiInit	0	{TetrisGUI.new()}	after	1	0	1
	timerInit	0	{ Timer.new()}	before	1	0	1
	deleteLines	0	{Counter.totalLines}	after	1	0	1
	newGame	0	{AspectTetris.restartGame()}	before	1	0	1
Menu	guiInit	0	{TetrisGUI.new()}	after	1	0	1
	tetrisInit	0	{ AspectTetris.new()}	before	1	0	1

Table D.12: Measured values of pointcut metrics in AspectTetris before refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
NewBlocks	getBlock	0	{ Blocks.getBlocks()}	around	1	0	1
	typeToString	0	{ Blocks.typeToString()}	around	1	0	1
	typeToColor	0	{Blocks.typeToColor()}	around	1	0	1
	typeToImage	0	{Blocks.typeToImage()}	around	1	0	1
	numberOfTypes	0	{Blocks.NUMBEROFTYPES}	around	1	0	1
NextBlock	guiInit	0	{TetrisGUI.new()}	after	1	0	1
	getNextBlock	0	{AspectTetris.getRandomBlock()}	around	1	0	1
TestAspect	logPoint	0	{TetrisImages.loadImage()}	before	1	0	1

Table D.13: Measured values of pointcut metrics in AspectTetris after refactoring.

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
Counter	deleteLine	0	{Blocks.deleteLine()}	before	1	0	1
	gameOver	0	{AspectTetris.gameOver()}	after	1	0	1
DesignCheck	outsideGUIPackage	0	{BlockPanel.new(), IEventListener.incomingEvent(), Blocks.new(), Blocks.combineBlocks(), Blocks.deleteBlocks(), Blocks.checkCombineBlocks(), Blocks.turnBlock(), Blocks.deleteLine(), Blocks.getBlocks(), Blocks.typeToString(), Blocks.typeToColor(), Blocks.typeToImage(), TetrisImages.preLoad(), TetrisImages.new(), TetrisImages.setInstance(), TetrisImages.getImage(), TetrisImages.loadImage(), Timer.new(), Timer.setSleepTime(), Timer.start(), Timer.stop(),	-		0	1

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.14: Measured values of pointcut metrics in AspectTetris after refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*	*NOAsP*	*NAdP*
			Timer.run(), AspectTetris.startTetris(), AspectTetris.incomingEvent(), AspectTetris.newBlock(), AspectTetris.deleteLines(), AspectTetris.gameOver(), AspectTetris.restartGame(), AspectTetris.pauseGame, AspectTetris.getRandomBlock()}			
	dontCallAspectTetris outside	0	{ AspectTetris.startTetris(), AspectTetris.incomingEvent(), AspectTetris.newBlock(), AspectTetris.deleteLines(), AspectTetris.gameOver(), AspectTetris.restartGame(), AspectTetris.pauseGame, AspectTetris.getRandomBlock(), IEventListener.incomingEvent(), Blocks.new(), Blocks.combineBlocks(), Blocks.deleteBlocks(), Blocks.checkCombineBlocks(), Blocks.turnBlock(), Blocks.deleteLine(), Blocks.getBlocks(), Blocks.typeToString(), Blocks.typeToColor(), Blocks.typeToImage(), TetrisImages.preLoad(),	-	0	1

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.15: Measured values of pointcut metrics in AspectTetris after refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
			TetrisImages.new(), TetrisImages.setInstance(), TetrisImages.getImage(), TetrisImages.loadImage(), Timer.new(), Timer.setSleepTime(), Timer.start(), Timer.stop(), Timer.run(), BlockPanel.new(), BlockPanel.paintComponent(), BlockPanel.setBlocks(), BlockPanel.setBlock(), BlockPanel.getMiniminSize(), BlockPanel.getMaximunSize(), BlockPanel.getPreferredSize(), Driver.new(), Driver.setup(), Driver.run(), TetrisGUI.new()}				
GameInfo	-	-	-	-		-	-
Levels	timerInit	0	{ Timer.new()}	before	1	0	1
Menu	tetrisInit	0	{ AspectTetris.new()}	before	1	0	1

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.16: Measured values of pointcut metrics in AspectTetris after refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
NewBlocks	getBlock	0	{ Blocks.getBlocks()}	around	1	0	1
	typeToString	0	{ Blocks.typeToString()}	around	1	0	1
	typeToColor	0	{Blocks.typeToColor()}	around	1	0	1
	typeToImage	0	{Blocks.typeToImage()}	around	1	0	1
	numberOfTypes	0	{Blocks.NUMBEROFTYPES}	around	1	0	1
	getNextBlock	0	{AspectTetris.getRandomBlock()}	around	1	0	1
TestAspect	logPoint	0	{TetrisImages.loadImage()}	before	1	0	1
XPI	guiInit	0	{TetrisGUI.new()}	0		5	5
	deleteLines	0	{Counter.totalLines}	0		2	3
	newGame	0	{AspectTetris.restartGame()}	0		2	2

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.17: Measured values of pointcut metrics in AJHotDraw before refactoring.

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
CmdCheckViewRef	commandExecute	0	{AbstractCommand.execute(), UndoCommand.execute(), RedoCommand.execute(), ToggleGridCommand.execute(), SendToBackCommand.execute(), DelectAllCommand.execute(), FigureTransferCommand.execute(), ChangeAttributeCommand.execute(), BringToFrontCommand.execute(), AlignCommand.execute()}	before	1	0	1
FigureSelection ObserverRole	-	-	-	-		-	-
FigureSelectionSubject Role	advice@line3	0	{StandardDrawingView.new()}	after	1	0	1
	advice@line9	0	{StandardDrawingView.readObject()}	after	1	0	1
PersistentAttribute Figure	-	-	-	-		-	-
PersistentComposite Figure	-	-	-	-		-	-

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.18: Measured values of pointcut metrics in AJHotDraw before refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
PersistentDrawing	-	-	-	-		-	-
PersistentFigure	-	-	-	-		-	-
PersistentImageFigure	-	-	-	-		-	-
PersistentTextFigure	-	-	-	-		-	-
SelectionChanged Notification	invalidateSelFigure	0	{ StandardDrawingView.addToSelectionImpl(),Standard DrawingView .removeFromSelection()}	after	1	0	1
	clear_toggleSelection	0	{ StandardDrawingView.clearSelection(),StandardDrawi ngView.toggleSelection()}	after	1	0	1

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.19: Measured values of pointcut metrics in AJHotDraw after refactoring.

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
CmdCheckViewRef	commandExecute	0	{AbstractCommand.execute(), UndoCommand.execute(), RedoCommand.execute(), ToggleGridCommand.execute(), SendToBackCommand.execute(), DelectAllCommand.execute(), FigureTransferCommand.execute(), ChangeAttributeCommand.execute(), BringToFrontCommand.execute(), AlignCommand.execute()}	before	1	0	1
FigureSelection ObserverRole	-	-	-	-		-	-
FigureSelectionSubject Role	StandardDrawing ViewConstructor	0	{StandardDrawingView.new()}	after	1	0	1
	readObjectMethod	0	{StandardDrawingView.readObject()}	after	1	0	1
PersistentAttribute Figure	-	-	-	-		-	-

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.20: Measured values of pointcut metrics in AJHotDraw after refactoring (continued).

Aspect	Pointcut	*NNSAdP*	*SJP*	*NAdAsP*		*NOAsP*	*NAdP*
PersistentComposite Figure	-	-	-	-		-	-
PersistentDrawing	-	-	-	-		-	-
PersistentFigure	-	-	-	-		-	-
PersistentImageFigure	-	-	-	-		-	-
PersistentTextFigure	-	-	-	-		-	-
SelectionChanged Notification	invalidateSelFigure	0	{ StandardDrawingView.addToSelectionImpl(),Standard DrawingView .removeFromSelection()}	after	1	0	1
	clear_toggleSelection	0	{ StandardDrawingView.clearSelection(),StandardDrawi ngView.toggleSelection()}	after	1	0	1

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.21: Measured values of aspect metrics in Telecom before refactoring.

Aspect	*SUPAs*	*NPCAs*	*SCAs*	*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
Billing	{advice@line7}	0	-	1	1	2	7	4	0
TimerLog	{advice@line2, advice@line5}	0	-	0	2	2	0	0	0
Timing	{advice@line7}	0	-	0	2	2	2	2	1

Table D.22: Measured values of aspect metrics in Telecom after refactoring.

Aspect	*SUPAs*	*NPCAs*	*SCAs*	*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
Billing	$\emptyset$	0	-	1	1	2	7	4	0
TimerLog	$\emptyset$	0	-	0	2	2	0	0	0
Timing	$\emptyset$	0	-	0	1	2	2	2	0
XPI	$\emptyset$	0	-	0	1	0	0	0	2

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.23: Measured values of aspect metrics in Spacewar before refactoring.

Aspect	*SUPAs*	*NPCAs*	*SCAs*	*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
Coordinator	∅	0	-	0	1	2	0	17	0
Debug	{advice@line16 , advice@line61, advice@line69, advice@line76, advice@line80, advice@line85. advice@line90}	0	-	0	10	13	0	0	0
EnsureShipIsAlive	∅	1	-	0	0	1	0	0	0
GameSynchronization	∅	0	-	0	1	0	0	1	0
RegistrySynchronization	∅	0	-	0	1	0	0	1	0

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.24: Measured values of aspect metrics in Spacewar after refactoring.

Aspect	*SUPAs*	*NPCAs*	*SCAs*	*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
Coordinator	$\varnothing$	0	-	0	1	2	0	17	0
Debug	$\varnothing$	0	-	0	10	13	0	0	0
EnsureShipIsAlive	$\varnothing$	0	-	0	1	1	0	0	0
GameSynchronization	$\varnothing$	0	-	0	1	0	0	1	0
RegistrySynchronization	$\varnothing$	0	-	0	1	0	0	1	0

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.25: Measured values of aspect metrics in AspectTetris before refactoring.

Aspect	*SUPAs*	*NPCAs*	*SCAs*	*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
Counter	∅	0	-	0	5	6	0	3	0
DesignCheck	{declarewarning@line2, declarewraning@line4}	0	-	0	2	0	0	0	0
GameInfo	∅	0	-	0	1	1	0	0	0
Levels	∅	0	-	0	4	4	0	5	0
Menu	∅	0	-	0	2	2	0	5	0
NewBlocks	∅	0	-	0	5	5	0	2	0
NextBlock	∅	0	-	0	2	2	0	2	0
TestAspect	∅	0	-	0	1	1	0	0	0

Table D.26: Measured values of aspect metrics in AspectTetris after refactoring.

Aspect	*SUPAs*	*NPCAs*	*SCAs*	*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
Counter	∅	0	-	0	2	6	0	3	0
DesignCheck	∅	0	-	0	2	0	0	0	0
GameInfo	∅	0	-	0	0	1	0	0	0
Levels	∅	0	-	0	1	4	0	5	0
Menu	∅	0	-	0	1	2	0	5	0
NewBlocks	∅	0	-	0	5	5	0	2	0
NextBlock	∅	0	-	0	1	2	0	2	0
TestAspect	∅	0	-	0	1	1	0	0	0
XPI	∅	0	-	0	3	0	0	0	3

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.27: Measured values of aspect metrics in AJHotdraw before refactoring.

Aspect	*SUPAs*	*NPCAs*	*SCAs*		*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
CmdCheckViewRef	∅	0	-		0	1	1	0	0	0
FigureSelection ObserverRole	∅	0	Abstract Command	{figureSelectionChanged()}	0	0	0	12	0	0
			Undoable Command	{figureSelectionChanged()}						
			Drawing Editor	∅						
			DrawApplet	{figureSelectionChanged()}						
			Draw Application	{figureSelectionChanged()}						
			JavaDraw Viewer	{figureSelectionChanged()}						
FigureSelectionSubject Role	{advice@line3, advice@line9}	0	Drawing View	∅	0	2	2	7	3	0

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.28: Measured values of aspect metrics in AJHotdraw before refactoring (continued).

Aspect	*SUPAs*	*NPCAs*	*SCAs*		*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
			Standard Drawing View	{addFigureSelectionListener(), removeFigureSelectionListener(), fireSelectionChanged()}						
			Null Drawing View	{addFigureSelectionListener(), removeFigureSelectionListener()}						
PersistentAttribute Figure	∅	0	-		0	0	0	2	0	0
PersistentComposite Figure	∅	0	-		0	0	0	2	0	0
PersistentDrawing	∅	0	Drawing	∅	0	0	0	1	0	0
PersistentFigure	∅	0	Figure	∅	0	0	0	3	0	0
			Abstract Figure	{write(), read()}						

Table D.29: Measured values of aspect metrics in AJHotdraw before refactoring (continued).

Aspect	*SUPAs*	*NPCAs*	*SCAs*	*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
PersistentImageFigure	∅	0	-	0	0	0	2	0	0
PersistentTextFigure	∅	0	-	0	0	0	2	0	0
SelectionChanged Notification	∅	0	-	0	2	2	0	0	0

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.30: Measured values of aspect metrics in AJHotdraw after refactoring.

Aspect	*SUPAs*	*NPCAs*	*SCAs*		*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
CmdCheckViewRef	∅	0	-		0	1	1	0	0	0
FigureSelection ObserverRole	∅	0	Abstract Command	∅	0	0	0	12	0	0
			Undoable Command	∅						
			Drawing Editor	∅						
			DrawApplet	∅						
			Draw Application	∅						
			JavaDraw Viewer	∅						
FigureSelectionSubject Role	∅	0	Drawing View	∅	0	2	2	7	3	0

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.31: Measured values of aspect metrics in AJHotdraw after refactoring (continued).

Aspect	*SUPAs*	*NPCAs*	*SCAs*		*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
			Standard Drawing View	{addFigureSelectionListener(), removeFigureSelectionListener(), fireSelectionChanged()}						
			Null Drawing View	{addFigureSelectionListener(), removeFigureSelectionListener()}						
PersistentAttribute Figure	∅	0	-		0	0	0	2	0	0
PersistentComposite Figure	∅	0	-		0	0	0	2	0	0
PersistentDrawing	∅	0	Drawing	∅	0	0	0	1	0	0
PersistentFigure	∅	0	Figure	∅	0	0	0	3	0	0
			Abstract Figure	{write(), read()}						

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table D.32: Measured values of aspect metrics in AJHotdraw after refactoring (continued).

Aspect	*SUPAs*	*NPCAs*	*SCAs*	*NAMA*	*NPAs*	*NAdAs*	*NIAs*	*NMAs*	*SNOAsP*
PersistentImageFigure	∅	0	-	0	0	0	2	0	0
PersistentTextFigure	∅	0	-	0	0	0	2	0	0
SelectionChanged Notification	∅	0	-	0	2	2	0	0	0

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX E

USER MANUAL OF A TOOL FOR DETECTING AO BAD SMELLS IN ASPECTJ CODE

The supporting tool for detecting AO bad smells is developed as a plug-in to the Eclipse program tool. This user manual is structured into two parts. The first part describes on how to install the plug-in. The second part shows the utilization of the plug-in.

E.1 Tool Installation

The plug-in can manually be installed by extracting the zip file of developed plug-in

into the plugins directory of Eclipse for example C:\eclipse\plugins\, while Eclipse does not run.

Bad Smell Detector menu on the Eclipse's menu bar will appears when Eclipse runs as shown in Figure E.1.

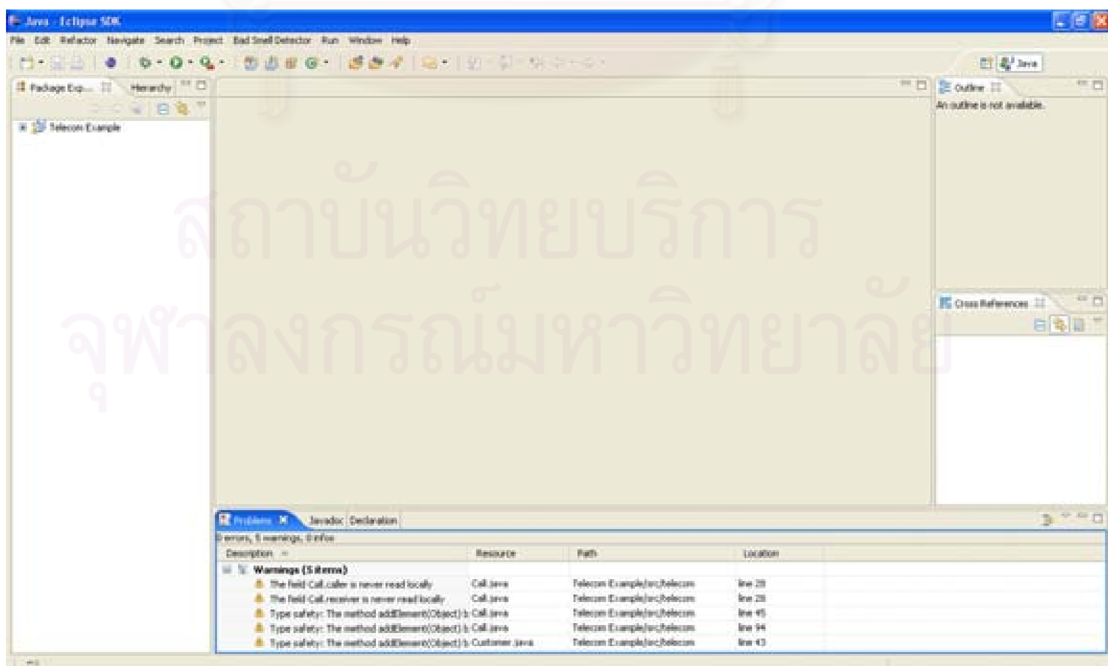


Figure E.1: Main preference page of Eclipse after extracting the plug-in's zip file.

AO Bad Smells View is an Eclipse View perspective that is used to illustrate all candidates of AO bad smells in all opened projects presented on the Eclipse Package Explorer perspective. AO Bad Smells View is shown in Figure E.2.

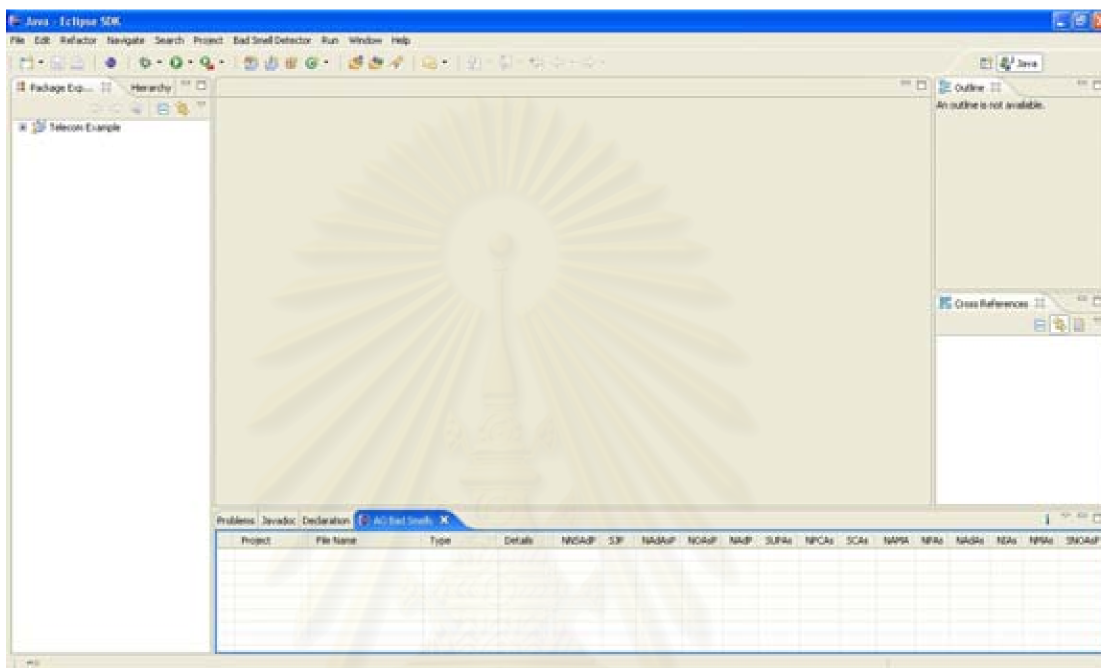


Figure E.2: AO Bad Smells View.

If AO Bad Smells View does not appear, user can open the View perspective by selecting Window->Show View->Others… on the Eclipse's menu bar as shown in Figure E.3.

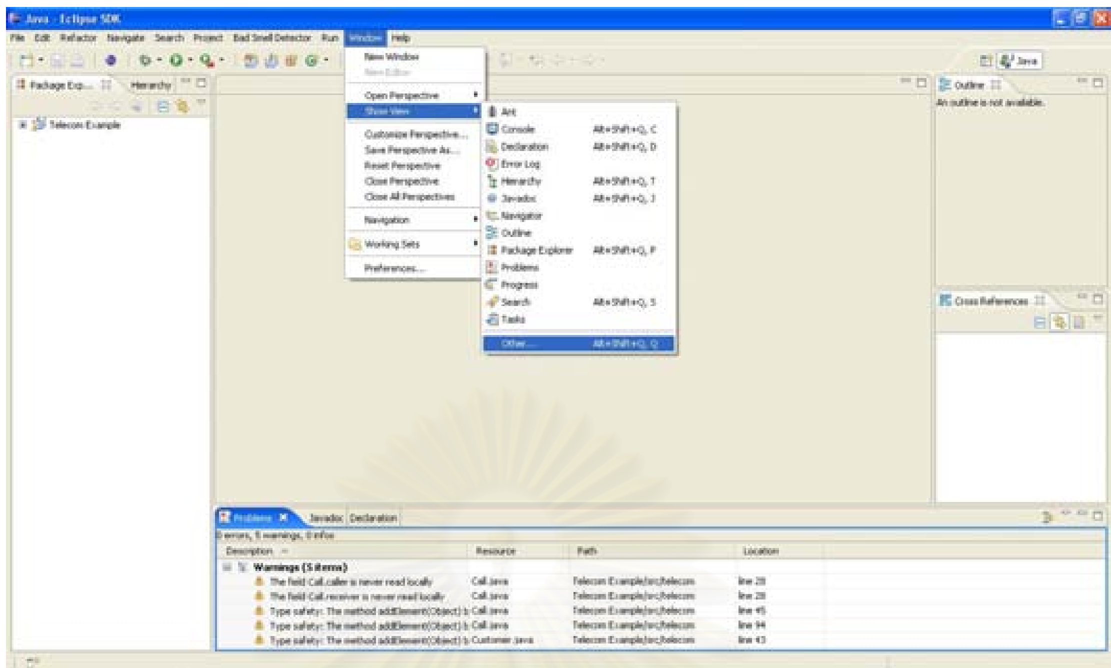


Figure E.3: Opening AO Bad Smells View.

After that, the Show View dialog will appear to ask user about which view perspectives dose he desires as shown in Figure E.4. To open the AO Bad Smells View, double click on the Bad Smells folder to open it and select the AO Bad Smells View. Then, click OK button.

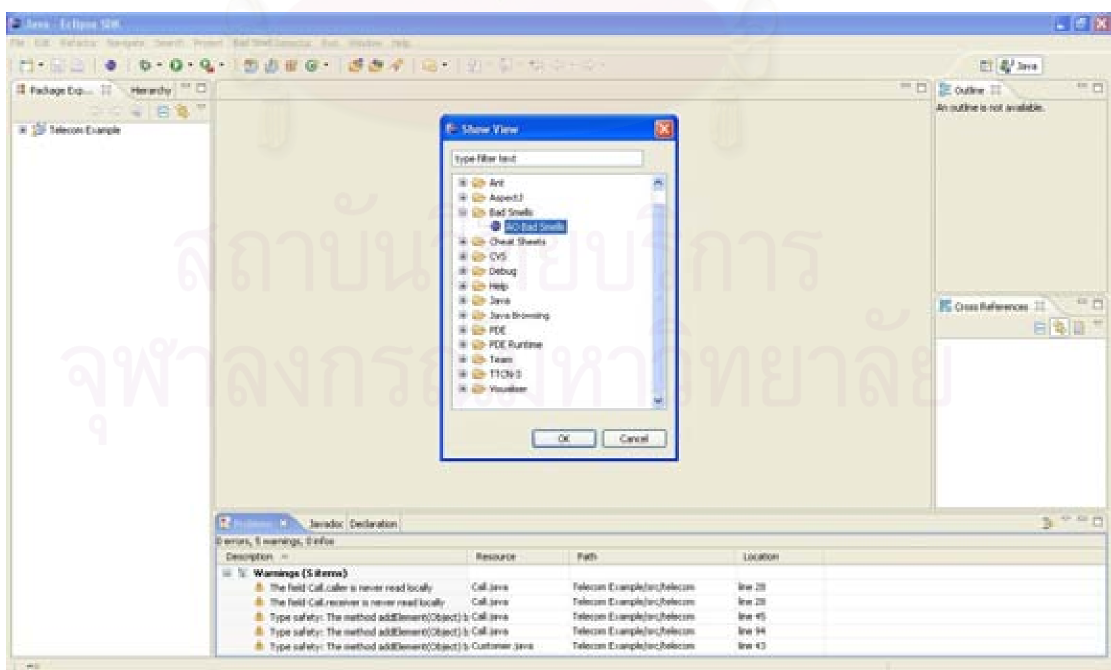


Figure E.4: Show View dialog.

E.2 Tool Utilization

To detect AO bad smells, a program which user want to detect the AO bad smells must be appear on the Package Explorer perspective. After that, select Bad Smell Detector->Detect Aspect-Oriented Bad Smells on the Eclipse's menu bar as shown in Figure E.5.

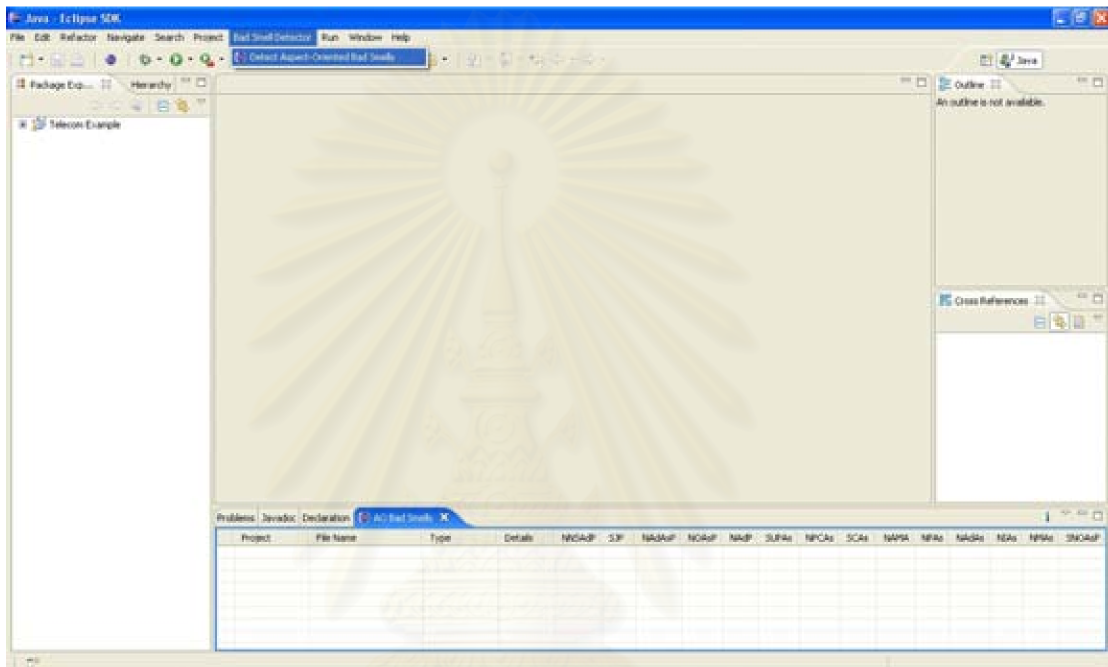


Figure E.5: Bad Smell Detector menu.

Consequently, measured values of all bad-smell metrics and all candidates of AO bad smells are presented on the AO Bad Smells View as shown in Figure E.6.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

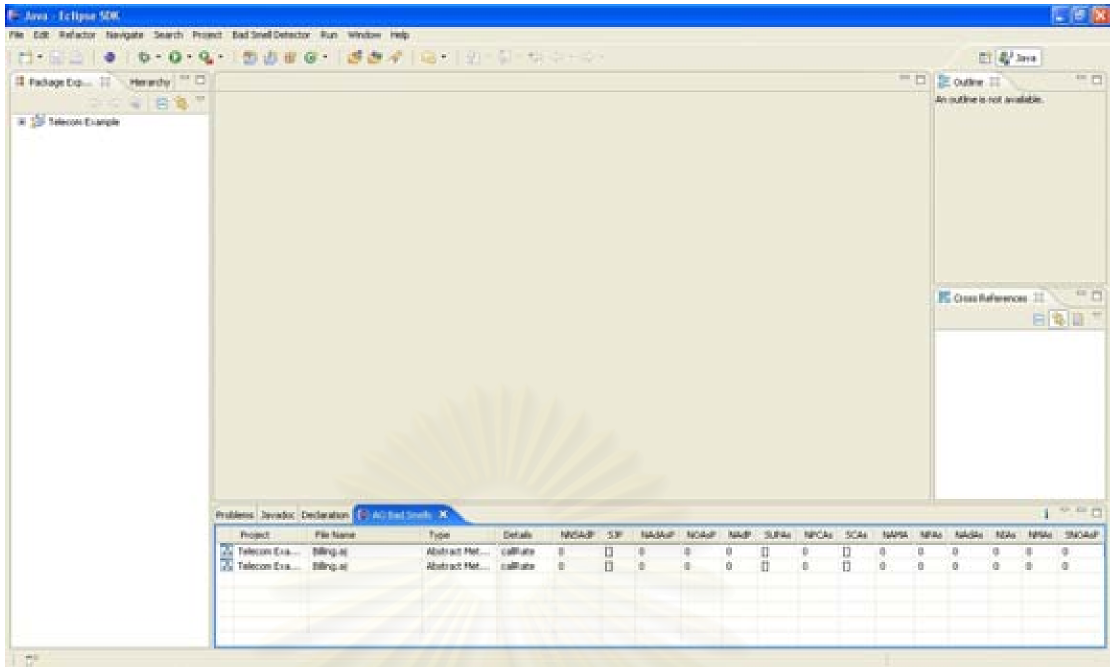


Figure E.6: An example of detection results.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

BIOGRAPHY

Name	Komsan Srivisut
Sex	Male
Date of Birth	October 21, 1981
Place of Birth	Phrae, Thailand
Education:	
2007	M.Eng. in Computer Engineering, Chulalongkorn University
2003	B.Eng. in Computer Engineering, Suranaree University of Technology

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย