วิธีการเมตาโปรแกรมและการเรียนรู้ของเครื่องสำหรับ
การตรวจจับข้อบกพร่องของซอฟต์แวร์เชิงวัตถุ

นายสาคร เมฆรักษาวนิช

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2553

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

A META-PROGRAM AND MACHINE LEARNING APPROACH FOR DETECTING

OBJECT-ORIENTED SOFTWARE DESIGN FLAWS

Mr. Sakorn Mekruksavanich

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

A Dissertation Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

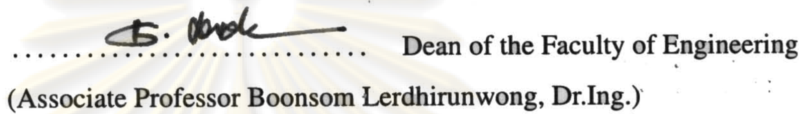
Chulalongkorn University

Academic Year 2010

Copyright of Chulalongkorn University

Thesis Title	A META-PROGRAM AND MACHINE LEARNING APPROACH FOR DETECTING OBJECT-ORIENTED SOFTWARE DESIGN FLAWS
By	Mr. Sakorn Mekruksavanich
Field of Study	Computer Engineering
Thesis Advisor	Associate Professor Pornsiri Muenchaisri, Ph.D.

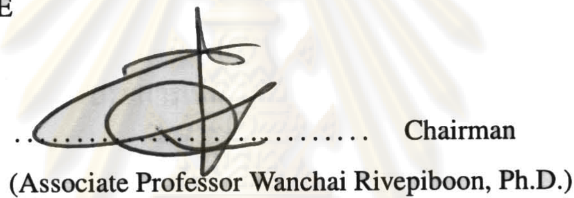
Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctoral Degree



.... Dean of the Faculty of Engineering

(Associate Professor Boonsom Lerdhirunwong, Dr.Ing.)

THESIS COMMITTEE



.... Chairman

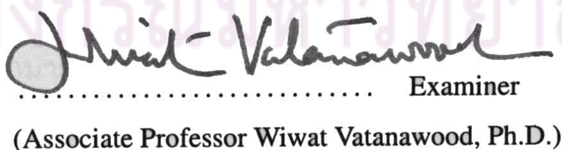
(Associate Professor Wanchai Rivepiboon, Ph.D.)

.... Thesis Advisor

(Associate Professor Pornsiri Muenchaisri, Ph.D.)

.... Examiner

(Professor Prabhas Chongstitvatana, Ph.D.)



.... Examiner

(Associate Professor Wiwat Vatanawood, Ph.D.)

.... External Examiner

(Assistant Professor Songsak Rongviriyapanich, Ph.D.)

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

สาคร เมฆรักษาวนิช : วิธีการเมตาโปรแกรมและการเรียนรู้ของเครื่องสำหรับการตรวจจับข้อบกพร่องของซอฟต์แวร์เชิงวัตถุ. (A META-PROGRAM AND MACHINE LEARNING APPROACH FOR DETECTING OBJECT-ORIENTED SOFTWARE DESIGN FLAWS) อ. ที่ปรึกษาวิทยานิพนธ์หลัก: รศ.ดร. พรศิริ หมื่นไชยศรี, 156 หน้า.

ข้อบกพร่องของการออกแบบถูกใช้เป็นวิธีการในการระบุชนิดของปัญหาในระบบซอฟต์แวร์เชิงวัตถุ ซึ่งส่งผลโดยตรงทำให้คุณภาพซอฟต์แวร์ลดลง เช่นความสามารถในการบำรุงรักษา เพราะฉะนั้นข้อบกพร่องทางการออกแบบเหล่านี้ต้องถูกตรวจจับเพื่อหลีกเลี่ยงผลกระทบในทางลบในขั้นตอนการพัฒนาและบำรุงรักษาระบบซอฟต์แวร์ อย่างไรก็ตาม ในปัจจุบันในทางปฏิบัติ เทคนิคและวิธีการในการตรวจจับข้อบกพร่องสามารถแก้ปัญหาได้เพียงบางส่วนเท่านั้นโดยเฉพาะในเชิงสมรรถนะและประสิทธิภาพในการตรวจจับ เทคนิคการตรวจสอบซอฟต์แวร์ได้ถูกนำมาใช้เพื่อแก้ปัญหาข้อบกพร่องของการออกแบบ อย่างไรก็ตาม เทคนิคนี้นำไปสู่ผลเสียบางประการเช่นใช้เวลานาน นอกจากนี้มีการนำมาตรวัดซอฟต์แวร์มาใช้เป็นเทคนิคการตรวสอบซอฟต์แวร์แบบอัตโนมัติ กลยุทธ์ของเทคนิคนี้คือการตรวจจับค่าความเบี่ยงเบนจากหลักการออกแบบและฮิวริสติกที่ดีโดยใช้ค่าขีดแบ่ง ดังนั้นประสิทธิภาพในการตรวจจับจึงขึ้นอยู่กับการปรับค่าขีดแบ่งให้เหมาะสมที่สุดซึ่งเป็นงานที่ยากยิ่ง

ในวิทยานิพนธ์นี้ได้นำเสนอวิธีการตรวจจับสำหรับระบบซอฟต์แวร์เชิงวัตถุแบบใหม่โดยใช้เทคนิคการโปรแกรมเมตาแบบการอธิบายร่วมกับเทคนิคการเรียนรู้แบบอธิบาย ในวิธีการที่นำเสนอนี้ เทคนิคการโปรแกรมเมตาแบบการอธิบาย ถูกใช้เพื่อเป็นตัวแทนองค์ประกอบเชิงวัตถุและความสัมพันธ์โดยแสดงในรูปของกฎทางตรรกะในระดับเมตา เพื่อใช้อธิบายข้อบกพร่องของการออกแบบ เทคนิคการเรียนรู้แบบอธิบายถูกใช้เพื่ออนุมานแบบรูปโดยการเรียนรู้แบบการอนุมานสำหรับคุณสมบัติของข้อบกพร่องบางอย่างที่ยากต่อการทำความเข้าใจ วิธีการที่นำเสนอนี้สามารถตรวจจับข้อบกพร่องได้อย่างมีประสิทธิภาพโดยการไม่นำข้อจำกัดของค่าเริ่มต้นเฉพาะในแต่ละสภาพแวดล้อมมาพิจารณาในการตรวจจับและส่งเสริมการตรวจจับในรูปแบบอัตโนมัติเพื่อลดค่าใช้จ่ายและเวลาในกระบวนการตรวจจับ กรณีศึกษาหลายกรณีถูกนำมาใช้เพื่อประเมินผลวิธีการตรวจจับที่นำเสนอ

ภาควิชา...วิศวกรรมคอมพิวเตอร์... ลายมือชื่อนิสิต [signature]
สาขาวิชา...วิศวกรรมคอมพิวเตอร์... ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก [signature]
ปีการศึกษา...2010...

4871878221: MAJOR COMPUTER ENGINEERING

KEYWORDS: DESIGN FLAW / OBJECT-ORIENTED DESIGN / RULE-BASED DETECTION / EXPLANATION-BASED LEARNING / PROLOG-EBG

SAKORN MEKRUKSAVANICH : A META-PROGRAM AND MACHINE LEARNING APPROACH FOR DETECTING OBJECT-ORIENTED SOFTWARE DESIGN FLAWS. ADVISOR : ASSOC. PROF. PORNSIRI MUENCHAISRI, Ph.D., 156 pp.

Design flaws are used as a mean to identify problematic classes in object-oriented software systems which directly decrease software quality, such as maintainability. Therefore such design flaws must be identified to avoid their possible negative consequences on development and maintenance of software systems. However, in recent practice, techniques and methodologies of design flaw detection can solve only some points especially in performance and efficiency of the detection. The software inspection technique is introduced to deal with design flaw problems. It, however, leads to some different issues such as time consumption. An additional proposed automated technique is software metrics. The strategies of this technique capture deviations from good design principles and heuristics by threshold values. Thus effective identifying depends on optimized threshold which is a difficult task.

This dissertation proposes a new detection methodology for object-oriented software system by using Declarative Meta Programming and Explanation-Based Learning technique. In the proposed approach, Declarative Meta-Programming is used to represent specific object-oriented elements and their relations in form of logic rules in meta level for describing design flaws. Explanation-Based Learning is used for extrapolating pattern by deductive learning for some characteristics of design flaws that are difficult to understand. The proposed methodology can efficiently detect design flaws by disregarding limitations of specific thresholds in each environment of detection and promoting the automatic detection for reducing cost and time consumption in the detection process. Case studies are conducted to evaluate the proposed detection approach.

Department: Computer Engineering | Student's Signature Sakorn Mekruksavanich

Field of Study: Computer Engineering | Advisor's Signature Pornsiri Muenchaisri

Academic Year: 2010

ศูนย์วิทยทรัพยากร จุฬาลงกรณ์มหาวิทยาลัย

Acknowledgements

First and foremost, I consider myself fortunate and privileged to have Assoc. Prof. Pornsiri Muenchaisri as my dissertation advisors. I am deeply indebted to her for shaping my full path to research by guiding me with her extensive knowledge and with her insightful discussions and questions. I would have never reached the point where I stand today without her fully supports.

I am honored and especially grateful to have Assoc. Prof. Wanchai Rivepiboon, Prof. Prabhas Chongstitvattana and Assoc. Prof. Wiwat Vatanawood as the chairman and examiners respectively, including Asst. Prof. Songsak Rongviriyapanich as an external examiner to be my thesis committee. I also wish to express my thanks to them for their valuable advices, reading and criticizing the manuscript.

My thankful admiration goes to Prof. James M. Bieman and Prof. Robert B. France for decisively influencing the new beginning of my scientific researcher on my research visit at Colorado State University. Without their guidance, support and enthusiasm for all of my early achievements, probably none of those dreams would have become reality.

Very special thanks to all my Ph.D. student friends who did not let me forget that life is more than writing a Ph.D. thesis. I am deeply indebted to "The three musketeers"– Jiradej Ponsawat, Narit Hnoohom and Anuchit Jitpattanakul – for all their brotherhood supports. Moreover this work would not have been possible without the whole environment in the CESELab at Chulalongkorn University. I would like to thank all my colleagues and friends at this laboratory.

I would like to warmly thank my parents for all their love, for all their mental and financial support and for believing in me even when I did not. In particular, I am very thankful to my mother for the "academic answers of real life "to all the profound questions of my childhood and to my father for teaching me to approach all things systematically.

My most special thanks are for Apinya and Karnteera for the unique way in which they loves and understands me day by day. I am so grateful to her, Apinya, for the many times she put aside her wishes and plans, and encouraged me to finish writing this thesis. Thank you for teaching me daily the "unconditional equations of love". I am so proud and grateful to have them by my side.

Thank you again, I could not have done this without all of you.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Contents

Page

Abstract (Thai) iv

Abstract (English) v

Acknowledgements vi

Contents vii

List of Tables xi

List of Figures xv

Chapter

I Introduction **1**

1.1 Motivation 2

1.2 Objectives of Study 4

1.3 Scopes of Study 4

1.4 Contributions 5

1.5 Research Methodology 5

1.6 Organization 6

II Object Oriented Design Flaws and Its Detection **7**

2.1 Object-Oriented Paradigm and Design Flaws 7

2.1.1 Object-Oriented Programming 7

2.1.1.1 Data Abstraction 8

2.1.1.2 Encapsulation 9

2.1.1.3 Modularity 10

2.1.1.4 Inheritance 10

2.1.1.5 Interfaces and Polymorphism 11

2.1.2 The Good Object-Oriented Design 12

2.1.3 Object-Oriented Design Flaws 13

2.1.3.1 Design Flaw Taxonomy 14

2.2 Declarative Paradigm in Meta-Programming 17

2.2.1 Declarative Paradigm 17

2.2.2 Logic Programming 18

2.2.3 Declarative Meta Programming 19

2.2.4 Logic Programming Theory 20

2.2.4.1 Terminology 20

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Chapter Page

2.2.4.2 The syntax of logic programs . . . 22
2.2.4.3 Properties of logic languages . . . 22
2.2.4.4 Model theory . . . 23
2.3 Machine Learning . . . 24
2.3.1 Analytical and Empirical Learning . . . 25
2.3.2 Analytical Learning . . . 25
2.3.3 Explanation-Based Theory . . . 27
2.4 Related Works . . . 29
2.4.1 Smells, Design flaws and Anti-patterns . . . 30
2.4.2 Analysis based on structural detection . . . 31
2.4.3 Analysis based on metrics . . . 32
2.4.4 Usability and Efficient Flaw Detection . . . 33
2.4.5 Tools of detection . . . 35

III The Proposed Flaw Detection Methodology . . . 38

3.1 An Overview of Detection Architecture . . . 38
3.2 The Proposed Detection Methodology . . . 40
3.3 Detection Methodology in Details . . . 42
3.3.1 Step 1: Building Block Synthesis . . . 42
3.3.2 Step 2: Concretization . . . 46
3.3.2.1 A concretization of `BBs` . . . 46
3.3.2.2 Meaning functions of R_{BBs} relations . . . 50
3.3.3 Step 3: Generalization . . . 52
3.3.3.1 Process 3.1: Arrangement . . . 52
3.3.3.2 Process 3.2: Explanation . . . 54
3.3.3.3 Process 3.3: Analyzing . . . 55
3.3.3.4 Process 3.4: Refinement . . . 56
3.3.4 Step 4: Procedure Generation . . . 57
3.3.5 Step 5: Code Analysis . . . 57
3.3.6 Step 6: Fact Specification . . . 58
3.3.7 Step 7: Detection . . . 58
3.3.8 Step 8: Validation . . . 59

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Chapter Page

IV Evaluation and Discussion . . . **60**

4.1 The validation of the proposed methodology . . . 61

4.1.1 Validation Process . . . 61

4.2 Case Studies . . . 62

4.2.1 Case I : `CommonCLI` . . . 63

4.2.2 Case II : `JUNIT` . . . 64

4.2.3 Case III : `GANTTPROJECT` . . . 66

4.3 Result discussion . . . 67

4.3.1 Result discussion of Case I . . . 68

4.3.2 Result discussion of Case II . . . 68

4.3.3 Result discussion of Case III . . . 72

4.3.4 Overall discussions of the proposed detection approach . . . 72

V Conclusion . . . **75**

5.1 Conclusion of dissertation . . . 75

5.2 Future research directions . . . 76

References . . . 77

Appendices . . . 84

Appendix A Examples and Domain theories of Research . . . **84**

1.1 The Bloaters Category . . . 84

1.1.1 The Long Parameter List . . . 84

1.1.2 Large Class . . . 86

1.1.3 Primitive Obsession . . . 92

1.1.4 Data Clump . . . 94

1.2 The Object-Oriented Abusers Category . . . 97

1.2.1 Switch Statements . . . 97

1.2.2 Temporary Field . . . 98

1.2.3 Refused Bequest . . . 100

1.2.4 Alternative Classes with Different Interfaces . . . 106

1.3 The Change Preventers Category . . . 111

1.3.1 Divergent Change . . . 111

Chapter Page

1.3.2 Shotgun Surgery . . . 114
1.3.3 Parallel Inheritance Hierarchies . . . 116
1.4 The Dispensables Category . . . 119
1.4.1 Lazy Class . . . 120
1.4.2 Data class . . . 121
1.4.3 Duplicate Code . . . 124
1.4.4 Dead Code . . . 126
1.5 The Couplers Category . . . 128
1.5.1 Feature Envy . . . 128
1.5.2 Inappropriate Intimacy . . . 130
1.5.3 Message Chains . . . 133
1.5.4 Middle Man . . . 135

Appendix B Meta Element Specifications . . . 138
2.1 Sourcefolders . . . 138
2.2 Building-Blocks Level . . . 139
2.3 Interface-Level Attributes . . . 143
2.4 Body Level Attributes . . . 155
Biography . . . 156

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

List of Tables

Table Page

1.1 Data Class metrics . . . 3

1.2 Measured values of Class `BillItem` . . . 4

2.1 The explanation-based generalization problem . . . 27

2.2 The explanation-based learning algorithm Prolog-EBG. . . . 29

3.1 Text descriptions of *Data Class* and *Long Parameter List* flaws . . . 44

3.2 Domain theories and a target concept of the `FilterMap` Data Class . . . 54

3.3 An example for regressing a set of literals given by *Frontier* through *methodOperation Rule* . . . 56

3.4 The final rule of `FilterMap` Data Class . . . 56

4.1 The result of *Bloaters* flaw detection in `CommonCLI v1.0` . . . 64

4.2 The result of *The Object-Orientation Abusers* flaw detection in `CommonCLI v1.0` . . 64

4.3 The result of *The Change Preventers* flaw detection in `CommonCLI v1.0` . . . 64

4.4 The result of *The Dispensables* flaw detection in `CommonCLI v1.0` . . . 65

4.5 The result of *The Couplers* detection in `CommonCLI v1.0` . . . 65

4.6 The result of *Bloaters* flaw detection in `JUNIT v1.3.6` . . . 66

4.7 The result of *The Object-Orientation Abusers* flaw detection in `JUNIT v1.3.6` . . . 66

4.8 The result of *The Change Preventers* detection in `JUNIT v1.3.6` . . . 67

4.9 The result of *The Dispensables* flaw detection in `JUNIT v1.3.6` . . . 67

4.10 The result of *The Couplers* flaw detection in `JUNIT v1.3.6` . . . 68

4.11 Specificity and its false positive rate of Data Class detection with other detection techniques in `JUNIT v1.3.6` . . . 69

4.12 Precision and recall of design flaws in `GANTTPROJECT V1.10.2` (Compare with Metric-Based Approach) . . . 70

A.1 Domain theories and a target concept of Ex.1- Ex.4 *Long Parameter List* . . . 85

A.2 Domain theories and a target concept of *Large Class* design flaw . . . 89

A.3 Domain theories and a target concept of *Primitive Obsession* design flaw . . . 93

A.4 Domain theories and a target concept of *Data Clump* design flaw . . . 95

A.5 Domain theories and a target concept of *Switch Statements* design flaw . . . 98

A.6 Domain theories and a target concept of *Temporary Field* design flaw . . . 99

A.7 Domain theories and a target concept of *Refused Bequest* design flaw . . . 104

A.8 Domain theories and a target concept of *Alternative Classes with Different Interfaces* design flaw . . . 110

A.9 Domain theories and a target concept of *Divergent Change* design flaw . . . 113

Table Page

A.10 Domain theories and a target concept of *Shotgun Surgery* design flaw 115
A.11 Domain theories and a target concept of *Parallel Inheritance Hierarchies* design flaw . 118
A.12 Domain theories and a target concept of *Lazy Class* design flaw 120
A.13 Domain theories and a target concept of *Data Class* design flaw 122
A.14 Domain theories and a target concept of *Duplicate Code* design flaw 125
A.15 Domain theories and a target concept of *Dead Code* design flaw 127
A.16 Domain theories and a target concept of *Feature Envy* design flaw 129
A.17 Domain theories and a target concept of *Inappropriate Intimacy* design flaw 132
A.18 Domain theories and a target concept of *Message Chains* design flaw 135
A.19 Domain theories and a target concept of *Middle Man* design flaw 137
B.1 packageT 138
B.2 compilationUnitT 138
B.3 importT 138
B.4 classT 139
B.5 methodT 139
B.6 constructorT 139
B.7 classInitializerT 140
B.8 enumConstantT 140
B.9 fieldT 140
B.10 paramT 140
B.11 typeParamT 141
B.12 annotationT 141
B.13 memberValueT 141
B.14 annotationMemberT 142
B.15 annotatedT 142
B.16 commentT 142
B.17 interfaceT 143
B.18 externT 143
B.19 enumT 143
B.20 annotationTypeT 143
B.21 markerAnnotationT 144
B.22 modifierT 144
B.23 implementsT 144
B.24 extendsT 144

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table Page

B.25 assertT . . . 145
B.26 assignT . . . 145
B.27 blockT . . . 145
B.28 callT . . . 146
B.29 caseT . . . 146
B.30 conditionalT . . . 146
B.31 doWhileT . . . 147
B.32 execT . . . 147
B.33 forT . . . 147
B.34 foreachT . . . 148
B.35 getFieldT . . . 148
B.36 identT . . . 148
B.37 indexedT . . . 149
B.38 labelT . . . 149
B.39 literalT . . . 149
B.40 localT . . . 150
B.41 newArrayT . . . 150
B.42 newClassT . . . 150
B.43 nopT . . . 151
B.44 operationT . . . 151
B.45 precedenceT . . . 151
B.46 returnT . . . 151
B.47 selectT . . . 152
B.48 switchT . . . 152
B.49 synchronizedT . . . 152
B.50 throwT . . . 153
B.51 tryT . . . 153
B.52 typeCastT . . . 153
B.53 typeTestT . . . 154
B.54 typeRefT . . . 154
B.55 whileT . . . 154
B.56 omitArrayDeclarationT . . . 155
B.57 inlineDeclarationT . . . 155
B.58 inlinedT . . . 155

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table | Page

B.59 variableArgumentT 155

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

List of Figures

Figure Page

1.1 Class BillItem 3

2.1 Checkstyle tool 35

2.2 FindBugs tool 36

3.1 The draft design of description detection 39

3.2 The meta architecture of description detection 40

3.3 The Proposed Detection Methodology 41

3.4 UML Model for the notation of the proposed domain model 51

3.5 Generalization Process of Step 3: Generalization 52

3.6 Class FilterMap 53

3.7 A explanation of FilterMap class (a Data Class flaw) 55

3.8 Step 5: Code Analysis and Step 6: Fact Specification 58

4.1 The precision rate of design flaws detection in GANTTPROJECT V1.10.2 (Compare with Metric-Based Approach). 67

4.2 The recall rate of design flaws detection in GANTTPROJECT V1.10.2 (Compare with Metric-Based Approach). 68

4.3 The average precision rate of proposed detection in CommonCLI 71

4.4 The average precision rate of proposed detection in JUNIT 71

A.1 The sematic net shows covering of Ex3. and Ex4. in domain theories 86

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER I

INTRODUCTION

In the recent software technology development, most object-oriented software systems are developed under evolutionary process models. Most software that is related to a real-world problem domain, must continuously evolve to cope with the problem domain changes — requirement and environment changes (Bravo, 2003). Object-oriented software design principles and heuristics (Gamma et al., 1995; Riel, 1996) are proposed to promote the good quality software. However, even when maintainers are familiar with those techniques, violation of these design rules may lead to *poor* solutions by deadline pressure, excessive focusing on pure functionality, or inexperience programming. Such solutions to recurring design and implementation problems hinder software evolution. They — low-level or local problems — are called *design flaws* (Marinescu, 2004).

Design flaws are potential errors of the internal organization of an object-oriented software system that have a negative impact on important quality factors eg., maintainability, understandability, ease of evolution, etc. (Fowler, 1999; Mens and Tourwé, 2004). They denote both source code and design artifacts. This can be concluded that, in a particular context, design flaws are in-between design and implementation. They may concern the design of a class, but they concretely manifest themselves in the source code as a class with specific implementation. In the recent literature, design flaws are referred as *Bad Smells* (Fowler, 1999; Riel, 1996) and *AntiPatterns* (Brown et al., 1998). Many authors denote design flaws normally by using metaphors.

One example of a design flaw is *God Class* (Brown et al., 1998), which is a characteristic of a procedural thinking in object-oriented programming. The classes of design (the God Class) is responsible for all (or most of the) behavior of an application while the rest of the classes (the Data Classes) are only responsible for encapsulating data. This type of designs shows a wrong distribution of responsibilities. The components of the god class are also not cohesive. This flaw does not exploit object-oriented mechanisms, such as encapsulation and modularity (Johnson and Foote, 1988).

The detection of flaws can substantially reduced the cost of subsequent activities in the development and maintenance phases (Pressman, 2001). Several approaches have been proposed to specify and detect flaws. Firstly, a manual detection of design flaws by software inspections (Travassos et al., 1999) is presented to detect flaws problems. It however leads to some different issues as time-expensive, non-repeatable and non-scalable (Langelier et al., 2005). Even more issues concerning the manual detection of design flaws are identified by Mäntylä et. al.

(Mäntylä et al., 2004; Mäntylä et al., 2003). They show that as the experience developer has with a certain software system increases, his ability to perform an objective evaluation of the system as well as his ability to detect design flaws decreases.

To avoid some drawbacks with a purely manual detection approach, metric-based heuristics for identifying design flaws in software systems are proposed (Marinescu, 2004). The strategies capture deviations from good design principles and consist of combining metrics with set operators and compare their values against threshold values. Therefore, effective identifying depends on proper metrics and thresholds which are used to detect such flaws.

Several approaches are proposed to detect design flaws. However, they have three limitations that challenging researcher to discover and find out. First, the formal representation that joins systematically and clearly an analysis process leading to a specification for detection. Second, detection always depends on threshold values that being coarse-grained characteristics. These characteristics affect directly to the accuracy of detection. Finally, results of detection from research works did not compared among approaches.

This dissertation proposes a different approach to detect design flaws in object-oriented software by using the *Meta-Programming* approach with machine learning. The proposed approach is a novelty technique that applies *Declarative Meta-programming* and *Machine Learning* to detect design flaws. In the approach, Declarative Meta-Programming is used to represent specific relations of elements in form of logic rules for describing design flaws. Machine learning is used for extrapolating pattern for some characteristics of design flaws that are difficult to understand. With this approach, design flaws of an object-oriented system can be detected at the meta-level in the Declarative Meta-Programming. With this declarative, paradigm design flaws can be detected in a simple way and the reliable detection results are obtained. The case studies are also presented to visualize the proposed approach concretely.

1.1 Motivation

Many research work emphasizes on identification and wildly discussion of anomalous problems in various research fields eg., software testing (Van Rompaey et al., 2007), networking (Patcha and Park, 2007) and databases (Bruno et al., 2007; Jorwekar et al., 2007). Moreover in software development and maintenance cycles, identification of design flaw problems is performed in requirement analysis, design and implementation phases.

Although detection techniques and methodologies are proposed in the literature, there are some points to consider in research works. First, the detection still depends on a metric value

(or a group of metric values) to designate the risk area in the systems. Because of coarse-grain judgement, the problem in these situations is that the detection may miss many real flaws or it will introduce probably large number of *under-fitting feature detection* such as false positives in which negative flaws from the particular set are viewed as positive flaws. Next in software inspection, performance and accuracy rate of detection depend directly on the experience of practitioners.

```
final public class BillItem{
   private String bDescription;
   private double bCost;

   public BillItem() {
       bDescription = "None";
       BCost = 0.0;
   }
   public BillItem(String description, double cost)
       bDescription = description;
       bCost = cost;
   }
   public String getDescription(){
       return bDescription;
   }
   public double getCost(){
       return bCost;
   }
}
```

Figure 1.1: Class `BillItem`

Table 1.1: Data Class metrics

Name	Description
WOC (Weight of Class)	Number of non-accessor methods in a class divided by the total number of members of the interface.
NOPA (Number of Public Attributes)	The number of non-inherited attributes that belong to the interface of a class.
NOAM (Number of Accessor Methods)	The number of non-inherited accessor methods declared in the interface of a class.

Figure1.1 shows a motivating example of class `BillItem` of `HotelSystem` application in JAVA source code. This class is a Data Class Flaw by manual investigation. Table 1.1 shows three metrics and their description for metric-based detection of Data Class flaw (Marinescu, 2001). The existence with Data Class flaw in such class occurs when one of the threshold value excess the determined value. One interesting result is the false identification of the `BillItem` class as a flaw of *Data Class* by metric-based detection. From the details in Table 1.2, The calculation of WOC for class `BillItem` yields a value of 0.5. This value fails to trigger a metric

Table 1.2: Measured values of Class `BillItem`

Metrics	Measured values of Class BillItem	Threshold value
WOC	0.5	≥0.66
NOPA	0	≥3
NOAM	2	≥3

threshold value of Data Class which the value should be equal or more than 0.66. Moreover *NOPA* and *NOAM* measured values also show in a negative result because both *NOPA* and *NOAM* are not more than the Data Class threshold value which can indicate that class `BillItem` is a Data Class. When class `BillItem` is identified as a Data Class by manual inspection, it is not picked up by any of the threshold values of metric-based detection. Therefore in this context, the value of the metric for flaw detection is quite incorrect result, and trying to obtain an accurate calculation is also difficult.

By these reasons, it inspires motivations to our work. It is possible or not that we can ignore these threshold values. Can the existence of flaws be explained by some giving reasonings? If we can give reasoning for happened flaws by good design and heuristic, we believe that these reasonings could support our detection technique. And in code inspection, can we describe specification of design flaws in formalization? If it is possible, time-consuming and error-proneness can be reduced or eliminated in the detection process.

1.2 Objectives of Study

The objectives of study are as follows:

- To propose a new approach for design flaws detection in object-oriented software using *Declarative Meta-Programming* technique and *Machine Learning* technique that can detect design flaws and extrapolate rules in order to enhance flaw detection coverage.
- Both patterned design flaws and quantitative design flaws are to be detected.

1.3 Scopes of Study

The scopes of this study are as follows:

- The research considers the problem of design flaws.

- The research is to apply the approach by using Declarative meta-programming technique and Machine learning technique to find five design flaw categories in Folwer's literature (Fowler, 1999).
- The approach performs flaw detection of java source code.

1.4 Contributions

This research will make the following contributions:

- A novel detection approach can detect design flaws for object-oriented software which may increase accuracy rate of design flaw detection as well as disregard the threshold value needed.
- A new tool can automatically detect design flaws in object-oriented software.

1.5 Research Methodology

The research methodology is the following step:

- Survey related researches in the fields of design flaws detection in object-oriented systems, Declarative meta-programming and Machine learning technique.
- Propose a novel approach that using Declarative meta-programming and Machine learning technique to detect design flaws.
- Apply the declarative meta-programming to detect some patterned design flaws without learning in learning systems.
- Develop a system with the proposed approach to detect patterned design flaws (including learning systems).
- Develop system with the proposed approach to detect quantitative design flaws.
- Compare results of proposed detection method with a metric-based design flaws detection method.
- Publish at least one journal article relating to the work.
- Conclude and prepare the dissertation

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

1.6 Organization

Chapter 2 defines some theoretical backgrounds that will be used further on in this dissertation. The background covers the fundamental concepts used in the object-oriented paradigm and learning mechanism used to extrapolate patterns. A state of the art in the fields of design flaw detection related to object-oriented design is also proposed in this chapter.

Chapter 3 proposes the detection methodology for design flaws detection. The methodology is described mainly in eight steps. The representation is shown in input/output scheme for more understanding.

Chapter 4. For the proposed methodology, we introduce a methodology for detecting flaws related to design of software. The evaluation of the methodology and discussion of the results are presented.

Finally, Chapter 5 concludes research work and some directions for the future are discussed.

CHAPTER II

OBJECT ORIENTED DESIGN FLAWS AND ITS DETECTION

As pointed out in the first chapter that this work is going to tackle the issue of using relation in order to assess and control the quality of object-oriented design. By moving toward this point, this dissertation is therefore related to three major fields: *Object-Oriented Design, Declarative Paradigm in Meta-Programming* and *Machine Learning*. The goal of this chapter is to present the foundations of these domains. Research works of design flaw detection is presented in the last section. The rest of this work mainly refers to the concepts which are presented in this chapter.

This chapter is structured into two parts. The first part describes related theoretical backgrounds of this detection work. In the first, we concentrate on answering the question of *what* is good object-oriented design and on the criteria for obtaining and assessing it. After discussing the object-oriented design, we detail the foundations of Meta-Programming on how it supports the proposed detection performs efficiently. Declarative Programming is discussed the issue of its appropriate properties to implement meta language in meta level. Machine Learning also supports to generate pattern logic rules of complex flaws in the meta program. In the second part, state of the art researches in design flaw detection is described. All of information – methodologies, techniques and tools, are discussed in the latter part.

Part I : Theoretical Background

2.1 Object-Oriented Paradigm and Design Flaws

This foundation concept of object-oriented paradigm is introduced in this section. We explain first ideas of object-oriented programming and its novel mechanisms to deal with the complex software development. Good object-oriented design principles are discussed later. Although there are many good mechanisms and principles deal with complex software system in object-oriented paradigm, design flaws still emerge in software. We discuss the characteristics of design flaws and their effect on software quality.

2.1.1 Object-Oriented Programming

The essential factor that influenced the evolution of programming paradigms is the necessity to deal with the increasing complexity of software programs (Coad and Yourdon, 1991).

Object-oriented programming provides us with a set of proper mechanisms for the management of this complexity, namely: data abstraction, encapsulation, modularity, inheritance, and polymorphism. In this heading we will discuss these mechanisms. Booch defined object-oriented programming as follows (Booch, 2004):

Definition 2.1 *Object-oriented programming is an implementation method in which programs are organized in object collections that cooperate among themselves, each object representing an instance of a class; each class is part of a class hierarchy and all classes are related through their inheritance relationships.*

Analyzing the definition above, we find three important elements of object orientation:

- objects and not algorithms are the fundamental logical blocks;
- each object is an instance of a class;
- classes are linked among themselves through inheritance relationships.

In the context of the previous definition, we can now introduce Sommervilles definition (Sommerville, 1995) of object-oriented design:

Definition 2.2 *Object-oriented design is a design strategy where system designers think in terms of things instead of operations or functions. The executing system is made up of interacting objects that maintain their own local state and provide operations on that state information.*

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

2.1.1.1 Data Abstraction

One of the fundamental ways used by all people in order to understand and comprehend a complex issue is by using abstractions. A good abstraction is one that underlines all the aspects that are relevance to the perspective from which the object is being analyzed while at the same time suppressing or diminishing all the other characteristics of the object. In the context of object-oriented programming Booch offers us the following definition of an abstraction (Booch, 2004):

Definition 2.3 *An abstraction expresses all the essential characteristics that make an object different from some other object; abstractions offer a precise definition of the objects conceptual borders from an outsiders point of view.*

In conclusion in the process of creating an abstraction our attention is focused solely towards the exterior aspect of the object and as such on the objects behavior while at the same time ignoring the implementation of this very behavior. In other words abstractions help us distinguish clearly between what an object does and how the object does it.

An objects behavior is characterized through a sum of services or resources the object offers to some other fellow objects. Such a behavior in which an object (server) offers services for other objects (clients) is described in the so called client-server model. The entirety of the services offered by a server object constitutes the objects contract or responsibility towards other objects. Responsibilities are fulfilled by means of certain operations (also called: methods or member functions). Each objects operation is characterized by a unique signature composed from: a name, a list of formal parameters and a return type. The sum of an objects operations and their corresponding rules for calling constitute the objects protocol.

2.1.1.2 Encapsulation

Just as abstractions are used for identifying an objects protocol, encapsulation deals with selecting an implementation and treating it as a secret of that particular abstraction. The encapsulation process will be viewed therefore as the action of hiding the implementation from most client objects. In a more concise way we can define encapsulation as follows:

Definition 2.4 *Encapsulation is the process of splitting the elements that form the structure and behavior of an abstraction into individual compartments; encapsulation is used for separating the contractual interface from its implementation.*

The definition above makes clear that an object has two distinct parts: the objects interface (protocol) and the implementation of this interface. Abstraction is the process that defines the objects interface and encapsulation defines the objects representation (structure) together with the interface implementation. The concealment of an objects structure and method implementation make up the so-called information hiding notion. Encapsulation provides a set of advantages:

- By separating the object interface from the objects representation one can modify the representation without affecting the various clients in any way because these depend on the server objects interface and not its implementation.
- Encapsulation allows one to modify programs efficiently, with a limited and localized effort.

2.1.1.3 Modularity

The purpose of splitting a program into modules is to reduce the costs associated with redesign and verification issues by allowing one do this for every module independently (Britton et al., 1981). The classes and objects obtained after the abstraction and encapsulation processes must be grouped and then deposited in a physical form called a module. Modules can be viewed as physical containers in which we declare the classes and objects that result after the logic level design. These modules form therefore the programs physical architecture. A program can be split into a number of modules that can be compiled separately but that are connected (coupled) among themselves. The languages that support the module concept also make the distinction between the modules interface and its implementation. We can say that encapsulation and modularization go hand in hand.

2.1.1.4 Inheritance

Abstractions are a good thing but in most non-trivial applications we will find a greater number of abstractions that we can simultaneously comprehend. Encapsulation manages complexity by hiding the interior of its abstractions. Modularity helps by offering the means of grouping abstractions that are logically linked among themselves. All these, although useful, are not enough. A group of abstractions often forms a hierarchy and by identifying this hierarchy we can greatly simplify the problem understanding. The most important class hierarchies in the object paradigm are: the class hierarchy ("is a"relationship) and the object hierarchy ("part of"relationship). Class Hierarchy Inheritance defines a relation among classes in which a class shares its structure and behavior with one or more other classes (we talk about simple and multiple inheritance). The existence of an inheritance relationship is the difference between object-oriented programming and object based programming.

From a semantic point of view inheritance indicates an "is a" relationship. For example a bear "is a"mammal so there is an inheritance relationship between the bear and mammal classes. Even as a programming issue this remains the best test for detecting the inheritance relationship between two classes A and B: A inherits B only if we can say that "A is a kind of B". If A "is not a" B, then A should not inherit B. In conclusion inheritance implies a hierarchy of the generalization/specialization type in which the class that derives specializes the more generalized the structure and behavior of the class from which it was derived. Object Hierarchy Aggregation is a relationship between two objects in which one of the objects is part of the other object. From a semantic point of view, aggregation indicates a "part of" relationship. For example there is such a relation between a wheel and a car because we can say that "a wheel is part of a car".

2.1.1.5 Interfaces and Polymorphism

Interfaces

As we mentioned earlier, the sum of all function signatures for the functions that can be called by clients of that particular object class form the classs interface. Interfaces are fundamental in object-oriented systems. The objects are known inside the system only through their interfaces. There is no other way of finding out something about the object or asking it to do something except by using its interface. An objects interface says nothing about its implementation. Therefore different objects can implement the same interface in different ways. This means that two objects with identical interfaces can have wildly different implementations.

Binding

When a certain operation is requested, the way in which the operation will be fulfilled depends not only on the operation itself but also on the object that will receive and execute the request by calling one of its member functions. This happens because there can be more than one objects that can respond to a particular request. In other words, the requested operation specifies the desired service and the concrete object represents the individual implementation of that service. The association between a requested operation and the object that will provide the concrete implementation of the operation through one of its member functions is called binding. Depending on the moment when this binding takes place, we differentiate between two types of binding:

- Static binding (early binding) - the association is created at compilation time. This binding is based on the types system known to the compiler through the various class declarations and the corresponding fixed (and therefore rigid) association of a class for each object.
- Dynamic binding (late binding) - the association is not created when the program is compiled but rather it takes place when the program is running (at run-time).

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Polymorphism

In this manner, when binding dynamically, the request for an operation does not lead to the automatic correspondence between that operation and a certain implementation, the correspondence takes place only when the program is running. The main advantage of dynamic binding is the possibility of substituting objects that have identical interfaces at run-time. The option of using some object in another objects instead when both objects share the same interface is called polymorphism. Polymorphism is therefore one of the fundamental concepts of object-oriented programming.

2.1.2 The Good Object-Oriented Design

In the previous section we introduce the key mechanism involved in object-oriented design. But, as in chess, knowing the chess pieces and the moves does not make you a good chess player. In this section we will therefore discuss what a good design is and what makes the difference between a good and a bad design. The quality of a design has an essential impact on the whole development process. Considering the life cycle of a software system, the design phase is responsible for no more than 10% - 15% of the total effort; yet, up to 80% of the costs are invested in the correction of erroneous design decisions that arise during this phase (Bell et al., 1987). So, what is good design? Coad defines good design as follows (Coad and Yourdon, 1991) :

Definition 2.5 *A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime.*

Thus, a good design is reflected by the minimization of costs, i.e. the costs of creating the design, transforming it into a proper implementation, testing, debugging and maintaining the system. Coad also emphasizes the fact that from the formerly mentioned cost categories, the most substantial one is related to maintenance, therefore he concludes: the most important characteristic of a good design is that it leads to an easily maintained implementation. More recently, Pfleeger also discusses the characteristics of a good software design in following terms (Pfleeger, 2001) :

Definition 2.6 *High-quality designs should have characteristics that lead to quality products: ease of understanding, ease of implementation, ease of testing, ease of modification, and correct translation from requirements specification. Modifiability is especially important, since changes to requirements or changes needed for fault correction sometimes result in design change.*

All these statements above guide us to the following couple of conclusions:

- It is hard to comprehend and quantify the goodness of a design by itself; therefore we have to apply the biblical principle: by their fruit you will recognize them, i.e. we can get an understanding of the quality of the design only by regarding its fruits: testing efforts, maintenance costs and the number of reusable fragments.
- We need criteria for evaluating a design not in order to build perfect software but to help us avoid badness. Therefore, good design is a matter of avoiding those characteristics that lead to bad consequences (Coad and Yourdon, 1991).

It is impossible to establish an objective and general set rules that would lead automatically to high-quality design if they would be applied. But on the other hand heuristic knowledge reflects and preserves the experience and quality goals of the developers. They also help the beginners to evaluate and improve their design. Therefore, we are going to discuss next the most relevant characteristics to avoid poor object-oriented design and show the reflection of these characteristics in terms of *heuristics*. According to those undesirable characteristics, a good object-oriented design should have a manageable complexity, should provide a proper data abstraction and it should reduce coupling while increasing cohesion.

2.1.3 Object-Oriented Design Flaws

The software industry is confronted with a large number of software systems in use, in the size of millions of lines of code. By their inherent size, complexity and development times, they have reached the suitable shape for object-orientation paradigm. Yet, most of these systems lack all of the aforementioned quality design and principles (stated in section 2.1.2): they are instead monolithic, inflexible and hard to extend. We can identify the following causes for this situation:

- *Time Pressure.* Often systems tend to start with a clear and rigorous design, but then the developers are confronted with a lot of time constraints. This fight against the clock forces them to choose the fastest design solution and not the one that keeps the integrity of the design.
- *Changing Requirements.* Requirements change in ways that cannot be anticipated in the initial design. These changes often require essential modifications on the architectural level. In many cases those who implement the changes are not aware of the initial design and because of this the design becomes blurred.
- *Immature Object-Oriented Designers.* Most legacy systems of the first generation were written by programmers that had less understanding of the principles of object-oriented design. The ignorance of those principles has led to a lot of poorly designed code.

As a conclusion, we may state that object-oriented programming is a basis technology, that supports quality goals like maintainability and reusability but just knowing the syntax elements of an object-oriented language or the concepts involved in the object-oriented technology is far from being sufficient to produce good software. A good object-oriented design needs design rules and practices that must be known and used.

One of the famous defect on quality attributes of software is *design flaws*. The design structure of these flaws have a strong negative impact on quality attributes such as flexibility or maintainability. Thus, the identification and detection of these design problems is essential for the evaluation and improvement of software quality. Design flaws are hard to define, because sometimes we encounter situations in which a code fragment might be considered as a flaw in one case while in another case, a similar, mostly identical design fragment is justifiable and may not be considered as a design flaw. In the context of this work we define design flaws based on Marinescu's definition (Marinescu, 2004) as follows:

Definition 2.7 ***(Design flaws)*** *The structural characteristic of a design entity or design fragment that expresses a deviation from a given set of criteria typifying the high-quality of a design is called a design flaw.*

A design flaw itself is not an error or problem but a strong indication of poor design of source code structure. In the literatures, many researchers introduce flaws in different ways. Fowler and Beck (Fowler and Beck, 2000) coin the term ”Bad Smell”. They present an informal definition of twenty two of bad smells that provide a set of characteristics used as indicators for design flaws. Anti-patterns, proposed by Brown et. al. (Brown et al., 1998), are a design level literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences. During the past years to now, the issue of identifying and correcting design problems become an important concern for the object-oriented community (Demeyer et al., 2002; Fowler, 1999; Riel, 1996).

2.1.3.1 Design Flaw Taxonomy

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

There are several ways in which design flaws can be classified; they could be classified according to their abstraction level or to the good-design criteria they are deviating from. In this work, we use the classification of design flaws belonged to Mäntylä (Mäntylä, 2010), which is according to the granularity level of the design entity affected by each flaw. The taxonomy of design flaws can be classified in seven categories as follows:

Bloaters Bloaters represent something in the code that has grown so large that it cannot be effectively handled. The smells in the Bloater category are: Long Method, Large Class, Primitive Obsession, Long Parameter List, and Data Clumps. In general it is more difficult to understand or modify a single long method than

several smaller methods. The same kind of argument holds also for Long Parameter List and Large Class. Primitive Obsession does not actually represent a bloat, but is a symptom causing bloats, because it refers to situations in which the logic handling the data appears in large classes and long methods. For Data Clumps we could also argue that it should be in the Object-Orientation Abusers, because in theory a class should be created from each Data Clump. However, since Data Clumps often appear with the Long Parameter List smell we have decided to include it in this category.

Object-Orientation Abusers

The smells in the Object-Orientation Abuser category are: Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces, and Parallel Inheritance Hierarchies. This category of smells is related to cases where the solution does not fully exploit the possibilities of OO design. In Switch Statements, smell type codes are used and detected using switch statements. In OO software design the need for these type codes should, however, be handled by creating subclasses. The Parallel Inheritance Hierarchies and Refused Bequest smells lack proper inheritance design, which is one of the key elements in OO programming. The Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse. The Temporary Field smell means a case where a variable is in the class scope, when it should be in the method scope. This violates the information hiding principle.

Change Preventers

The third category of smells refers to code structures that considerably hinder the modification of the software. The smells in the Change Preventers category are: Divergent Change and Shotgun Surgery. The key is that according to (Fowler and Beck, 2000) the classes and the possible changes need to have a one-to-one relationship, e.g., one class that is modified when a database is changed, another class which is modified when new sorting algorithms are added. The smells in this category violate this principle. The Divergent Change smell means that we have a single class that is modified in many different types of changes. The Shotgun Surgery smell is the opposite. Developers need to modify many classes when performing one type of change.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Dispensables

The smells in the Dispensables category are Lazy Class, Data Class, Duplicate Code, and Speculative Generality. These smells represent something unnecessary that should be removed from the code. Classes that are not doing enough need to be removed or their responsibility needs to be increased. Data Class

and Lazy Class represent such smells. Also unused or redundant code needs to be removed, which is the case with Duplicate Code and Speculative Generality. Interestingly, Fowler and Beck (Fowler and Beck, 2000) do not present a smell for dead code. Developers find this quite surprising, since in their experience it is a quite common problem. Dead code is code that has been used in the past, but is currently never executed. Dead code hinders code comprehension and makes the current program structure less obvious.

Encapsulators The Encapsulators deal with data communication mechanisms or encapsulation. The smells in the Encapsulators category are Message Chains and Middle Man. The smells in this category are somewhat opposite, meaning that decreasing one smell will cause the other to increase. Removing the Message Chains smell does not always cause the Middle Man smell and vice versa, since the best solution is often to restructure the class hierarchy by moving methods or adding subclasses. Naturally, one could argue that the Message Chains smell belongs in the Couplers group and that the Middle Man smell belongs in the Object-Orientation Abusers. Developers believe that in order to get a better understanding of these smells they should be introduced together.

Couplers There are two coupling related smells, which are Feature Envy and Inappropriate Intimacy. The Feature Envy smell means a case where one method is too interested in other classes, and the Inappropriate Intimacy smell means that two classes are coupled tightly to each other. Both of these smells represent high coupling, which is against the object-oriented design principles. Of course, here developers could make an argument that these smells should belong in the Object- Orientation Abusers group, but since they both focus strictly on coupling, Developers think it is better if they are introduced in their own group.

Others This class contains the two remaining smells Incomplete Library Class, and Comments.

A Mäntylä's taxonomy for design flaws is introduced. The purpose of this taxonomy is to prevent the problems arising from the flat list of twenty two code smells. With this classification, developers feel that it makes the flaws more understandable, recognizes the relationships between flaws and puts each smell into a larger context. Next section, another concepts used in detection of this work are discussed.

2.2 Declarative Paradigm in Meta-Programming

In this section, the declarative paradigm is described in detail. By means of a declarative programming language, PROLOG, and some examples, the basic concepts is explained of such a language and use such concepts in building the proposed detection approach. After that, the topic of Declarative Meta Programming is discussed. The reasons about the environment of Declarative Meta Programming efficiently supported in the proposed detection is discussed.

2.2.1 Declarative Paradigm

Before descriptions which show how to write programs of a declarative programming language in meta programs of Declarative Meta Programming environment are described, the declarative programming paradigm is discussed. In order to do this, there is a comparison of it with some other programming paradigms:

- **Imperative programming** The typical property of imperative languages is that programs written in them have some sort of state. A program is written by specifying a number of steps at time of execution and manipulating that state of the program. Programs in both procedural as object-oriented languages are generally written in imperative style. Examples of this paradigm are C++, C, Pascal and Java

- **Declarative programming** Declarative programming is a way of specifying *what* a program should do, rather than specifying *how* to do it. Unlike most imperative programming languages are based on the steps needed to solve a problem, declarative programming languages only indicate the essential characteristics of the problem and leave it to the computer to determine the best way to solve the problem. Examples of such languages are PROLOG and SOUL.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

For descriptions of Declarative Programming, some advantages of Declarative Programming can utilize to the proposed detection of this work. First consideration is that the programs written in it are easy to understand. Learned logic rules can also be easy to understand and give proper reasons with Explanation-based learning mechanism in the reasonable way. This issue is discussed in detail in section 2.3. Second, programs written in a declarative programming language specify what is needed to be computed instead of how it has to be computed. When flaw identification in software systems is performed, points which consider is just what flaws want to find. Therefore this paradigm is suitable for modeling the detection. In the following section the basic concepts of a specific declarative language, Logic Programming language, that used in this

dissertation is proposed. This language is used for implementing meta language of the proposed detection. The detail of implementing is described in section 2.2.2.

2.2.2 Logic Programming

Logic programming is one of the declarative programming paradigm. It consists of logic programs to identify knowledge of a specific problem. The definition of logic programming can be defined as follows:

Definition 2.8 ***(Logic Programming)*** *The Logic Programming paradigm is based on first-order predicate logic. Programs in a logic language are written by specifying the base knowledge that is available about a problem and the relationships between this knowledge is so-called facts. The part of the program that will derive new information out of these facts consists of rules. These rules are used to deduce new facts out of already existing ones.*

PROLOG (Deransart et al., 1996) is probably the most famous (Flach, 1994) of the implementation of logic languages. PROLOG is used for implementing meta language in meta-programming in this dissertation..

The simple syntax of PROLOG by means of an example is shown following. There is a consideration the following situation: some information about a set of people and know who is the parent of who. A small program that shows the `grandParent` relationship between those people is written. This program can be expressed as the following PROLOG program as follows.

```
(1)  parent(jim,bob)
(2)  parent(bob,julie)
(3)  parent(bob,eric)
(4)  grandParent(X,Y) :- parent(X,Z), parent(Z,Y)
```

The first three lines ((1)-(3)) of PROLOG examples above express some base knowledge of the example (namely that `jim` is a parent of `bob`, that `bob` is a parent of `julie`,...). This information is called **facts** . The last line (4) of the example form a **rule** that defines the `grandParent` relationship. Variables in this rule are written with a capital letter. Notice that this rule does not say how to compute the grandparent relationship. It gives a definition of it: someone (`X`) is the grandparent of someone (`Y`) if there is another person (`Z`) such that person `X` is the parent of `Y` and `Z` is the parent of `Y`. As the reader will remark, this logic program is a very

intuitive definition of the problem which should be expressed. If `grantParent` is processed to show who is a grandparent of who with PROLOG. A query is posed to the PROLOG interpreter. In this case the query would look like:

```
grandParent(X,Y).
```

The logic language then processes the query and it output:

```
X → jim, Y → julie
X → jim, Y → eric
```

Above information shows the variable *bindings* that the logic language will return. Not only a single solution is returned. Instead the query results in a set of variable bindings for every possible solution. Every such binding exists out of a variable name and a value for that variable. If we take a look at our example we see that if we take as value for `X = jim` and for `Y = julie`, then this pair would be a correct result of the `grandParent` relationship. If consideration in the example, `julie` is a grandchild of `jim`.

2.2.3 Declarative Meta Programming

Declarative Meta Programming (DMP) is defined as the use of a declarative programming language for writing meta programs. Meta programs are programs that process programs; as opposed to more plain programs that process data. Programmer and software engineer use meta programs as compilers, program editors, integrated development environments, UML editors and program verifiers. The definition can be described as follows:

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Definition 2.9 ***(Declarative Meta Programming: DMP)*** *The use of a Declarative Programming Language at Meta level to reason about and manipulate programs built in some underlying base language.*

Declarative programming languages are very suitable for writing meta programs because they allow the programmer to focus on what needs to be achieved rather than how to achieve it. This is the very definition of declarative programming languages, which refers back to Kowalski's well-known equation *"Algorithm = logic + control"* (Kowalski, 1979). His notion that the logic or "what" part of a program is easier to define when it is separated from the control or "how" part also holds for meta programs. A typical processing step in a meta program is doing a search

on the program being processed; for example finding a function that calls three specific other functions. This is made easier if one can define what one is searching for separately from how to, for example, loop over all functions and all the calls in those functions. Another typical task is transforming the processed program, for example, adding calls into certain functions. Again, this is made easier if one does not have to focus on how to exactly apply the transformations to ensure they are done in the correct order.

In this dissertation, logic programming is introduced for meta language. It means in this context that logic programs is written that reasons about programs which written in an object oriented language.

DMP has already been used for a lot of different applications in the context of creating development support tools in software engineering field, but all of them can be put in one of the following five categories:

1. Verification of source code (eg. conformance checking, coding conventions, design models, architectural description.
2. Extraction of information out of source code (eg. code metrics).
3. Transformation of source code (eg. refactoring, translation and evolution).
4. Generation of source code.
5. Aspect Oriented Programming.

2.2.4 Logic Programming Theory

This section gives a short introduction to logic programming theory. An in-depth overview of the subject is not considered but it limits a level to the concepts that are useful for explaining Explanation-Based Learning and its algorithms. A more extensive treatment of the subject is referred to (Flach, 1994) and (Lloyd, 1984). This section starts by introducing some terminology which use in the Explanation-Based Learning section. Not only the syntax of the building blocks of logic programs (Horn Clauses) is described, but also their semantic meaning of logic program is briefed including model theory and proof theory.

2.2.4.1 Terminology

In the course of this chapter and later, the reader encounter a few terms related to logic programming theory and predicate logic. The definition of a few relevant terms is given in this

subsection. It is not the intention that the reader looks at these terms now, but that he/she refers to this list whenever a term is encountered that is not all clear.

Definition 2.10 ***(Predicate)*** *A predicate consists out of a predicate symbol (a constant) and a number of arguments (the* arity *of the predicate). A predicate has a truth value true or false.*

For example the predicate `sum(1,2,3)`, the predicate symbol is sum and the arity is 3. if considering interprets the predicate as ”the third argument of the predicate is the sum of the first two arguments” then the truth value of the predicate is true.

Definition 2.11 ***(Truth value of a clause)*** *A Horn clause can be true or false. This value is dependent on the truth values of the predicates in the clause.*

Definition 2.12 ***(And, Or and Implication)*** *The $\wedge$ is the and-operator. The clause $A \wedge B$ is true if and only if A and B are both true. The $\vee$ is called the or-operator. The clause $A \vee B$ is true if either A or B are true. The clause $A \rightarrow B$ is true if A and B have both the same truth value or if A is false and B is true. The implication $A \rightarrow B$ can be rewritten as the clause $\neg A \vee B$.*

Definition 2.13 ***(Soundness)*** *A logic program is sound if everything is deduced from it is true.*

Definition 2.14 ***(Completeness)*** *A logic program is complete if it covers all positive examples.*

Definition 2.15 ***(Consistency)*** *A logic program is consistent if it does not cover any of the negative examples.*

Definition 2.16 ***(Derivation or Deduction)*** *Clause C_2 is derivable from clause C_1 ($C_1 \vdash C_2$) if clause C_1 is gotten from clause C_2 by applying rewrite operators to C_2.*

Definition 2.17 ***(Logic consequence)*** *Clause C_2 is a logic consequence of C_1 ($C_1 \models C_2$) if every model of C_1 is a model of C_2.*

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

2.2.4.2 The syntax of logic programs

In the previous subsection, a practical viewpoint to how logic languages work by studying the PROLOG language is given. Additional formal viewpoints of logic programs are described. Programs in logic programming language is defined as a collection of Horn clauses. The following grammar gives a formal definition of such a Horn clause.

clause := head [$\leftarrow$ body]
body := atom
body := [atom [$\wedge$ atom]*]
atom := predicate[(term [,term]*)]
term := variable | constant | list
variable := ?identifier
constant := identifier
identifier := ”a single word”
list := $\langle$ [term]*$\rangle$

The above conventions grammar is used to define the grammar: productions between [and] may be omitted. For example `a[b]` produces the string `a` or the string `ab`. `A*` means to repeat the production zero or more times. The instance `i*` means that it can produce the strings like the empty string or string { *i,ii,iii,* ...}. The | symbol is equivalent with an ’or’. For example `a|b`, this can produce two strings namely the string ’`a`’ or the string ’`b`’.

2.2.4.3 Properties of logic languages

The foundation of the basic syntactical concepts of a logic language is introduced in last subsection. A few properties of these languages that are important for the further discussion of this dissertation are discussed. Note that a clause or atom is **grounded** if it contains no variables (eg. `parent(mia, louise)`).

Let *Variables* be the set of all the variables in a logic program *P* and let *Literals* be the set of all the terms that occur in a clause *C* of a logic program *P*.

Definition 2.18 ***(Substitution)*** *A substitution* $C\theta$ *is a mapping Variables* $\rightarrow$ *Literals in which we change every occurrence of variable* `X` *in* `C` *into literal* `l`*. Note this as:* $\theta = \{X/l\}$.

For example: `C = parent(jim,X)`, $\theta = \{X/bob\}$, $C\theta$ `= parent(jim,bob)`. It

can say that a substitution θ is a **unifying substitution** of clauses C_1 and C_2 if $C_1\theta = C\theta$. So θ = {x/jim, y/bob} is a unifying substitution of father(X,bob) and father(jim,Y).

2.2.4.4 Model theory

The syntax of the logic language consisting of Horn clauses is discussed in the previous subsection. The semantics of logic language is described now . In order to make it easier to determine the truth value of a Horn clause, the clause is rewritten such that the implication is removed. The following general Horn clause is considered.

$$H \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_n$$

In order to know its meaning, an astonishment a truth value to this clause by assigning a value to each of the literals is performed.

The rewriting this clause by replacing the implication with a disjunction is shown how this is done. The new clause shows as following.

$$H \vee \neg L_1 \vee \neg L_2 \vee \ldots \vee \neg L_n.$$

This clause is true if the head is true or if the negation of one of the literals from the body is true. The reader can verify that this is correct by comparing the truth values of the clause with those of the implication. And now to know whether a Horn clause is true, every L_i are associated each truth value and check whether the entire clause is true or false.

Now a few concepts that should be considered in semantics are described.

Definition 2.19 ***The Herbrand Universe (U_P)*** *of a logic program P is the set of all the grounded terms in the program P.*

In case of the example from the previous subsection, U_P = {jim, bob, louise, mia}.

Definition 2.20 ***The Herbrand Base (B_P)*** *of a program P is the set of all the grounded atoms in P. The elements of the Herbrand Base are all possible combinations of predicates and constants.*

The Herbrand Base of the example is:

```
B_P = {grandparent(jim,jim), ... , parent(jim,louise),
mother(louise,louise), ...}.
```

It is easy to see that the Herbrand Base can be quite large. It can be made a mapping between the elements of the Herbrand Base and the values {true, false}. This mapping is called the *Herbrand Interpretation* (I_P) of a program *P*. Since it have to specify for every element of a very large set if it is true or false, the using the inverse relation in practice is considered.

This means that the readers only write down the elements of B_P of which the readers say that they are true (formally this is written down as I_P^{-1}(true)). For the rest of the elements of the Herbrand base it assumes that they are mapped to false. Finally, it says that:

Definition 2.21 ***(Model)*** *M is a model for a program P if M is a subset of interpretation I and that all the clauses of P are true with respect to M.*

2.3 Machine Learning

Machine Learning is the study of methods for programming computers to learn. Computers are applied to a wide range of tasks, and for most of these it is relatively easy for programmers to design and implement the necessary software. However, there are many tasks for which it is difficult or impossible to solve such as problems for which there exist no human experts or problems where human experts exist, but where they are unable to explain their expertise. Fortunately, humans can provide machines with examples of the inputs and correct outputs for these tasks, so machine learning algorithms can learn to map the inputs to the outputs.

A machine learning system is one that automatically improves with experience, adapts to an external environment, or detects and extrapolates patterns. An appropriate machine learning technology could relieve the current economically-dictated one-fits-all approach to application design. Machine learning addresses many of the same research questions as the fields of statistics, data mining, and psychology, but with differences of emphasis. In contrast, machine learning is primarily concerned with the accuracy and effectiveness of the result of computer systems. To illustrate this, we consider the different questions that might be asked about speech data. A machine learning approach focuses on building an accurate and efficient speech recognition system.

2.3.1 Analytical and Empirical Learning

Learning tasks can be classified along many different dimensions. One important dimension is the distinction between empirical and analytical learning. Empirical learning is learning that relies on some form of external experience, while analytical learning requires no external inputs.

Consider, for example, the problem of learning to play tic-tac-toe (noughts and crosses). Suppose a programmer has provided an encoding of the rules for the game in the form of a function that indicates whether proposed moves are legal or illegal and another function that indicates whether the game is won, lost, or tied. Given these two functions, it is easy to write a computer program that repeatedly plays games of tic-tac-toe against itself. Suppose that this program remembers every board position that it encounters. For every final board position (i.e., where the game is won, lost, or tied), it remembers the outcome. As it plays many games, it can mark a board position as a losing position if every move made from that position leads to a winning position for the opponent. Similarly, it can mark a board position as a winning position if there exists a move from that position that leads to a losing position for the opponent. If it plays enough games, it can eventually determine all of the winning and losing positions and play perfect tic-tac-toe. This is a form of analytical learning because no external input is needed. The program is able to improve its performance just by analyzing the problem.

In contrast, consider a program that must learn the rules for tic-tac-toe. It generates possible moves and a teacher indicates which of them are legal and which are illegal as well as which positions are won, lost, or tied. The program can remember this experience. After it has visited every possible position and tried every possible move, it will have complete knowledge of the rules of the game (although it may guess them long before that point). This is empirical learning, because the program could not infer the rules of the game analytically – it must interact with a teacher to learn them.

2.3.2 Analytical Learning

We now turn our attention to analytical learning. Because analytical learning does not involve interaction with an external source of data, analytical learning systems cannot learn knowledge with new empirical content. Instead, analytical learning focuses on improving the speed and reliability of the inferences and decisions that are performed by the computer. This is analogous in many ways to the process of skill acquisition in people.

Considering a computation involves search. Examples include searching for good se-

quences of moves in chess, searching for good routes in a city, and searching for the right steps in a cooking recipe. The task of speedup learning is to remember and analyze past searches so that future problems can be solved more quickly and with little or no search. The simplest form of speedup learning is called caching – replacing computation with memory. When the system performs a search, it stores the results of the search in memory. Later, it can retrieve information from memory rather than repeating the computation.

For example, consider a person trying to bake a cake. There are many possible combinations of ingredients and many possible processing steps (e.g., stirring, sifting, cooking at various temperatures and for various amounts of time). A cook must search this space, trying various combinations, until a good cake is made. The cook can learn from this search by storing good combinations of ingredients and processing steps (e.g., in the form of a recipe written on a card). Then, when he or she needs to bake another cake, the recipe can be retrieved and followed. Analogous methods have been applied to speed up computer game playing. Good sequences of moves can be found by searching through possible game situations. These sequences can then be stored and later retrieved to avoid repeating the search during future games. This search for good move sequences can be performed without playing any real games against opponents.

A more interesting form of speedup learning is generalized caching – also known as *Explanation-Based Learning*. Consider a cook who now wants to bake bread. Are there processing steps that were found during the search for a good cake recipe that can be re-used for a good bread recipe? The cook may have discovered that it is important to add the flour, sugar, and cocoa powder slowly when mixing it with the water, eggs, and vanilla extract. If the cook can identify an explanation for this part of the recipe, then it can be generalized. In this case, the explanation is that when adding powdered ingredients (flour, sugar, cocoa) to a liquid batter (water, eggs, and vanilla extract), adding them slowly while stirring avoids creating lumps. This explanation supports the creation of a general rule: Add powdered ingredients slowly to a liquid batter while stirring. When baking bread, the cook can retrieve this rule and apply it, but this time the powdered ingredients are flour, salt, and dry yeast, and the liquid batter is water. Note that the explanation provides the useful abstractions (powdered ingredients, liquid batter) and also the justification for the rule. Explanation-based learning is a form of analytical learning, because it relies on the availability of background knowledge that is able to explain why particular steps succeed or fail. Retrieving a rule is usually more difficult than retrieving an entire recipe. To retrieve an entire recipe, it just needs to look up the name (chocolate cake). But to retrieve a rule, it must identify the relevant situation (adding powdered ingredients to liquid batter).

Sometimes, the cost of evaluating the rule conditions is greater than the time saved by

Table 2.1: The explanation-based generalization problem

Given:

- Instance space X: Each instance describes a pair of objects.
- Hypothesis space H: Each hypothesis is a set of Horn clause rules. The head of each Horn clause is a literal containing the target concept predicate. The body of each Horn clause is a conjunction of literals based on the same predicates used to describe the instances.
- Goal Concept: A concept definition describing the concept to be learned.
- Training Example D: An example of the goal concept.
- Domain Theory B: A set of rules and facts to be used in explain how the training example is an example of the goal concept.

Determine:

- A hypothesis from H is consistent with the training examples and domain theory.

not searching. This is known as the utility problem. One solution to the utility problem is to restrict the expressive power of rule conditions so that they are guaranteed to be cheap to evaluate. Another solution is to approximate the rule conditions with different conditions that are easier to evaluate, even if this introduces some errors. This is known as knowledge compilation. Explanation-based learning mechanisms incorporates into cognitive architectures such as the *SOAR* architecture (Laird et al., 1987) and the various ACT architectures (Pavlik and Anderson, 2004).

2.3.3 Explanation-Based Theory

After concepts of machine learning and preliminary analytical learning in previous subsection are discussed, the detail of explanation-based learning is described in this section. This learning algorithm constrains the search by relying on knowledge of the task domain and of the concept under study. After analyzing a single training example in terms of this knowledge, these methods are able to produce a valid generalization of the example along with a deductive justification of the generalization in terms of the system's knowledge. The explanation-based method analyzes the training example by first constructing an explanation of how the example satisfies the definition of the concept under study. The feature of the example identified by this explanation are then used as the basis for formulating the general concept definition.The justification for this concept definition follows from the explanation constructed for the training example. The generic problem definition of Explanation-Based Learning shown in Table 2.1. It summarizes the class of generalization problems considered in this dissertation.

From Table 2.1, the learner is given a hypothesis space H from which it must select an output hypothesis, and a set of training examples $D = \{\langle x_1, f(x_1)\rangle, \ldots, \langle x_n, f(x_n)\rangle\}$ where $f(x_i)$ is the target value for the instance x_i. The desired output of the learner is a hypothesis h from H that is consistent with these training examples.

In analytical learning, the input to the learner includes the same hypothesis space H and training examples D as for inductive learning. In addition, the learner is provided an additional input: A domain theory B consisting of background knowledge that can be used to explain observed training examples. The desired output of the learner is a hypothesis h from H that is consistent with both the training examples D and the domain theory B.

To illustrate more concretely, in example each instance x_i would describe a particular characteristic of inputs, and $f(x_i)$ would be True when x_i is a characteristic for which approach to target concept, and False otherwise.

In this dissertation, we consider explanation-based learning from domain theories that are perfect, that is, domain theories that are correct and complete. A domain theory is said to be *correct* if each of its assertions is a truthful statement about the world. A domain theory is said to be *complete* with every positive example in the instance space, if the domain theory covers every positive example in the instance space. For more detail, it is complete if every instance that satisfies the target concept can be proven by the domain theory to satisfy it.

Explanation-Based Learning operates by learning a single horn clause rule by the algorithm called Prolog-EBG (Mitchell et al., 1986; DeJong and Mooney, 1986). Prolog-EBG is a sequential covering algorithm. For more detail, it learns one rule, removing the positive training examples covered by this rule, then iterating this process on the remaining positive examples until no further positive examples remain uncovered. When given a complete and correct domain theory, Prolog-EBG is guaranteed to output hypothesis (set of rules) that is itself correct and that covers the observed positive training examples. The Prolog-EBG algorithm is summarized in Table 2.2.

From learning algorithm in table 2.2, each new positive training example that is not yet covered by a learned Horn clause forms a new Horn clause by step:

1. Explaining the new positive training example. Each training example is to construct an explanation in term of the domain theory.

Table 2.2: The explanation-based learning algorithm Prolog-EBG.

Data: *TargetConcept, TrainingExamples, DomainTheory*
Result: *LearnedRules*

LearnedRules ← {};
Pos ← the positive example from *TrainingExamples*;

Prolog-EBG(*TargetConcept, TrainingExamples, DomainTheory*);
forall the *positive elements of example in Pos* **do**
  1. Explain:;
  Explanation ← an explanation (proof) in terms of the *DomainTheory* that *PositiveExample* satisfies the *TargetConcept*;
  2. Analyze:;
  SufficientConditions ← the most general set of features of *PositiveExample* sufficient to satisfy *theTargetConcept* according to the *Explanation*;
  3. Refine:;
  LearnedRules ← *LearnedRules* + *NewHornClause*,;
  where *NewHornClause* is of the form *TargerConcept* ← *SufficientConditions*;
  Return *LearnedRules*;
end

---

2. Analyzing this explanation to determine an appropriate generalization. The algorithm computes the *weakest preimage* of the target concept with respect to the explanation, using a general procedure called *regression* (Waldinger, 1977).

3. Refining the current hypothesis by adding a new Horn clause rule to cover this positive example, as well as other similar instances. A new instance is classified as negative if the current rules fail to predict that it is positive.

Thus, Explanation-Based Learning involves reformulating the domain theory to produce general rules that classify examples in a single inference step. One interesting capability of PROLOG-EBG is its ability to formulate new features that are not explicit in the description of the training examples, but that are need to describe the general rule underlying the training example. After all concepts which involved with this dissertation is described, in next section the-state-of-the-art design flaw detections are introduced .

2.4 Related Works

The problems of design flaw detection are discussed from different groups in this section. Typical approaches can be divided into two categories. On the one hand, there are approaches

use usability aspect and user experience to exploit the static structure of flaws. Some of these use Declarative Programming to describe the structures of the programs source code. On the other hand, some approaches base the detection on metrics and their automated interpretation. Both groups are important for efficient flaw detection and acceptance of automated approaches.

2.4.1 Smells, Design flaws and Anti-patterns

Firstly, Beck and Fowler coin design flaws as the term *Smell* in (Fowler and Beck, 2000) for structures in code that possibly need a refactoring. They describe a set of twenty-two smells. However, they explicitly do not give any precise criteria to identify those smells in code. Instead, Fowler and Beck refer to the developers intuition and experience (Fowler and Beck, 2000) . These twenty-two smell descriptions range from complex design problems (e.g. *Parallel Inheritance Hierarchies*) and problems on class level (e.g. *Large Class*) to tiny problems like Method Has Too Many Parameters (Van Emden and Moonen, 2002). These different levels are not stated though implicitly given. They give only some hints on the relation of design patterns (Gamma et al., 1995) and smells. They reason on the relation of feature envy and the strategy and visitor pattern (Fowler, 1999). Gamma et al. give further hints, however no comprehensive catalog is provided currently.

Marinescu gives a formal definition called *Design Flaw* (Marinescu, 2005). A design flaw is a negative property of an entity in a software system. The design flaws are explicitly related to not only implementation level, but also the level of design, i.e. package, class, method, etc. Such entities with negative properties expose a deviation from criteria characterizing non-functional high-quality designs (Marinescu, 2001). This deviation from a given set of criteria is expressed in metrics and automated interpretation of measured values. The characterization of high-quality designs is based upon a study of forty-five JAVA projects in (Lanza and Marinescu, 2010).

Tom Tourwé and Tom Mens refer to these structures also as *refactoring opportunities* (Tourwé and Mens, 2003). They report on a framework using flaws to propose adequate refactorings. Brown et al. provide a pattern language to describe flaws so called *AntiPatterns* (Brown et al., 1998). AntiPatterns are commonly occurred solutions that cause obvious negative consequences. AntiPatterns can be the result of actions taken by the different participants of a software project. AntiPatterns can occur, among others, in design, architecture and processes.

Mäntylä et al. propose a taxonomy for the initial set of smells provided by Beck and Fowler (Mäntylä et al., 2003). The twenty-two smells of Beck and Fowler are grouped into six categories: *Bloaters, Object Orientation Abusers, Change Preventers, Dispensables, Encapsulators* and *Cou-*

plers. From their survey done in an industrial project, Mäntylä et al. compute correlations between smells. Their taxonomy is applied to these correlations. They conclude that this taxonomy and the empirical study of smell relations is only of initial nature. The study shows that relations between smells exist, and are not only pure theory.

2.4.2 Analysis based on structural detection

One type of famous approach to flaw detection is the use of *Structure Analysis*. Several approaches in research make use of such *a structural detection* to find design problems by their typical structure in code.

The detection in early age is in manual approach, based on software inspection techniques on text-based descriptions (Travassos et al., 1999; Fagan, 1986, 2002). Software inspection involves carefully examining the code, the design and the documentation of software and checking them for aspects that are known to be potentially problematic based on past experience. It is generally accepted that the cost of repairing a flaw is much lower when that flaw is found early in the development cycle. Van Emden and Moonen (Van Emden and Moonen, 2002) present in their early work on code inspection and smell flaw detection. Their approach is based on a generic parser generation and term rewriting framework. Some flaws of them are detected and presented as graphs (Slinger, 2005) by implementing as plug-in for the Eclipse Integrated Development Environment (Eclipse IDE). They state that flaws are characterized by different flaw aspects. They distinguish between primitive flaw aspects and derived flaws aspects to split flaw detection into two steps. Derived flaw aspects are inferred from primitive flaw aspects. Primitive flaws and flaw aspects are collected by visitors traversing the abstract syntax tree of the analyzed source code. These facts are input to a calculator for relational algebra. This calculator is used to infer more complex design problems from the collected facts.

However, manual inspection is time-consuming process and strongly depends on developer's programming expertise. For lessening this problem, software visualization (Langelier et al., 2005) is used to support flaw detection. This strategy reduces the search space – time- and resource-consuming – which compensates for human intervention.

Bravo (Bravo, 2003) develops a framework based on the Declarative Programming. His framework is applied to use logic to reason about source code and propose opportunities to preform refactorings by Tourwe (Tourwé and Mens, 2003). The detectors are implemented as logic predicates that reason about the structures presented in analyzed source code. Some detectors make also use of metrics. All flaw instances have an attached weighting. This weighting is com-

puted from the badness of a flaw itself, weighting between flaw pairs and user given sorting. The selection of logic programming is based on the fact that this paradigm enables fast execution and easy description of complex queries on huge factbases. The work of Kniesel et al. do support this selection (Kniesel et al., 2007). GenTL (Appeltauer and Kniesel, 2008) is presented, a generic analysis and transformation language. GenTL uses snippets of the analyzed language, called *code patterns*, to select elements during analysis. The patterns may contain variables as placeholders for elements of the analyzed language. The use of code patterns and variables balances needed expressiveness, ease of use and high abstractness with the power of Logic Meta Programming. However, currently no implementation is available.

2.4.3 Analysis based on metrics

Another type of approach to flaw detection is the use of metric computations. Either complex, specialized metrics are developed or several simpler metrics are combined into a strategy to detect flaws.

Based on cohesion consideration, Simon et al. (Simon et al., 2001) report a detection algorithm to find opportunities for four different refactorings. Those are the refactorings *Move Method*, *Move Field*, *Extract Class* and *Inline Class* (Fowler, 1999). Simon et al also develop the metric *Distance Based Cohesion* to measure the similarity between two entities (methods and fields in the cited paper). This metric can be used to cluster entities. It provides opportunities for refactorings that improve code towards the principle “Put together what belongs together ”. These detected opportunities suggest to move a method or field and extract or inline a class.

Marinescu develops a whole framework to define flaw detectors from the composition of metrics (Marinescu, 2004, 2005; Lanza and Marinescu, 2010). Marinescu uses metrics and filterings, that are logically composed into so called *Detection Strategies*. To define new detection strategies for a design heuristic, appropriate metrics have to be chosen first. The next step is to select filterings for these metrics. These metric filters are logically composed with and, or and not operators. The result is a single, encapsulated detection rule for a design flaw, that can be effectively computed. Some heuristics explicitly state semantics that can be related to the selected metrics. Those allow for a “semantical filtering ”. If absolute numbers are part of the heuristic, an “absolute semantical filter ”is possible. *Classes should not contain more objects than a developer can fit in his or her short-term memory. A favorite value for this number is six* (Riel, 1996).

Marinescu states that some heuristics do not allow for absolute semantical filters and he calls these heuristics fuzzy and presents two further types of filters. A “relative semantical fil-

ter"considers the highest or lowest values of a dataset. An example for this heuristic is *Methods of high complexity should be split*. For heuristics mentioning extreme values , "statistical filter"is proposed by Marinescu. Statistical filters catch extreme, abnormal values. An example for such a heuristic is "Avoid packages with an excessively high number of classes".

In the latest publication (Lanza and Marinescu, 2010) of Marinescu and Lanza, only absolute semantical filters are used. For fuzzy heuristics, deviations from normal values are encoded into the thresholds. These are based on a study about forty-five JAVA projects, measuring complexity per line, lines of code per method and number of methods per class. Averages and standard deviations as well as lower and higher margins are computed from this collected data. Assuming normal distribution, 70% of the values will be in this interval between lower and higher margin. Very high values are assumed to be 50% higher than the higher margin. This statistic is completed by universally accepted thresholds like one, few (2-5), short memory cap (7-8) and fractions that seem natural to humans like a quarter, a half and two thirds.

There are improvements to detection strategies using history information. Ratiu et al. (Ratiu et al., 2004) improve the detection strategies of Marinescu (Marinescu, 2004), for the God Class and Data Class design flaws, by applying them to a range of versions of a project. Whenever a method is added or removed to a class, its version was taken into account. Changes within methods are not considered. Ratiu et al. compute stability and persistence of a design flaw by testing the class under inspection for the presence of design flaws for each version. Stability is the ratio of versions having the tested design flaw, while the version before also exposed the design flaw. Persistence is the ratio of versions having the design flaw in relation to all changed versions. They conclude that persistent and stable design flaws are "harmless". Such design flaws are part of the system for a long time and are almost refactored. Thus they seem not to do any harm to the development of the system, which aligns with the idea of irrelevant flaws. Improving accuracy detection of metric-based techniques in design flaws detection are supported by the *Tuning Machine* method (Mihancea and Marinescu, 2005), based on a Genetic Algorithm, which tries to find automatically the proper threshold values. However, design flaws cannot be directly measured by software metrics. Consequently metric-based techniques translate a flaw into measurable code properties which are thought to be related to the flaw. This technique is insufficient to precisely identify design flaws (Moha et al., 2006).

2.4.4 Usability and Efficient Flaw Detection

Murphy-Hill and Black distinguish between floss refactoring and root canal refactoring (Murphy-Hill and Black, 2008a). *Floss refactoring* is characterized by frequent refactorings that

are interleaved with other changes (e.g. adding a feature). *Root canal refactoring* are infrequent, large blocks of refactoring, during which nearly no other changes are made. According to their research, the majority of refactorings is carried out as floss refactorings. Murphy-Hill and Black postulate seven habits in order to build usable, highly effective, flaw detectors (Murphy-Hill and Black, 2008b). These guidelines are backed by an empirical study, an experiment with a questionnaire. They conclude that programmers value these guidelines and that the guidelines enable programmers to understand more flaws with greater confidence. Among these guidelines are the following (Murphy-Hill and Black, 2008b):

- Context-Sensitivity. A flaw detector should first and foremost point out flaws relevant to the current programming context. Fixing flaws in a context-insensitive manner may be a premature optimization.
- Availability. Rather than forcing the programmer to frequently go through a series of steps in order to see if a tool finds any flaws, a flaw detector should make flaw information available as soon as possible, with little effort on the part of the programmer.
- Scalability. A proliferation of flaws in code should not cause the tool to overload the programmer with flaw information.
- Relationality. A flaw detection tool should be capable of showing relationships between code fragments that give rise to flaws.

Mealy et al. conduct a usability study of software refactoring tools (Mealy et al., 2007). They derive a set of usability guidelines from eleven collections of such guidelines. Further, guidelines on the required level of automation are added. Mealy et al. propose nearly full automation (called level six out of eight levels) for the phases of acquiring and analysis during refactorings. Within this automation level, the computer selects a way to do the task, executes it automatically and then informs the human. Some flaws are too fuzzy to be computed. An example is the *Speculative Generality* flaw that is found in classes that do more or are more flexible than is required by the users of a system. It is not possible to reason about this “extra flexibility”(Mealy et al., 2007). Thus they conclude that flaw detection should not be completely automated.

Mealy et al. use the derived set of guidelines to analyze four common refactoring tools. From the results of this study they infer that work on the provided level of automation in current tools is needed. Automation of flaw detection and refactoring proposal is required to improve the usability of such tools (Mealy et al., 2007).

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

2.4.5 Tools of detection

A wide range of tools exists that provide static analysis of software systems. Available tools range from style and bug checks as well as maintainability indices and metric measuring used in industrial projects to research prototypes to detect flaws..

Checkstyle [1] is a static analysis tool to validate source code conventions. The provided rules can be configured and the tool can be extended with new, custom defined rules. Additionally Checkstyle is capable of computing several metrics. All results are presented in a report. The analysis and reporting are also integrated into Eclipse. The user interface of tool show in Fig. 2.1.

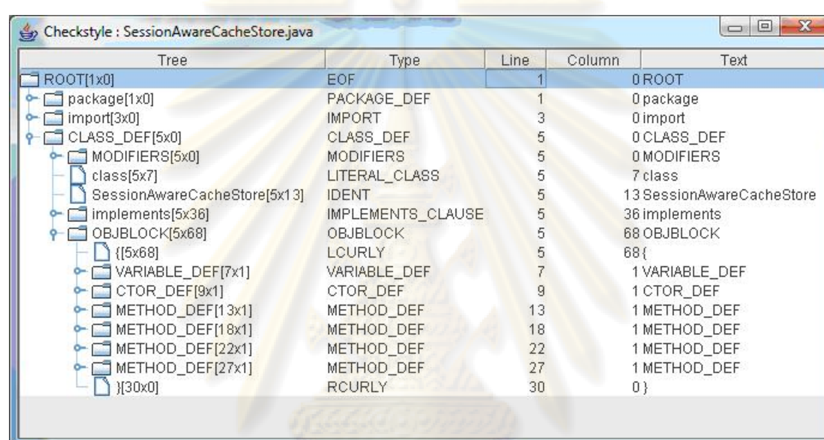


Figure 2.1: Checkstyle tool

The interpretation of violations and metric results is up to the user of Checkstyle. There is no integration with a refactoring tool, like the refactorings that are part of the Eclipse Java Development Tools.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

FindBugs [2] uses static analysis to detect bugs in Java source code. The analysis is based on patterns that are often errors. Tool does not detect any design problems nor is it integrated with refactoring tools. FindBugs program show in Fig. 2.2.

XRadar is a meta-tool. It combines nine different tools to quantify several quality attributes for design, architecture, maintenance and testing. XRadar is successfully used in an industrial project to identify problem areas and measure the refactoring progress in a legacy system (Kvam et al., 2005). The reported process of prioritized refactorings and architectural changes supports

[1] http://checkstyle.sourceforge.net/ accessed on 29.12.2009
[2] http://findbugs.sourceforge.net/ accesses on 29.12.2009

the conclusion that XRadar fits a root canal refactoring approach.

CodeNose is the prototype developed by Slinger and Moonen (Slinger, 2005). CodeNose is a plug-in for Eclipse. It detects design problems by traversing abstract syntax trees. Design problems are inferred with a relational algebra calculator using collected primitive and derived flaw aspects. CodeNose does not provide any further integration with refactorings of Eclipse Java Development Tools.

JDeodorant is the application to detect flaws and can automatically refactor selected flaw instances (Fokaefs et al., 2007). The selection is provided by the user and it starts only upon the command of the user. The tools is limited to only two flaws. The two flaws supported are Feature Envy and Type Checks (Fokaefs et al., 2007; Tsantalis et al., 2008). Each flaw is presented in its own view, listing all detected instances.

Summarizing the current state of tooling in static analysis and especially flaw detection, all presented tools lack one or several important points. Checkstyle and FindBugs are examining no design problems. XRadar is a meta-tool and provides no analysis itself. CodeNose and JDeodorant use Slice-based cohesion metrics to find design problems in each modules, not entire program. The tool of Bravo et al. (Bravo, 2003) works with Smalltalk and is not available for the currently more popular Java language. Currently nearly no tool is available to experiment with

Figure 2.2: FindBugs tool

contexts and evaluate the usage of contexts in an industrial project. Thus for the development of different contexts and their evaluation, the state of art in flaw detection has to be implemented in the Cultivate Project. Cultivate provides a stable platform to implement metrics and flaw detectors within the logic meta-programming paradigm.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER III

THE PROPOSED FLAW DETECTION METHODOLOGY

Based on the idea of " *the environment of design extraction that is able to retrieve not only the structural information, but also information related to method implementation. The reasoning of a particular design in that environment is chanced to open up*" motivated from (Wuyts, 1998) and (Mens et al., 2003), a problem of design flaw detection with learned pattern is introduced and defined in this chapter. Before the proposed flaw detection methodology is proposed, the meta architecture of detection is discussed firstly. By avoiding false positive problems by using learn-one-rule algorithm, this approach might help the developers from the false specifying design flaws and increasing precision of flaws detection.

3.1 An Overview of Detection Architecture

Although useful information can be extracted from reverse engineering modules for language such as JAVA, the information extracted is limited to what can be obtained from a structural analysis of the source code — package, classes, attributes, methods and inheritance relationships. Such information narrows the scope of design flaw patterns of detection. Also the analysis of information extracted is defined in language-dependent way. Actually, the required information for detection contains certain elements and relations, not how they are represented in a particular language. To step beyond these limitations, it was necessary to provide a design extraction module that was able to retrieve not only the structural information, but also information related to method implementation. This extraction is done according to the *Meta Programming* paradigm. This meta programming has been investigated as a technique to support software development (Mens and Kellens, 2006, 2005; Mens et al., 2003).

Declarative Meta programming (Tourwe and Mens, 2002) is defined as the use of a declarative programming language for writing meta programs. Declarative programming languages are suitable for writing meta programs because they allow the programmer to focus on *what* needs to be achieved rather than *how* to achieve it. The concept of declarative meta programming is very simple. A *meta program* must have access to a representation of its *object program* or *base program*. More precisely, the language of the object program must be represented in the language of

the meta program. The mediator interface is a mapping from the symbols of the base language to the symbols of the meta-language, but this mapping is not simply a translation from one language to another. The mapping must also enable the meta-language to make statements about structures of the base language and must also be a quotation mechanism. In this dissertation, Logic Programming is used for writing meta program because it is the most suited for meta programs that perform searching on the program they process and it answers in term of boolean value of the existence of design flaws.

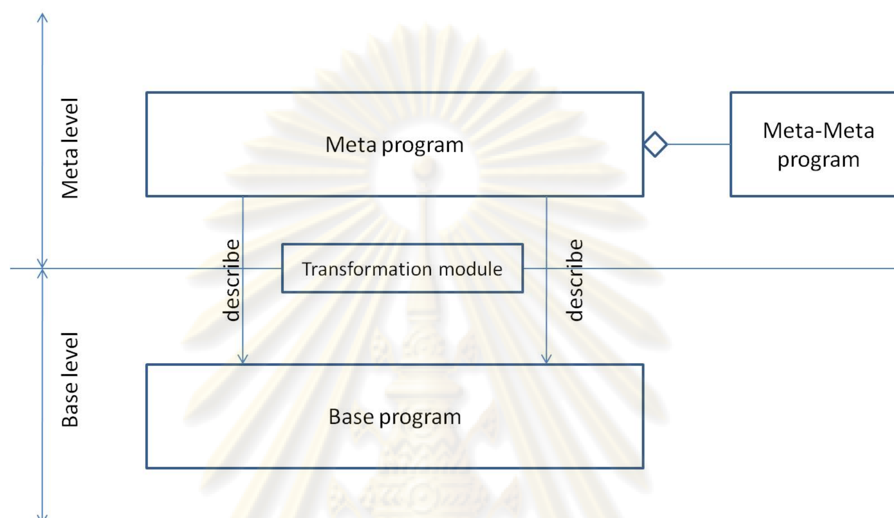


Figure 3.1: The draft design of description detection

According to the previous concept of Declarative Meta programming, the draft design of description detection is shown in Fig. 3.1. The draft design of description detection consists of two levels: the base level and the meta level. In the base level, base program is represented. The meta program in meta level describes the base program in the base level. This level constitutes a number of facts and rules of design flaw. The transformation module analyzes and transforms the syntactical and semantical information of the base program into the meta program between the base level layer and the meta level layer. The meta-meta program defines constituents to represent rules, set operators, relationships among rules and properties in meta program. The transformation uses a general parse tree representation, which enables the use of fine-grained static information. The transformation module allows both layers to be language-independent. A base program is represented indirectly by means of a set of logic propositions. These logic propositions are stored in a logic database and they link between the represented base program and its logic representation in meta program.

After the draft design is proposed, the meta architecture of description detection is created according to the detail of the previous draft design of description detection. The meta architecture

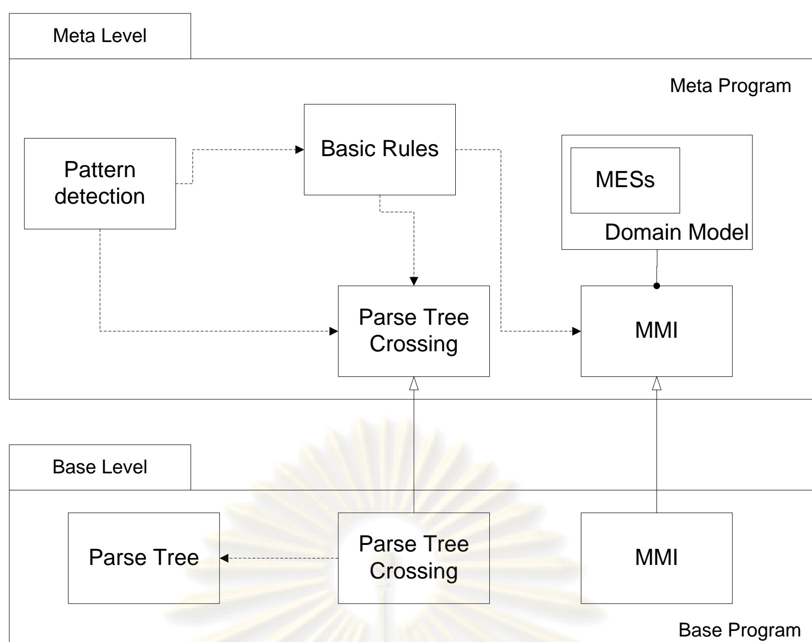


Figure 3.2: The meta architecture of description detection

of description detection shows in Fig. 3.2. The details of the meta architecture consists of the same as the draft design, two level. The *Meta Mediator Interface* module (MMI) analyzes and transforms the syntactical and semantical information of the base program into the meta program between the base level layer and the meta level layer. The MMI consists of the *Domain Model* which determines the object program structures. The domain model possess a set of sixty-five specific Meta Element Specifications. The Meta Element Specifications (MESs) representation of a base program gives a concept how the logic-based representation works. The domain model of meta program defines constituents of MESs to represent rules, set operators, relationships among rules and properties in meta program (the MESs specification is in Appendix B). The MMI uses a general parse tree representation for its analyzing process.

3.2 The Proposed Detection Methodology

Although previous works offer ways to specify and to detect design flaws, each work has a particular benefits and points on a subset of all the steps necessary to define a detection technique systematically. The processes use to specify and implement the flaw detection algorithms are not obvious — they are always driven by the service of the underlying detection framework rather than by systematic study of the flaw descriptions.

In this section, therefore, we describe the detail of detection methodology that subsume all the steps necessary to define a detection technique. Fig. 3.3 shows the eight steps of the proposed

methodology. The following item summarizes its steps:

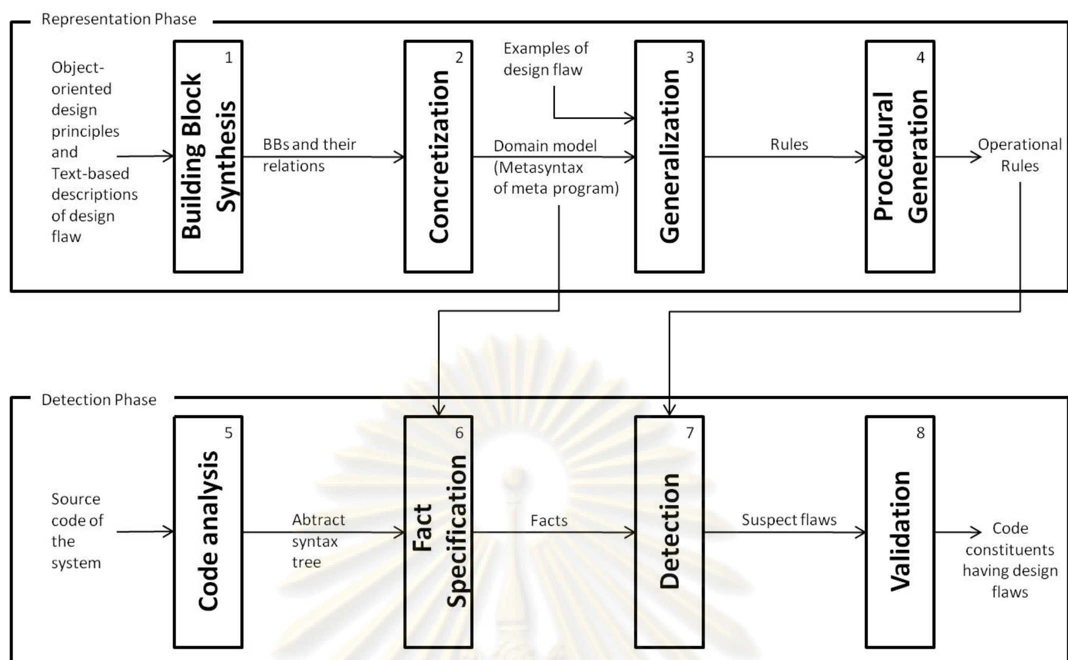


Figure 3.3: The Proposed Detection Methodology

- **Step 1. Building Block Synthesis**: Elements of core in domain model of meta program from design flaw description are identified. Such descriptions form a unified elements of design concepts of object-oriented.
- **Step 2. Concretization**: The relation in meta program, which constitutes derived elements, are combined to specify relation systematically and consistently. They are represented in meta model of meta program in the form of the formal representation.
- **Step 3. Generalization**: The learning mechanism extrapolates specific logic rules for flaws detection. The learner performs the learning mechanism by deductive proof according to Explanation-Based Learning algorithm.
- **Step 4. Procedural Generation**: All rules are optimized by reordering literals. The basic idea is that also first order queries become more efficient to execute if selective literals are placed first.
- **Step 5. Code Analysis**: Source code are parsed and formed in Abstract Syntax Tree (AST). It represents syntactical and semantical information of source code.
- **Step 6. Fact Specification**: Syntax trees are specified in information facts. They are defined according to the domain model from step 2.

- **Step 7. Detection**: The detection of design flaw is performed on system using the operational rules from step 4. It returns the list of code constituents (eg. classed, methods) suspected of having flaws.

- **Step 8. Validation**: The suspected code constituents are manually validated to verify that they actually have flaws.

The first and second steps are generic and must be based on a representation set of elements and relations of object-oriented concepts. Step 5 and 6 are also the same. Step 3 and 4 must be followed when specifying a new flaw. Step 5-8 are repeatable and must apply on each source code of system.

We believe the proposed methodology is original because the detection algorithm are not an ad hoc method, but it is generated using formalization representation that obtained from an analysis and specification of flaws. A formalization representation benefits the software engineering quality because they can specify and modify manually the detection rules by using high-level pertaining to their domain, taking into account the context of the analyzed systems. The context corresponds to all information related to the characteristics of the system including types (prototypes, system in development or maintenance, etc.), design choices (related to design heuristics and principles) and coding standards.

3.3 Detection Methodology in Details

The following subsections describe details of the eight steps of the proposed detection methodology. Each step is explained by the clear presentation which based on common patterns: input, output and description of each step.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

3.3.1 Step 1: Building Block Synthesis

Input: Concepts of object-oriented principles (Coad and Yourdon, 1991) and descriptions of design flaws (Fowler, 1999) in the literature.

Output: Building blocks type of elements and basic relations which aggregate together to represent each design flaw. They are the key components used to build the domain model for object program in the next step.

Description of the step: The first step deals mainly with identifying and defining sets of elements and their preliminary relations of such design flaws which belong to object-oriented paradigm. We call these elements and their relations *the building blocks of design flaw* (abbreviate - `bbs`) and type of elements and their relations *the building block types of design flaw* (abbreviate - `BBs`)

where each $\texttt{bbs}_{type1} \in \texttt{BBs}_{type1}$.

By the identification of building blocks, we utilize the concept of *Domain Analysis* (Neighbors, 1984). We do the process performs by analyzing related software systems in a domain to find their common and variable parts. It is concerned with relations and objects in all systems in an application area.

According to domain analysis activities, this step performs finding of needed building blocks from the description of design flaws which fellows object-oriented paradigm. We can give some definitions of building blocks as:

Definition 3.1 ***(Building Blocks Type(BBs))****: A building block (bbs) is in the set of interesting building blocks type (BBs) : ($\forall bb \in BBs$). They have relations (R_{BBs}) between them which are the Cartesian Product written as $R_{BBs} = BBs_1 \times BBs_2 \times ... \times BBs_n$. For example, when we consider n = 2, the relation of BBs_1 and BBs_2 is defined as $BBs_1 \times BBs_2 = \{(bb_1, bb_2) \in R | bb_1 \in BBs_1 \wedge bb_2 \in BBs_2\}$.*

Definition 3.2 ***(The domain of building blocks type)****: The domain of building blocks type (BBs) in this dissertation is considered as components which exist in design flaws (DF) in object-oriented paradigm ($BBs \subseteq$ design flaws $\subset$ object-oriented paradigm).*

Definition 3.3 *Let σ be a building block (bbs) of design flaw. A building block of σ' is a subbuilding block of σ iff:*

1. *The composition of σ' is a subset of the composition of σ.*
2. *The set of interaction in σ' is a subset of the interactions in σ.*

Definition 3.4 *A function L exists on the decomposition process of the building block, $L : BBs \rightarrow BBs$ where $\sigma, \sigma' \in BBs$, such that $L(\sigma) = \sigma'$ for every σ and σ'. $L(\sigma) \neq \sigma$ iff σ is a non-terminal building block, $L(\sigma) = \sigma$ iff σ is a terminal building block.*

For each input description of a flaw concept, we extract all building block types including mainly basic relations of them. These elements and their relations refer to specific integrated concepts of object-oriented design and implementation which used to describe design flaws. This

domain analysis process performs in the iterative way. In each iteration, we compare them with already-found building block type, and add them to the meta space for avoiding duplicated building block type and relations. Thus, we obtain a compilation of the building block type and relations that expresses a concise and unified natural entities of design flaw.

For more details, we explain some examples of flaws analysis to discover `BBs`. We choose to analyze flaws in their two varying properties — *Data Class* (pattern-based flaw) and *Long Parameter List* (quantitative-based flaw) from code smells (Fowler, 1999). We summarize the text description of two flaws in Table 3.1.

Table 3.1: Text descriptions of *Data Class* and *Long Parameter List* flaws

Data Class : A class has the Data Class flaw if the class has data fields and the only operations are the getting and setting operations. The existence of Data Class indicates low quality of data abstraction. It should avoid classes that passively store data. Classes should contain data and methods to operate on that data.
Long Parameter List: The more parameters a method has, the more complex it is. Limit the number of parameters you need in a given method, or use an object to combine the parameters.

Analysis of the Long Parameter List: In object-oriented paradigm (Coad and Yourdon, 1991), most of the data which a method needs can be directly obtained from the objects themselves if they are visible to the method, or they can be derived by making a request on another parameter. Therefore, parameter lists should keep necessarily short in object-oriented programs. When considering in long parameter list flaws, these flaw make programs hard to read and difficult to use and they change a lot when developers need more (or other) data. Making a change to a parameter list means changing every reference to the method involved. So, avoiding or eliminated this flaws is necessary. In the description of the Long Parameter List, we identify the `bbs` and relations of their structure and behavior with:

- `bbs`: *parameter, method* and *object*.
- relation among `bbs`: *own*.

The `bbs` which involve with this flaw consist of three entities: parameter, method and object. Such three `bbs` are no synonym in each other and type of them are `PARAMETER, METHOD` and `OBJECT`. As the same analysis, a relation of Long Parameter List is own and its `BBs` is `OWN`. We can define the set of `BBS` and relation:

$\texttt{BBs}_{\textit{Long Parameter List}} =$
$\{\{\texttt{PARAMETER,METHOD,OBJECT}\},\{\texttt{OWN}\}\}$

Analysis of Data Class: In the data abstraction, all the data and the methods that are rational to the objects of a class that is been designed need to be a part of the class (Coad and Yourdon, 1991; Isner, 1982). All unnecessary details should not be considered. The Data Class flaw disobeys of this rule. A Data Class is loosely defined as a data holder without behavior. Any corrections of this flaw consists of adding behavior to the data class (Fowler, 1999). In the description of the Data Class of Table 3.1, we identify the `bbs` and their relation. We obtain the following information for the Data Class:

- `bbs`: *class, data, operation, getting operation* and *setting operation*
- relation among `bbs`: *store, contain* and *operate*

The `bbs` which involve with Data Class flaw consist of three entities: *class, data, operation, getting operation* and *setting operation*. Such three `bbs` are synonym – operation, getting operation and setting operation – in each other and common type of them is *method* in object oriented paradigm. As the same in relation, *store* and *contain* are the synonym in such type – `OWN`. When analysis is deeply processed in object-oriented point, this *operation* relation of class is performed by method and attribute in such class. `ACCESSFIELD` `BBs` is proposed to describe this relation type. We can define the set of `BBs` and their relations of Data Class :

$$\texttt{BBs}_{\textit{Data Class}} = \{\{\texttt{CLASS,ATTRIBUTE,METHOD}\},\{\texttt{ACCESSFIELD,OWN}\}\}$$

Related BBs: All design flaws are analyzed (we analyze twenty flaws from (Fowler, 1999) in this work). For the strategy of proposed detection, we consider the detection domain in a set of eight `BBs` of object program to define meta program. The set of all `BBs` are defined by a direct product :

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

ObjectProgramOfBBs

$p : \mathbb{P} PACKAGE$

$c : \mathbb{P} CLASS$

$o : \mathbb{P} OBJECT$

$a : \mathbb{P} ATTRIBUTE$

$m : \mathbb{P} METHOD$

$s : \mathbb{P} STATEMENT$

$R_{BBs} : RelationOfBuildingBlockType$

$R_{BBs} = (p \times c \times o \times a \times m \times s)$

where the given relation types (R_{BBs}) which describes the set of all BBs relations and its values, is a set of :

`[OWN,ISA,EXTEND,IMPLEMENT,INVOCATION,ACCESSFIELD,ASSIGN,EXEC]`

3.3.2 Step 2: Concretization

Input : Related `BBs` for specificizing details.
Output : Semantic domain model follow to object-oriented semantic. It is the proposed domain model of problems used to describe design flaws in object program.
Description: In this step, the semantic base of subset of object-oriented paradigm is considered. We consider sematic of derived `BBs` in previous step for creating the *domain model* of object program structures. We formalize the specification of domain model which used to describe design flaws of object program. The formal notation like Z (Spivey, 1992) is used to express the semantic of object program structure. We choose this notation because of its maturity as a formal specification notation and its mathematical concepts that are well-known and understood (set theory and predicate calculus). Furthermore, the schema constructs of formal notation can be directly linked to the concepts of object-oriented, especially classes and its structure, providing a clear link between object-oriented constructs and their formally expressed interpretations.

3.3.2.1 A concretization of `BBs`

Abstract Model: The first step of formalizing the specification, we formalize the description of `BBs` and its abstract syntax. Firstly, a class is defined as a descriptor of a set of objects with specific properties (according to each `BBs` of {`CLASS, ATTRIBUTE, METHOD`} and its

relation of {`OWN, ISA`} which is analyzed in step 3.3.1). in terms of structure, behavior and relationships.

Therefore a considered class in which a name, attributes and methods are stated. Attributes have names and types. Methods have names, return types and parameters. Each parameter of an operation has a name and a given type.

[ClassName, Name, Type, MethodStatements]

Modifier ::= PUBLIC | PROTECTED | PRIVATE

Char ::= CLASS | INTERFACE | ABSTRACT

ClassDecl

$modifier = Modifier$

$char = Char$

$extend : ClassName$

$attribute : \mathbb{F} Name$

$method : \mathbb{F} Name$

$attrType : Name \nrightarrow Type$

$attrModifier = Modifier$

$methBlock : Name \nrightarrow MethodBlock$

$attribute = \mathrm{dom}\, attrtype$

$method = \mathrm{dom}\, methBlock$

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

$$
\begin{array}{l}
\textit{MethodBlock} \\
\hline
parameter : \mathbb{F} Name \\
return : \mathbb{F} Name \\
owner = ClassDecl \\
modifier = Modifier \\
paraType = Name \pfun Type \\
paraValue = Name \pfun ParaValue \\
returnType = Name \pfun Type \\
returnValue = Name \pfun ReturnValue \\
statement = \mathbb{F} MethodStatements \\
\hline
parameter = \operatorname{dom} paraType \\
parameter = \operatorname{dom} paraValue \\
return = \operatorname{dom} returnType \\
return = \operatorname{dom} returnValue
\end{array}
$$

Class names should be unique in the enclosing name space. Thus, the set of classes is defined as a partial function from `ClassName` to `ClassDecl`.

$$
\begin{array}{l}
\textit{Class} \\
\hline
classes : ClassName \pfun ClassDecl
\end{array}
$$

At any point in time, classes in the system contain a set of uniquely named classes.

$$
\begin{array}{l}
\textit{ClassModel} \\
\hline
classes : \mathbb{F}\, Class \\
\hline
\forall c_1, c_2 : classes \mid c_1 \neq c_2 \bullet \\
\qquad c_1.ClassName \neq c_2.ClassName
\end{array}
$$

The constraint of the schema states that each class must have a unique name.

Concrete Model: In order to give meaning to classes, value must be assigned. In object-oriented concepts, a class is viewed as defining a set of possible object instances. This is the

ศูนย์วิทยทรัพยากร จุฬาลงกรณ์มหาวิทยาลัย

system model that we adapt for our formalization.

Like detail of class, the given type `ObjectName`, `Name` and `Type` describe the set of all object identities and its values.

[ObjectName, Name, Type, ObjectValue]

ObjectDecl

$owner : ClassName$

$\ldots$

$attrvalue : Name \pfun ObjectValue$

$\ldots$

[*Declarations on an object*]

$attributes = \mathrm{dom}\ attrvalue$

$\ldots$

[*Predicates on an object*]

Object names instantiated from such a class should be unique in the enclosing name space. Thus, the set of object is defined as a partial function from `ObjectName` to `ObjectDecl`.

Object

$classes : ObjectName \pfun ObjectDecl$

At any point in time, the meaning of object-oriented concepts is a finite set of unique object instances.

ObjectModel

$objects : \mathbb{F}\ Object$

$\forall o_1, o_2 : objects \mid o_1 \neq o_2 \bullet$
$\quad o_1.ObjectName \neq c_2.ObjectName$

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

3.3.2.2 Meaning functions of R_{BBs} relations

The relation of BBs is described by the meaning of relation as mapping from one BBs to another BBs. For example, the meaning of classes as a mapping from classes to its object instances. It is assumed that there is a relationship between attribute and their values.

$$| \; value_of : ObjectValue \nrightarrow Name_{attribute}$$

The following function describes the meaning of a class. It maps a class to a set of possible combinations of object instances (Objects), whose attribute values conform to that permitted by the class. By conforming, it is meant that the values of each object's attributes conform to those permitted by the attribute of the owing class.

$$\mathcal{M}_{CtoO} : Class \nrightarrow \mathbb{P}\, Object$$

$$\forall c : Class \bullet$$
$$\mathcal{M}_{CtoO}(c) \subseteq \{o : Object \mid$$
$$o.owner = c.name \land$$
$$o.attributes =$$
$$\{a : c.attributes;\ v : AttrLink \mid$$
$$value_of(v) = a\}\}$$

The structure relations of BBs are relations among BBs. For example, the meaning of a class invocates another class as the structure of invocation relation among BBs.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

AccessorMethod

$ClassModel$

$m_{selector} : MethodSelector$

$a_{selector} : InstanceVariable$

$\forall\, c_1 : classes \wedge$

$c_1.method = m_{selector} \wedge$

$c_1.attribute = a_{selector} \bullet$

$\quad returnValue \circ methBlock(c_1.method) =$

$\quad\quad c_1.attribute \wedge$

$\quad accessField(c_1.attribute) \subseteq$

$\quad methBlock(c_1.method).statement$

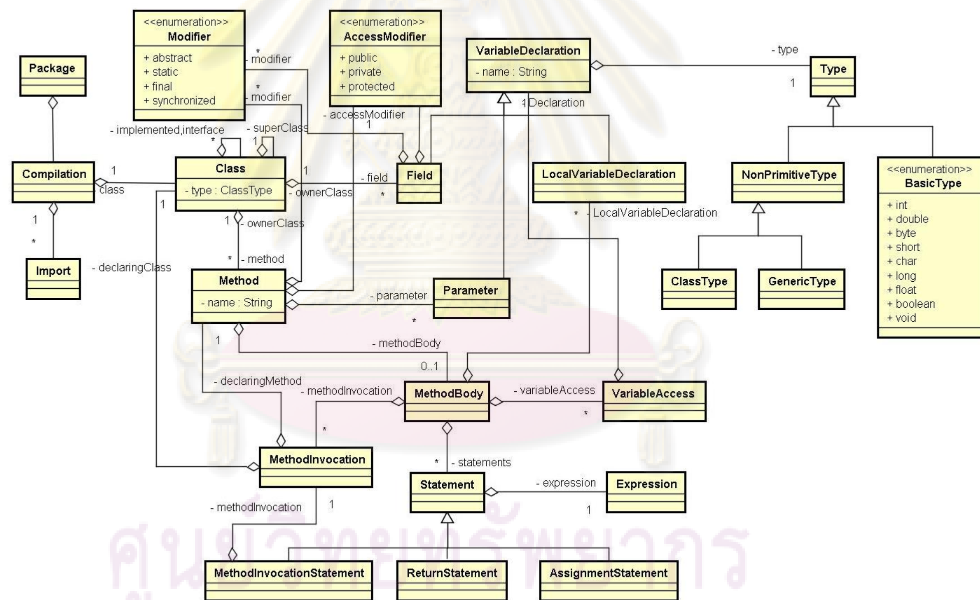


Figure 3.4: UML Model for the notation of the proposed domain model

The notation required for the rest of the BBs and R_{BBs} of domain model is graphically illustrated in the UML class diagram in Fig.3.4. The domain model represents the BBs and R_{BBs} which are necessary in order to identify design flaws for given source code programs.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

3.3.3 Step 3: Generalization

Input : Source code examples of a particular design flaw and design principles and heuristics in form of logic program which follow to domain model.
Output : Logic rules derived from learning mechanism for detecting a particular design flaw
Description: The main processes of this step involve with the learning mechanism to extrapolate specific logic rules for flaws detection. The deductive learning algorithm of Explanation-Based Learning is used for learning mechanism in this step.

The generalization processes in this step for extrapolating design flaw rules shows in Fig. 3.5. We consider explanation-based learning from domain theories that are perfect, that is, domain theories that are *correct* and *complete*. To show how to learn inferenced rules and detect design flaws by such rules, the proposed methodology performs the following step-by-step. The generalization processes are based on PROLOG-EBG learning algorithm. A known design flaws, Data Class, is chosen to concretely illustrate here for better understanding of all processes of Generalization (the detailed rest of all design flaws is in Appendix A).

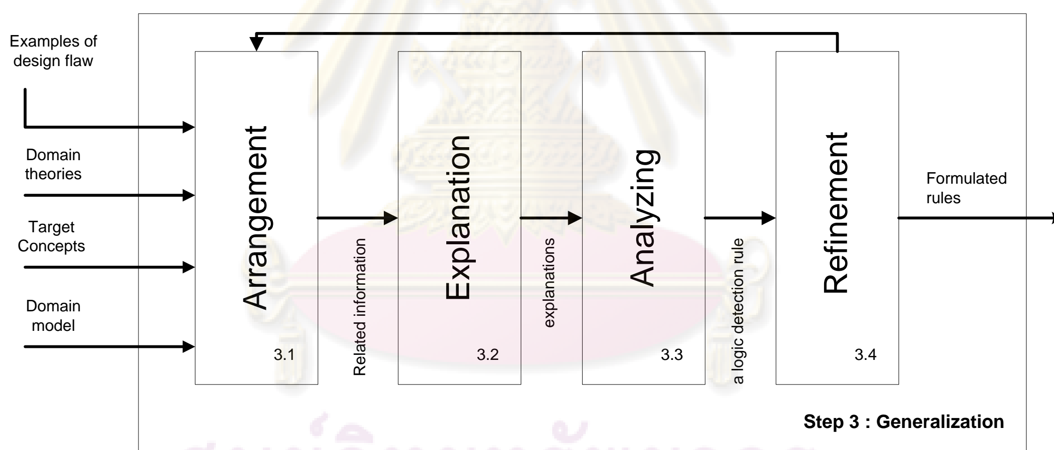


Figure 3.5: Generalization Process of Step 3: Generalization

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

3.3.3.1 Process 3.1: Arrangement

Input:

1. Domain theories which are derived from design principles and heuristics (Coad and Yourdon, 1991; Riel, 1996).
2. Examples of design flaw.
3. Target concepts of design flaw that are used to be learned.

4. Semantic domain model from Step 3.3.2 .

All of information inputs are in the form of predicate calculus and Horn clause (declarative form).
Output: Related information of domain theories and a target concept which are sufficient to generate a detection rule from a specific training example in each learning cycle for generating a logic detection rule.
Description of Process 3.1: The first step deals mainly with the arrangement of domain theories and a target concept to be learned for a training example in each learning cycle for generating a logic detection rule (Process 3.1-3.4). We begin with a training example of `FilterMap` class which is a Data Class flaw – shown in JAVA source code — in Fig. 3.6. In this source code example, we provide the single training object, `FilterMap` class which is taken from Tomcat's source code (`org.apache.catalina.deploy.FilterMap` class), which is a known positive example of a Data Class.

```
public class FilterMap implements Serializable {
   private String filterName = null;
   public String getFilterName() {
       return (this.filterName);
   }
   public void setFilterName(String filterName)
       this.filterName = filterName;
   }
   private String servletName = null;
   public String getServletName(){
       return (this.servletName);
   }
   public void setServletName(String servletName){
       this.servletName = servletname;
   }
}
```

Figure 3.6: Class `FilterMap`

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Then, domain theories related to a training example of `FilterMap` Data Class flaw are defined. We define such information according to principles and heuristics of object-oriented paradigm (Coad and Yourdon, 1991). For our purposes in flaw detection, domain theories are any set of prior beliefs about the object-oriented design and implementation principles and an inference mechanism is any procedure that suggests new beliefs by combining existing beliefs. The elements and their relations of such domain theory are defined that based on syntax and semantic of domain model. We consider learning concept descriptors for a simplified Data Class. A suitable target concept and domain theories (formed in Horn clause) for this example of Data Class are given in Table 3.2.

3.3.3.2 Process 3.2: Explanation

Input: Domain theories and a target concept which are related to a training example in each learning cycle for generating a detection logic rule.

Output: A explanation h in the hypothesis space which h ($h \in H$) is consistent with domain theory B and $B \nvdash \neg h$.

Description of Process 3.2: Given the information in the Process 3.1, this process is to determine a generalization of the training example which is a sufficient concept definition for the target concept. This process provides its *justification*: constructing an explanation (a proof tree) in terms of the domain theory that proves how the training example satisfies the target concept definition. Then a set of sufficient conditions under which the explanation structure holds is determined. This determination is accomplished by regressing the target concept through the explanation structure. To see more concretely how the Explanation-Based Learning approach works, consider learning the concept of Data Class in Fig. 3.7.

Table 3.2: Domain theories and a target concept of the `FilterMap` Data Class

Target concepts and domain theory:
R1: $\forall x$ class(x) $\wedge$ $\neg$not−dataclass(x) $\Rightarrow$ dataclass(x)
R2: $\forall x$ $\forall y$ not−dataclass(y) $\wedge$ $\neg$ is(x,y) $\Rightarrow$ $\neg$not−dataclass(x)
R3: $\forall x$ $\forall y$ hasMethod(x,y) $\wedge$ method−operation(y) $\Rightarrow$ not−dataclass(x)
R4: $\forall x$ $\neg$mutator−method(x) $\wedge$ $\neg$accessor−method(x) $\Rightarrow$ method−operation(x)
R5: $\forall x$ $\forall y$ accessor−method(y) $\wedge$ $\neg$is(x,y) $\Rightarrow$ $\neg$accessor−method(x)) .
R6: $\forall x$ $\forall y$ mutator−method(y) $\wedge$ $\neg$is(x,y) $\Rightarrow$ $\neg$mutator−method(x))
R7: $\forall x$ $\forall y$ $\forall z$ has−attribute(z,y) $\wedge$ has−method(z,x) $\wedge$ method−returntype(x,*[VOID,NULL]*) $\wedge$ method−parameter(x,[{y,_}]) $\Rightarrow$ mutator−method(x)
R8: $\forall x$ $\forall y$ $\forall z$ has−attribute(z,y) $\wedge$ has−method(z,x) $\wedge$ method−returntype(x,[y,_]) $\wedge$ method−parameter(x,[{NULL,NULL}]) $\Rightarrow$ accessor−method(x)

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

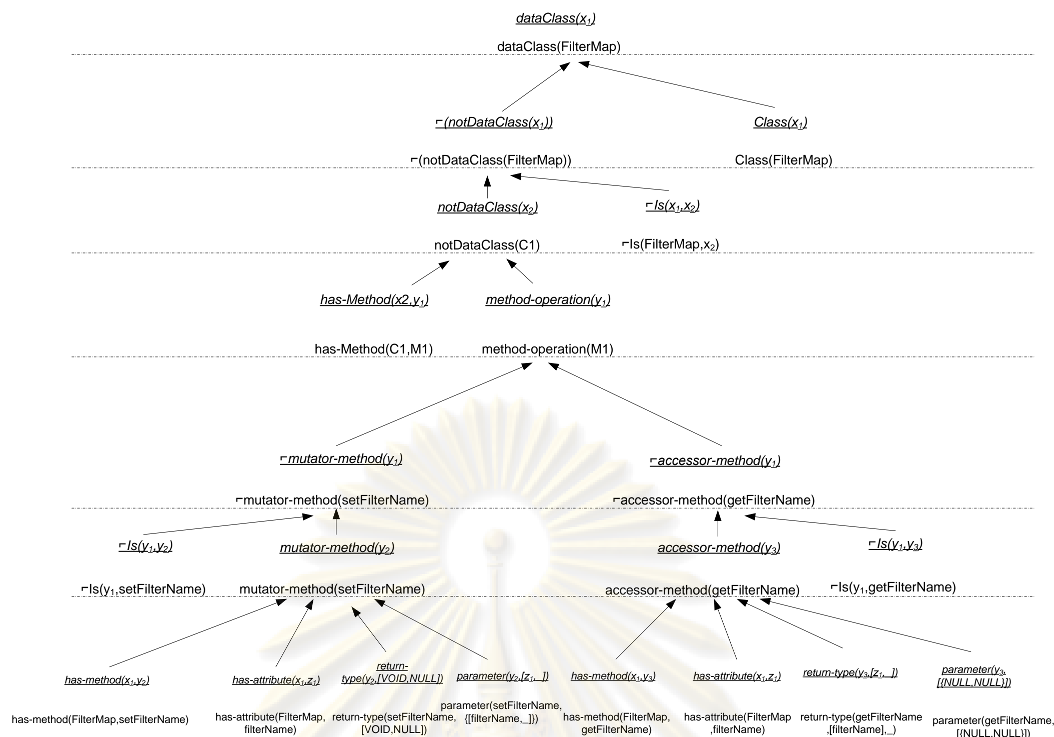


Figure 3.7: A explanation of `FilterMap` class (a Data Class flaw)

Given `FilterMap` class as a positive example, EBL system attempts to construct an explanation for why `FilterMap` class is indeed a Data Class flaw. From Fig. 3.7, arrows denote the contribution of each domain theory rule to the explanation. They point from a rule's antecedents to its consequences. For example from a rule in table 3.2 which shows in Fig. 3.7, rule R8 allows the consequence property `accessMethod` to be concluded from the antecedents `hasMethod, hasAttribute, returnType` and `parameter`. Each pair of literals (nonitalic font) and generalized literals (italic font) between each step (dashed lines) in the explanation show expressions across rules that must match for the explanation to hold. These are enforced by unifying the connected expression.

3.3.3.3 Process 3.3: Analyzing

Input: An explanation of a training example.
Output: A logic detection rule from an explanation.
Description of Process 3.3 : The explanation constructed by the previous process is generalized with a rule that is the most general relevant to the target concept. EBL computes the most general rule that can be justified by the explanation, by computing the *weakest preimage* of the explanation. The weakest preimage of the target concept is computed by a general procedure called *regression* (Waldinger, 1977). The regression procedure operates on a domain theory represented

Table 3.3: An example for regressing a set of literals given by *Frontier* through *methodOperation Rule*

REGRESS(*Frontier, Rule, Literal,*θ_{hi}) where
Frontier = Class(x_1), ¬ *Is*(x_1,x_2), *hasMethod*(x_2,y_1), *methodOperation*(y_1)
Rule = methodOperation(z) ← ¬ *mutatorMethod (z)* ∧ ¬ *accessorMethod(z)*
Literal = methodOpearation(y_1)
σ_{hi} = *[z/SetFilterName]*
head ← *methodOperation(z)*
body ← ¬ *mutatorMethod (z)* ∧ ¬ *accessorMethod(z)*
σ_{hl} ← [z/y_1] *,where* σ_{li} = [y_1/SetFilterName]
Return *Class*(x_1), ¬ *Is*(x_1,x_2), *hasMethod*(x_2,y_1),¬ *mutatorMethod* (y_1), ¬ *accessorMethod*(y_1)

Table 3.4: The final rule of `FilterMap` Data Class

dataCLass(x_1) ← Class(x_1) ∧ ¬ Is(x_1,x_2) ∧ hasMethod(x_2,y_1) ∧ ¬ Is(y_1,y_2) ∧ hasMethod(x_1,y_2) ∧ hasAttribute(x_1,z_1) ∧ returnType(y_2,[VOID,NULL]) ∧ parameter(y_2,[z_1,_]) ∧ ¬ Is(y_1,y_3) ∧ hasMethod(x_1,y_3) ∧ hasAttribute(x_1,z_1) ∧ returnType(y_3,[z_1,_]) ∧ parameter(y_3,[{NULL,NULL}])

by an arbitrary set of Horn clause. It works iteratively backward through the explanation, first computing the weakest preimage of the target concept with respect to the final proof step in the explanation, then computing the weakest of the resulting expressions with respect to the preceding step, and so on. The procedure terminates when it has iterated over all steps in the explanation. The illustrated example of `accessMethod` rule is shown in Table 3.3. We use the negation-as-failure approach to detect Data Class flaw. The final rule for the current example is illustrated in Table 3.4.

3.3.3.4 Process 3.4: Refinement

Input: A logic rule from the proof tree.

Output: New formulated rules.

Description of Process 3.4: In this step, logic rules is refined by generalizing rules in accordance with a logic rule from the proof tree. Rules are pruned some literals for making generalization. At

each learning cycle for generating a logic detection rule (Process 3.1-3.4), the sequential covering algorithm of EBL learning algorithm picks a new positive example that is not yet covered by the current Horn clause rules, explains the new example, and formulates a new rule according to the learning cycle. When we provide more examples learning, the rule of Data Class flaw is refined to be more general that the attribute of mutator method and accessor method may not be the same attribute.

3.3.4 Step 4: Procedure Generation

Input : Logic rules from EBL leaning mechanism
Output : Operational logic rules for detecting design flaws
Description: Derived explanation trees from the EBL learning in previous step gives many rules in each design flaw. The query transformation is introduced in this step that optimizes first order queries by reordering literals. The basic idea is that also first order queries become more efficient to execute if selective literals are placed first.

We start by listing a number of requirements for a reordering transformation for first order queries in the context of:

- ***R1 (Correctness)*** The reordering transformation should be correct, i.e. the transformed query should succeed (fail) for the same examples as the original query succeeds (fails).
- ***R2 (Disagreement)*** The reordering transformation should minimize conflicts among literals when conflicts exist. The literal that most clearly reflects to such design flaw will be chosen in the initial order in query literals.

If all predicates are defined by sets of facts, then the order of the literals does not influence the result of the query (cf. the switching lemma (Lloyd, 1993)) and the correctness requirement (R1) is met. With disagreement requirement (R2), literals are inspected manually to classify level of conflicts. For example, two rules of Data Class are derived from learning mechanism – (1): Class with accessor and mutator method, (2): Class contains public fields. The rule (1) is chosen in first order in query because it reflects to flaw more than rule(2) can.

3.3.5 Step 5: Code Analysis

Input : Object-oriented source code which is used detect design flaws.
Output : Syntactical and semantical information of source code.
Description: Source code are parsed and formed in Abstract Syntax Tree (AST). It represents

ศูนย์วิทยทรัพยากร จุฬาลงกรณ์มหาวิทยาลัย

syntactical and semantical information of source code. The representation of source code as a tree of nodes representing constants or variables (leave node) and operators or statements (inner nodes). For each AST node type, there is a separate MESs (Meta Element Specifications) type. Fig. 3.8 (middle part) shows an example of parsing source code in AST.

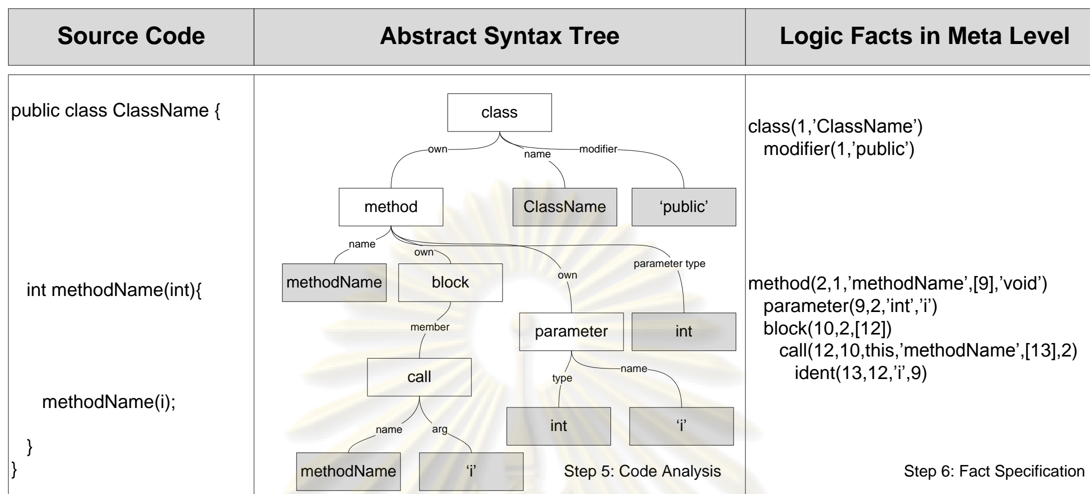


Figure 3.8: Step 5: Code Analysis and Step 6: Fact Specification

3.3.6 Step 6: Fact Specification

Input : Abstract Syntax Tree that described syntactical and semantical information of source code
Output : Logic facts which belong to domain model
Description: In this step, Abstract Syntax Tree which nodes representing constants or variables (leave nodes) and operators or statements (inner nodes) is transformed to logic facts. This transformation accords to domain model that is specified in Step 2. The argument in predicate calculus is represented by leave node and predicate part is represented by inner nodes of AST respectively. Fig. 3.8 (right part) shows an example of parsing AST into logic facts. The number in each MESs is used to represent relation among MESs.

3.3.7 Step 7: Detection

Input : Logic facts and optimized logic rules
Output : Results of the detection in each flaw
Description: The detection of design flaw happens in this step. The detection performs in declarative programming by using a backward chaining search as performed by PROLOG. Prolog-EBG approach halts once it finds the first valid proof.

3.3.8 Step 8: Validation

Input : Results of the detection

Output : Accuracy rate of the detection of the proposed methodlogy

Description: The results of the proposed detection methodology by analyzing the suspicious classes in the context of the complete model of the system and its environment. The validation is inherently a manual task. Therefore, we apply the detection of a few design flaws in different behavioral types.

We use the validation measure by using the measures of precision and recall, where precision assesses the number of true identified flaws, while recall assesses the number of true smells missed by the detection. The computation of precision and recall is performed using independent results obtained manually because only humans can assess whether a suspicious class is indeed a flaw or false positive depends on the specifications and the context and characteristics of the system.

In the next chapter, the experiment to validate the proposed detection methodology is conducted. A few case study is used to validate the proposed detection methodology. The discussion is also described in a few ways.

CHAPTER IV

EVALUATION AND DISCUSSION

In the previous chapter we describe a proposed detection methodology for detecting design flaws. We also introduce an example by showing that one of the major flaws, Data Class, for the potential detection process of the meta level of declarative meta programming, Although the specification of the proposed methodology is specified from high-level text description of each design flaw, detection rules are low level detection enough to capture the design aspects that are relevant for evaluating and improving the quality of object-oriented design. We introduce the detection methodology which consists of eight main steps as a mechanism for detecting design flaws systematically. At first, we define the assembly of building block types to describe design flaws in object-oriented scheme. The next step is to define proper relations of such building block types for creating the *domain model*. This model is domain-specific model used to describe the detection domain of the proposed detection methodology. For this purpose we define a logic rule suite of detection methodology that informally addresses a set of well-known design problems by *proof learning* of Explanation-Based Learning. The rules from learning mechanism help us to bridge the gap between pattern-based and quantitative-based design flaws concerning object-oriented design.

In this chapter we evaluate the practical applicability of the entire approach. For this purpose we design a real-world experiment, implement an adequate prototype and apply the previously described methods and techniques.

The chapter begins with the validation process with a presentation of the entire case study setup (Section 4.1). In this context, we validate our approach over three case studies with different characteristics. `CommonCLI v1.0`, `JUNIT v1.3.6` and `GANTTPROJECT v1.10.2` are the applications used for reasoning about code. The experiments consist of detecting the design flaw on the cases, analyzing the most interesting results and checking whether the proposed detection are successful or not (Section 4.2). These results also led to discussions proposed methodology on advantages and disadvantages of the proposed detection methodology (Section 4.3). The treat of validity is also discussed in this section.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

4.1 The validation of the proposed methodology

The prototype model is implemented for the proposed design flaws detection. We use `Eclipse v3.6 HERIOS`, `Prolog Development Tools v0.2.3` and `SWI-prolog v5.8.3` for implementing this prototype. To perform experiments, we select three different applications on which we detect design flaws using the prototype.

CommonCLI: `CommonCLI v1.0` is the Apache Commons CLI library provides an API for parsing command line options passed to programs. It contains 18 classes with 4132 lines of code.

JUNIT: `JUNIT v1.3.6` , the open source system, is a testing framework for JAVA which performs automated testing. It contains 111 classes with over 5000 lines of code. JUNIT is chosen as a control system on our experiment that, as it is respected in the field of software development and it has a large user base, it is likely to be well-designed. The role of the control system is to primarily check for the presence of false-positives.

GANTTPROJECT: `GANTTPROJECT v1.10.2` is a project management tool used to plan projects with Gantt charts. It contains 21,267 lines of code, 188 classes and 41 interfaces.

We seek in the following to obtain a precision which may close to 100% because we aim that the proposed methodology is a new approach which can detect flaws and correct them in automated manner. All candidates can be corrected automatically without formal technical review with low false positive rates. It means that enormous time and cost are saved. Moreover, we can ignore the uncertain experiences of experts that perform reviewing process.

4.1.1 Validation Process

First, we build models of the analyzed software systems. These models – a set of logic rules – of each flaw are obtained by the learning algorithm. Then, we apply the generated detection algorithms on the fact model of the software system and obtain all suspicious code that potentially have flaws. The detail of suspicious components and their positions are return in a document file. We validate the results of detection approach by analyzing the derived suspicious components in the context of the complete view of such software system and its environment. The validation is inherently a manual task. Therefore, we choose to apply the detection of the twenty design flaws on three case studies.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

We recast the validation in the domain of classification context and use the measures of three criterions – *Precision, Recall and Specificity* (Olson and Delen, 2008).

We use the validation measure which are the precision rate to assess the number of true identified flaws, and use the recall rate to assess the number of true flaws missed by the detection. Two such measures are shown in the equations (4.1) and (4.2).

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{4.1}$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{4.2}$$

Additionally, the proposed detection methodology aims to derive the high precision. Therefore specificity rate is also proposed to use for validation in the methodology. This rate is shown in equation (4.3).

$$Specificity = \frac{true\ positive}{true\ nagative + false\ positive} \tag{4.3}$$

The computation of three criterions is performed using independent results obtained manually because only the review process can assess whether a suspicious component is *indeed* a flaw or a false positive. It depends on the specifications and the context and characteristics of such software system. We show the result according to software validated by the prototype model. Next section, results from three experiments are shown.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

4.2 Case Studies

The manual analysis of three software case studies is performed by two independent graduate SE students with a known expertise in object-oriented design and coding and design flaws. Each time a doubt on a candidate component arises, two students consult books as references in deciding by consensus whether or not this class is actually a flaw.

Three case studies are set up in different types of software system. The detail of each case study can be described as the following.

- **Case I : `CommonCLI`**. `CommonCLI` is an small application which contains 18 classes. The detection of all design flaws is shown in this case study.
- **Case II : `JUNIT`** : `JUNIT` is a medium size of software which contains 111 classes. The detection of all design flaws is considered with this case study. We want to know the difference of results detection between the medium case study and the a small cast study (Case I). With this case study, the evaluation of true negative detection is also performed.
- **Case III : `GANTTPROJECT`** : The study of the design flaws detection between the proposed approach and another approach in `GANTTPROJECT` is proposed in this case study.

4.2.1 Case I : `CommonCLI`

In this case study, all flaws are examined by the proposed detection approach. The evaluation of precision detection of `CommonCLI` is considered.

We report results of design flaws detection of `CommonCLI` in three aspects. First, we report the number of flaws which are detected by the proposed detection methodology. Second, the numbers of detected flaws which are actual flaws. The last is a precision rate of each design flaw detection. The results are reported in groups which follow the Mäntylä's taxonomy (Mäntylä et al., 2003) as shown in Table 4.1, 4.2, 4.3, 4.4 and 4.5.

One Large Class, one Primitive Obsession and one Data Clump design flaws are detected as the results shown in the Table 4.1. The result shown in 100% precision with these three flaws detections. There are no Large Class and Long Parameter flaw detected in the Bloaters group.

One Switch Statements, one Temporary Field and one Refused Bequest design flaw are detected as the results shown in Table 4.2. The result is 100% precision with these three flaws detections. There are no Alternative Classes with Different Interfaces flaw detected in The Object-Orientation Abusers group.

One Divergent Change design flaw is detected as the results shown in Table 4.3. The result is 100% precision with this flaw detection. There are no Shotgun Surgery and Parallel Inheritance Hierarchies flaw detected in The Change Preventers group.

Five Lazy Class flaws and two Dead Code are detected as the results shown in Table 4.4. The result is 100% precision with these flaw detections. There are no Data Class and Duplicate Code flaw detected in The Dispensables group.

Table 4.1: The result of *Bloaters* flaw detection in `CommonCLI v1.0`

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Long Method	0	0	N/A
Large Class	1	1	100.00
Primitive Obsession	1	1	100.00
Long Parameter List	0	0	N/A
Data Clump	1	1	100.00

Table 4.2: The result of *The Object-Orientation Abusers* flaw detection in `CommonCLI v1.0`

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Switch Statements	1	1	100.00
Temporary Field	1	1	100.00
Refused Bequest	1	1	100.00
Alternative Classes with Different Interfaces	00	00	N/A

One Message Chains flaw is detected as the results shown in Table 4.5. The result is 100% precision with this flaw detection. There are no Feature Envy, Inappropriate Intimacy and Middle flaw detected in The Couplers group.

Table 4.3: The result of *The Change Preventers* flaw detection in `CommonCLI v1.0`

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Divergent Change	1	1	100.00
Shotgun Surgery	0	0	N/A
Parallel Inheritance Hierarchies	0	0	N/A

4.2.2 Case II : `JUNIT`

In this case study, all flaws detection study of the proposed detection approach with `JUNIT` is proposed. the evaluation of precision detection of `JUNIT` is considered.

We report results of design flaws detection of `JUNIT` in three aspects as the same as result reports in Case I. The results show in Table 4.6, 4.7, 4.8, 4.9 and 4.10.

Three Long Method, ten Large Class, eight Primitive Obsession, five Long Parameter List

Table 4.4: The result of *The Dispensables* flaw detection in `CommonCLI v1.0`

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Lazy Class	5	5	100.00
Data Class	0	0	N/A
Duplicate Code	0	0	N/A
Dead Code	2	2	100.00

Table 4.5: The result of *The Couplers* detection in `CommonCLI v1.0`

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Feature Envy	0	0	N/A
Inappropriate Intimacy	0	0	N/A
Message Chains	1	1	100.00
Middle Man	0	0	N/A

and two Data Clump design flaw are detected as the results shown in Table 4.6. The result is 100% precision with Long Method and Primitive Obsession, 83.33% precision with Large Class and estimated 60% precision with Long Parameter List and Data Clump flaw.

One Switch Statements, fourteen Refused Bequest and forty-five Alternative Classes with Different Interfaces flaw design are detected as the results shown in Table 4.7. The result is 100% precision with these three flaws detections. There are no Temporary Field flaw detected in The Object-Orientation Abusers group.

Twenty-seven Divergent Change and twenty-two Shotgun Surgery design flaw are detected as the results shown in Table 4.8. The result is 100% precision with both flaw detection. There are no Parallel Inheritance Hierarchies flaw detected in The Change Preventers group.

Seven Lazy Class , forty-five Duplicated Code and forty-five Dead Code are detected as the results shown in Table 4.9. The result is 100% precision with Duplicated Code and Dead Code detections and 87.50% precision with Lazy Class detection.

Thirty-eight Feature Envy, one Message Chains and one Middle Man flaw are detected as the results shown in Table 4.9. The result is 100% precision with Message Chains and Middle Man flaw detection and 77.50% precision with Feature Envy flaw detection. There is no Inappropriate Intimacy flaw detected in The Couplers group.

Table 4.6: The result of *Bloaters* flaw detection in JUNIT v1.3.6

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Long Method	3	3	100.00
Large Class	10	12	83.33
Primitive Obsession	8	8	100.00
Long Parameter List	5	8	62.50
Data Clump	2	3	66.66

Table 4.7: The result of *The Object-Orientation Abusers* flaw detection in JUNIT v1.3.6

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Switch Statements	1	1	100.00
Temporary Field	0	0	N/A
Refused Bequest	14	14	100.00
Alternative Classes with Different Interfaces	45	45	100.00

Because the high precision is desired for the proposed approach, the true negative rate (also called Specificity) has to be validated in this case study. The true negative rate shows the ability of classification algorithm can classify the examples which are not the target of such classification. The evaluation of true negative and false positive rate are performed as shown in Table 4.11. The specificity rate is proposed to detect sixty-three true negatives in JUNIT. The result shows 100% of precision rate to detect true negative flaws. While compared with three metrics-based approaches, the proposed approach presents the highest precision rate – 100% of the proposed approach, 98.41% with the metric approach III and 96.82% with the metric approach I and II.

4.2.3 Case III : GANTTPROJECT

The evaluation of precision rate and recall rate of the detection are considered in this case study. The source code of GANTTPROJECTV1.10.2 application is chosen. Large Class, Lazy Class, Long Method, Long Parameter List and Refused Parent Bequest design flaws are explored to indicate precision and recall rate with this application.

Table 4.12 reports detection results of the proposed approach in GANTTPROJECT. It reports numbers of true positive flaws, numbers of suspicious flaws, precisions and recalls. We compare our detection methodology with the novel literature (Moha and Guéhéneuc, 2007). The precision of our results range from 78.37% to 100% and recalls range from 44.44% to 95.12%.

Table 4.8: The result of *The Change Preventers* detection in `JUNIT v1.3.6`

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Divergent Change	27	27	100.00
Shotgun Surgery	22	22	100.00
Parallel Inheritance Hierarchies	0	0	N/A

Table 4.9: The result of *The Dispensables* flaw detection in `JUNIT v1.3.6`

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Lazy Class	7	8	87.50
Data Class	0	0	N/A
Duplicate Code	45	45	100.00
Dead Code	45	45	100.00

Fig. 4.1 and Fig. 4.2 show graphical representations of precision rate and recall rate (Compare with Metric-Based Approach) of design flaws in `GANTTPROJECTV1.10.2` respectively.

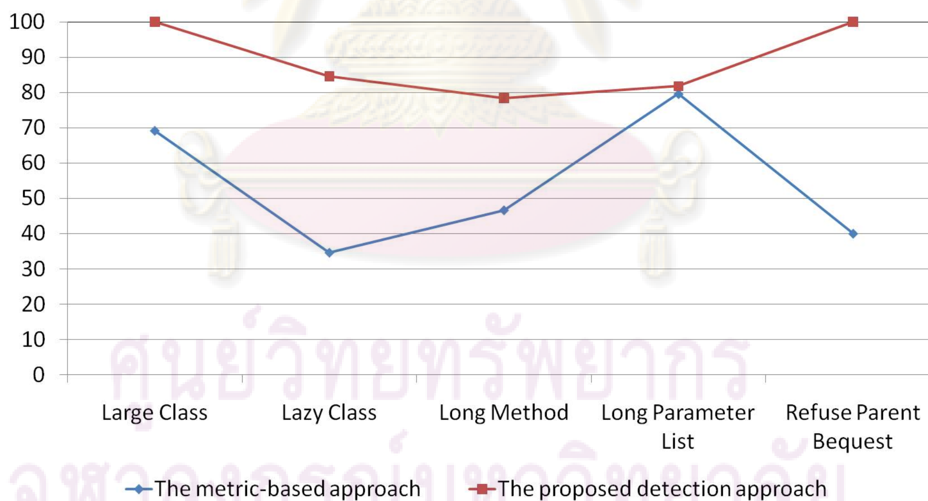


Figure 4.1: The precision rate of design flaws detection in `GANTTPROJECT V1.10.2` (Compare with Metric-Based Approach).

4.3 Result discussion

In this section, the discussion of three case studies is detailed. The interesting issues from the results of the proposed detection are also indicated.

Table 4.10: The result of *The Couplers* flaw detection in JUNIT v1.3.6

Design flaw	Number flaws of detection	Number flaws of true detection	Precision of detection (%)
Feature Envy	38	49	77.50
Inappropriate Intimacy	0	0	N/A
Message Chains	1	1	100.00
Middle Man	10	10	100.00

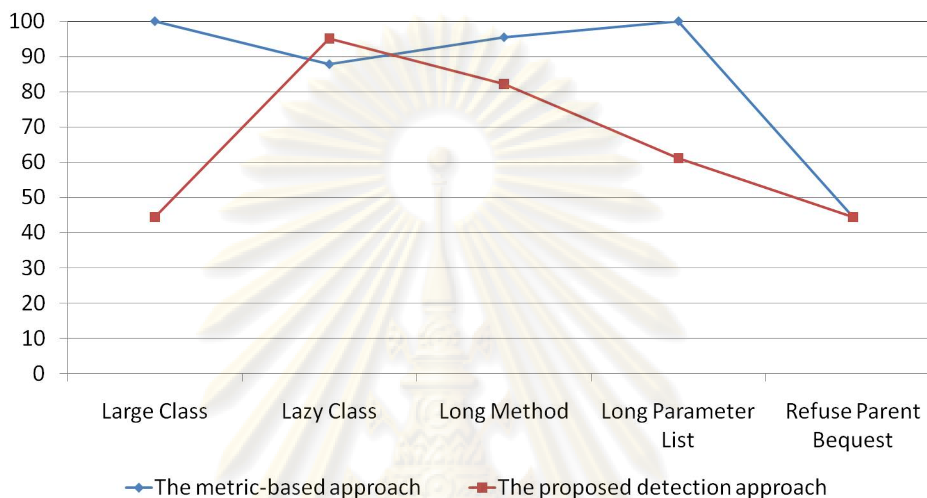


Figure 4.2: The recall rate of design flaws detection in GANTTPROJECT V1.10.2 (Compare with Metric-Based Approach).

4.3.1 Result discussion of Case I

The summary of precision rate of CommonCLI is shown in Fig. 4.3. The overall result reaches the excellent detection level with CommonCLI source code. All groups of flaw (Flaws of Bloaters, Object-oriented Abusers,Change Preventers, The Dispensable and The Couplers) can be detected. Although, for example, it has only one Switch Statement flaw that has parameter numberOfArgs in class Option, the proposed approach can detect it correctly. As well as class HelpFormatter which contains Primitive Obsession flaw is detected with the proposed approach.

4.3.2 Result discussion of Case II

The summary of precision rate of JUNIT is shown in Fig. 4.4. The overall result still indicates a good detection level in JUNIT source code except flaws of Bloaters group and the Feature Envy flaw. The precision of the proposed methodology ranges between 87.5% and 100% (not including flaws of Bloaters group and the Feature Envy flaw). At this point, we find that

Table 4.11: Specificity and its false positive rate of Data Class detection with other detection techniques in `JUNIT v1.3.6`

Properties	A metric approach I (WOC, NOPA and NOAM)	A metric approach II (WOC and NOPA)	A metric approach III (NOPA and NOAM)	The proposed methodology
Number of known true negatives	63	63	63	63
Numbers of detected as true negatives	61	61	62	63
Specificity rate(%)	96.82	96.82	98.41	100.00
False positive rate(%)	3.17	3.17	1.58	0.00

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

69

Table 4.12: Precision and recall of design flaws in GANTTPROJECT V1.10.2 (Compare with Metric-Based Approach)

Design Flaw	Numbers of known true positives	Number of detected flaw	Precision	Recall
Large Class				
Metric approach	9(4.79%)	13(6.91%)	69.23	100.00
Our Approach		4(2.12)%	100	44.44
Lazy Class				
Metric approach	41(21.81%)	104(55.32%)	34.61	87.80
Our Approach		39(20.74)%	84.61	95.12
Long Method				
Metric approach	45(23.94%)	22(11.70)%	46.66	95.45
Our Approach		37(19.68)%	78.37	82.22
Long Parameter List				
Metric approach	54(28.72%)	43(22.87)%	79.63	100.00
Our Approach		33(17.55)%	81.81	61.11
Refused parent Bequest				
Metric approach	18(9.57)%	20(10.64)%	40.00	44.45
Our Approach		8(4.25)%	100	44.44

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

the quantitative-based design flaws affect the precision rate of the proposed methodology which is suffered from *The subjective refection of design flaws*, especially Long Parameter List flaw. The decision of the existence of such flaws involves with mental of emotion of developers who perform review process. The logic rules cannot *cut through* these flaws to classify them. And, of course, this is a drawback of the proposed detection methodology.

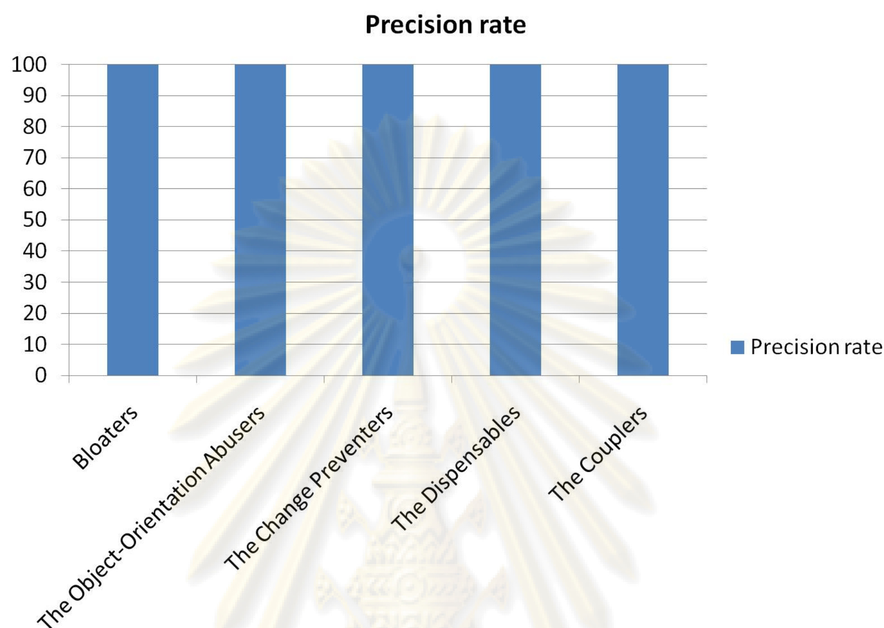


Figure 4.3: The average precision rate of proposed detection in `CommonCLI`

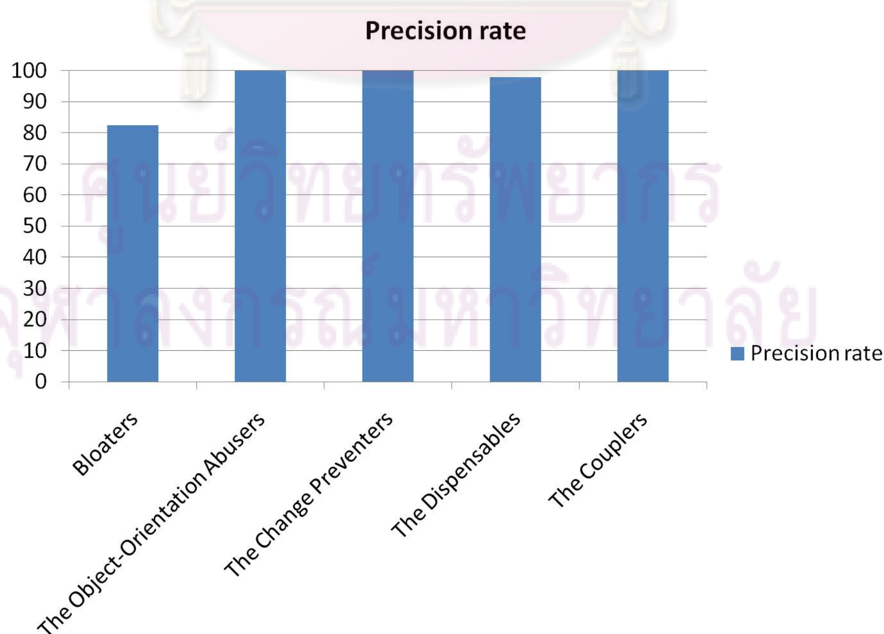


Figure 4.4: The average precision rate of proposed detection in `JUNIT`

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

The true negative (also called specificity) rate indicates the ability of the classifier in the inverse direction of precision rate. The detection approach not only can detect real flaws efficiently, but it also can classify all non-flaws as true negatives. From the results of this case, the proposed detection approach can classify all true negatives at the excellent level. This reason can confirm to support in this context that the proposed detection has quite high precision rate with the true positive and the true negative.

4.3.3 Result discussion of Case III

According to the detection results of Case III, five detected design flaws with precision and recall rate are proposed (one pattern-based design flaw and four quantitative-based design flaws). The precision rate is 100%, 84.61%, 78.37%, 81.81% and 100% with flaw detections of Large Class, Lazy Class, Long Method, Long Parameter List and Refuse Bequest respectively. The recall rate is 44.44%, 95.12%, 82.22%, 61.11% and 44.44% with flaw detections of Large Class, Lazy Class, Long Method, Long Parameter List and Refuse Bequest respectively.

For the Lazy Class, we find 39 suspicious flaws. We gain the optimum recall rates because the proposed methodology matches to the structure of this flaw perfectly. It means that the level of subjective reflection of design flaws is the least of other detected flaws. For the same pattern-based, the Refused Parent Bequest flaw is different. This flaw illustrates the inverse problem: it is very difficult for software engineers to identify all its occurrences because they must appreciate if a class uses proper public and protected methods/fields of any of its superclasses. Moreover, software engineers always consider bequests provided by library classes whereas we apply our detection on the chosen software system only, not considered libraries.

For the Large Class, we obtain optimum precision because the proposed methodology uses many feature rules to detect this flaw. It seems clear to detect this flaw with expected precision. However, automatic finding this flaw takes a lot of time because of vast of space in searching to find the answer.

4.3.4 Overall discussions of the proposed detection approach

After all result data is analyzed, two important points have to discussed:

Result data from proposed methodology: The validation shows that the proposed detection methodology can detect design flaws, especially with the domain model of design flaw lead to generate detection rules, with expected recall and good precisions. Therefore it confirms with the results that:

1. The specification of domain model allows describing two types of design flaws, quantitative-based design flaw and pattern-based design flaw.
2. The results from the prototype have, mainly, an average precision rate in detection of 80% with quantitative-based flaw detection and 100% pattern-based flaw detection.
3. When considering precision and recall rate at the same time, good average precision rate is presented with greater than 88.95%. And an expected of recall rate that also acquired with an average recall rate at 76.55%. The detection methodology reports 2/3 of known design flaws can be discovered.
4. The good specificity rate, also called true negative rate, of the proposed detection methodology is obtained. The implemented prototype from the proposed methodology can detect true negative flaws efficiently when compared with specificity rate of other approaches.

Threats to the validity: Threats of the proposed detection methodology to validity can be indicated as the following.

- **The internal validity:** The validity of the results depends on directly on the pattern of flaws specifications. The experiments on a representative set of flaws are used to lessen to this threat of the validation. The logic rule detection always suffer from *the sharpness of deduction*. It means that logic rules from proof trees have the high level of specialization. Thus obtaining the most general rules to detect in such context is a difficult task. The proposed detection deals this major problem with the tree pruning technique. Rules of the proposed detection are almost the most general rules in which *the edge* of the classification between the positive example and the negative example.
- **The external validity:** This threat in this context relates to exclusive use of open-source systems to validate the proposed methodology. The free available system is always used to perform experiments to allow their verification and replication. However, these systems may prevent the recent detection approaches to generalize to other system. The proposed approach is the technique which based-on BBs and R_{BBs} of Meta Programming. It considers only *the skeletal relationship* of BBs. Therefore the meta environment can deal with this threat properly as long as the subject of detection is still the object-oriented software system.
- **The constructive validity:** The subjective characteristics of interpreting, specifying and identifying design flaws are the threat of constructive validity, especially for quantitative-

based design flaw detection. The proposed detection lessens such threats by specifying design flaws based-on real examples in literatures and manual assessment of the results.

In the next chapter, the conclusion is described. All of contribution work are summarized. The further research is also presented.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER V

CONCLUSION

This chapter concludes the research work of this dissertation. It also presents some directions for the future work.

5.1 Conclusion of dissertation

In the last decade software quality became more and more an important criterions for managing cost and effort that spend in its evolution cycle. Existence of potential errors such as design flaw lead to more cost in maintenance phase. Design flaws recur design problems in object-oriented software systems have to be detected to avoid their possible negative consequences on software development and maintenance. Consequently, design flaws detection methodologies and techniques remain an active research field and several approaches in the recent literature have been proposed to detect design flaws.

The contribution of this dissertation is to present an efficient design flaw detection approach by ignorance limitations of specific thresholds in each environment of detection and promoting the automatic detection for reducing time and cost consuming in the detection process. Toward moving to that step, the novel methodology of design flaw detection is proposed by using Declarative Meta Programming and Explanation-Based Learning technique. Two techniques are applied to investigate design flaw detection of object-oriented software design. In the proposed approach, Declarative Meta-Programming is used to represent specific object-oriented elements and their relations in form of logic rules for describing design flaws. Explanation-Based Learning is used for extrapolating pattern by deductive learning for some characteristics of design flaws that are difficult to understand.

With this approach, design flaws of an object-oriented system are detected at the meta-level in the Declarative Meta-Programming. The problem domain of design flaws in consideration is narrowed down to *what* is the structure of design flaws rather than *how* is the structure of such flaws. Therefore design flaws can be detected in a simple way and the suitable detection results are obtained.

Case studies are conducted to validate the proposed detection approach. Several open-source object-oriented software systems are used in experiments. The measure criterions such as precision and recall are used to assess experiment results. Results of the measured rate show good

and expected rates of detection.

5.2 Future research directions

A number of questions is encountered during the research that we believe that are worth of further investigation in the future. We classify the possible continuations of this work in two categories: refinement and integration.

Refinement

- The issue of learning classification. Although the work provides some answers on the question of how to specify and detect the design flaws, some improvement techniques are needed on refining for more general usage in widely environment.
- A rule suite of proposed detection for design flaw and recovery. Some flaws have relation among them. To uncover these flaws, it is possible to use similar rules to detect. If we can develop a suite of rules, possibly we can find the origin of such flaw. Moreover, we plan to discover the rules for correcting these flaws in meta environment after the proposed detection is performed. Because the flaw detection in meta environment is in the easy way, we believe that the flaw correction in this environment might be in the same way.
- Migration to emerging programming paradigms of base program. The question here is: How can the method and the strategies presented in this work be used beyond the limits of object-orientation? Can we, for example, define detection strategies for adaptive (AP) or aspect-oriented programming (AOP)? Which would be the invariants of the approach? Which are the parts that are going to change?

ศูนย์วิทยทรัพยากร

Integration

จุฬาลงกรณ์มหาวิทยาลัย

- The whole approach presented in this thesis, rather than being theoretical is very close to the world of practical software engineering. Therefore, we assume from the beginning that the approach will become in the near future very interesting for CASE tool providers. Some preliminary discussions with several important companies are justified our initial assumptions. This raises the question of integration, i.e. how can the techniques and methods developed during this dissertation be integrated in an existing development environment? That is the big question that we have to find out.

References

Appeltauer, M. and Kniesel, G. Towards concrete syntax patterns for logic-based transformation rules. Electron. Notes Theor. Comput. Sci., November 2008, 219:113–132.

Bell, D., Morrey, I., and Pugh, J. Software engineering: a programming approach. Hertfordshire, UK, UK, Prentice Hall International (UK) Ltd., 1987.

Booch, G. Object-Oriented Analysis and Design with Applications (3rd Edition). Redwood City, CA, USA, Addison Wesley Longman Publishing Co., Inc., 2004.

Bravo, F. M. A logic meta-programming framework for supporting the refactoring process. Master's thesis, Vrije Universiteit Brussel, 2003.

Britton, K. H., Parker, R. A., and Parnas, D. L. A procedure for designing abstract interfaces for device interface modules. In ICSE'81, pp. 195–206, 1981.

Brown, W. H., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. New York, Wiley, 1998.

Bruno, G., Garza, P., Quintarelli, E., and Rossato, R. Anomaly detection in xml databases by means of association rules. In DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications, pp. 387–391, Washington, DC, USA, 2007. IEEE Computer Society.

Coad, P. and Yourdon, E. Object-oriented design. Upper Saddle River, NJ, USA, Yourdon Press, 1991.

DeJong, G. and Mooney, R. J. Explanation-based learning: An alternative view. Machine Learning, 1986, pp. 145–176.

Demeyer, S., Ducasse, S., and Nierstrasz, O. Object Oriented Reengineering Patterns. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2002.

Deransart, P., Cervoni, L., and Ed-Dbali, A. Prolog: the Standard: Reference Manual. London, UK, Springer-Verlag, 1996.

Fagan, M. Design and code inspections to reduce errors in program development. 2002, pp. 575–607.

Fagan, M. E. Advances in software inspections. IEEE Trans. Software Eng., 1986, 12,7:744–751.

Flach, P. Simply Logical: intelligent reasoning by example. New York, NY, USA, John Wiley & Sons, Inc., 1994.

Fokaefs, M., Tsantalis, N., and Chatzigeorgiou, A. Jdeodorant: Identification and removal of feature envy bad smells. In Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, pp. 519 –520, 2007.

Fowler, M. Refactoring: Improving the Design of Existing Code. Boston, MA, USA, Addison-Wesley, 1999.

Fowler, M. and Beck, K. Bad Smells in Code, pp. 75–88. Addison-Wesley, 2000.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns. Boston, MA, Addison-Wesley, 1995.

Isner, J. A programming methodology based on data abstraction. Journal of Geodesy, 1982, 56: 149–164.

Johnson, R. E. and Foote, B. Designing reusable classes. Journal of Object-Oriented Programming, June/July 1988, 1,2:22–35.

Jorwekar, S., Fekete, A., Ramamritham, K., and Sudarshan, S. Automating the detection of snapshot isolation anomalies. In VLDB '07: Proceedings of the 33rd international conference on Very large data bases, pp. 1263–1274. VLDB Endowment, 2007.

Kniesel, G., Hannemann, J., and Rho, T. A comparison of logic-based infrastructures for concern detection and extraction. In Proceedings of the 3rd workshop on Linking aspect technology and evolution, LATE '07, New York, NY, USA, 2007. ACM.

Kowalski, R. Algorithm = logic + control. Communications of the ACM, July 1979, 22:424–436.

Kvam, K., Lie, R., and Bakkelund, D. Legacy system exorcism by pareto's principle. In Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05, pp. 250–256, New York, NY, USA, 2005. ACM.

Laird, J. E., Newell, A., and Rosenbloom, P. S. Soar: an architecture for general intelligence. Artif. Intell., September 1987, 33:1–64.

Langelier, G., Sahraoui, H., and Poulin, P. Visualization-based analysis of quality for large-scale software systems. In ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 214–223, New York, NY, USA, 2005. ACM.

Lanza, M. and Marinescu, R. Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems. Springer Verlag, 2010.

Lloyd, J. W. Foundations of logic programming. New York, NY, USA, Springer-Verlag New York, Inc., 1984.

Lloyd, J. W. Foundations of Logic Programming. Secaucus, NJ, USA, Springer-Verlag New York, Inc., 2nd edition, 1993.

Mäntylä, M. Empirical software evolvability - code smells and human evaluations. In Software Maintenance, IEEE International Conference on, pp. 1–6, 2010.

Mäntylä, M., Vanhanen, J., and Lassenius, C. A taxonomy and an initial empirical study of bad smells in code. In Proceedings of the International Conference on Software Maintenance, ICSM '03, pp. 381–, Washington, DC, USA, 2003. IEEE Computer Society.

Mäntylä, M. V., Vanhanen, J., and Lassenius, C. Bad smells " humans as code critics. In ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance, pp. 399–408, Washington, DC, USA, 2004. IEEE Computer Society.

Marinescu, R. Detecting design flaws via metrics in object-oriented systems. Technology of Object-Oriented Languages, International Conference on, 2001, 0:0173.

Marinescu, R. Detection strategies: Metrics-based rules for detecting design flaws. Software Maintenance, IEEE International Conference on, 2004, 0:350–359.

Marinescu, R. Measurement and quality in object-oriented design. In Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 701–704, Washington, DC, USA, 2005. IEEE Computer Society.

Mealy, E., Carrington, D., Strooper, P., and Wyeth, P. Improving usability of software refactoring tools. Software Engineering Conference, Australian, 2007, 0:307–318.

Mens, K. and Kellens, A. Towards a framework for testing structural source-code regularities. Software Maintenance, IEEE International Conference on, 2005, 0:679–682.

Mens, K. and Kellens, A. Intensive, a toolsuite for documenting and checking structural source-code regularities. Software Maintenance and Reengineering, European Conference on, 2006, 0:239–248.

Mens, T. and Tourwé, T. A survey of software refactoring. IEEE Trans. Softw. Eng., 2004, 30,2: 126–139.

Mens, T., Wuyts, R., Volder, K. D., and Mens, K. Declarative meta programming to support software development: Workshop report. ACM SIGSOFT Software Engineering Notes, 2003, 28,2:1.

Mihancea, P. and Marinescu, R. Towards the optimization of automatic detection of design flaws in object-oriented software systems. Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on, March 2005, pp. 92–101.

Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. Explanation-based generalization: A unifying view. Machine Learning, 1986, 1,1:47–80.

Moha, N. and Guéhéneuc, Y.-G. Decor: a tool for the detection of design defects. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ASE '07, pp. 527–528, New York, NY, USA, 2007. ACM.

Moha, N., Gueheneuc, Y.-G., and Leduc, P. Automatic generation of detection algorithms for design defects. In ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 297–300, Washington, DC, USA, 2006. IEEE Computer Society.

Murphy-Hill, E. and Black, A. Refactoring tools: Fitness for purpose. Software, IEEE, 2008a, 25 ,5:38 –44.

Murphy-Hill, E. and Black, A. P. Seven habits of a highly effective smell detector. In Proceedings of the 2008 international workshop on Recommendation systems for software engineering, RSSE '08, pp. 36–40, New York, NY, USA, 2008b. ACM.

Neighbors, J. M. The draco approach to constructing software from reusable components. IEEE Trans. Software Eng., 1984, 10,5:564–574.

Olson, D. L. and Delen, D. Advanced Data Mining Techniques. Springer Publishing Company, Incorporated, 1st edition, 2008.

Patcha, A. and Park, J.-M. An overview of anomaly detection techniques: Existing solutions and latest technological trends. Comput. Netw., 2007, 51,12:3448–3470.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Pavlik, P. I. and Anderson, J. R. An act-r model of memory applied to finding the optimal schedule of practice. In ICCM, pp. 376–377, 2004.

Pfleeger, S. L. Software Engineering: Theory and Practice. Upper Saddle River, NJ, USA, Prentice Hall PTR, 2nd edition, 2001.

Pressman, R. S. Software Engineering: A Practitioner's Approach. McGraw-Hill Higher Education, 2001.

Ratiu, D., Ducasse, S., Gerba, T., and Marinescu, R. Using history information to improve design flaws detection. Software Maintenance and Reengineering, European Conference on, 2004, 0:223.

Riel, A. J. Object-Oriented Design Heuristics. Reading, MA, Addison-Wesley, 1996.

Simon, F., Steinbrückner, F., and Lewerentz, C. Metrics based refactoring. In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, CSMR '01, pp. 30–, Washington, DC, USA, 2001. IEEE Computer Society.

Slinger, S. Code Smell Detection in Eclipse. Master's thesis, 2005.

Sommerville, I. Software engineering (5th ed.). Redwood City, CA, USA, Addison Wesley Longman Publishing Co., Inc., 1995.

Spivey, J. M. The Z notation: a reference manual. Hertfordshire, UK, UK, Prentice Hall International (UK) Ltd., 1992.

Tourwe, T. and Mens, T. A declarative meta-programming approach to framework documentation. In In Proceedings of the Workshop on Declarative Meta Programming to Support Software Development (ASE02), 2002.

Tourwé, T. and Mens, T. Identifying refactoring opportunities using logic meta programming. In Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, pp. 91–, Washington, DC, USA, 2003. IEEE Computer Society.

Travassos, G., Shull, F., Fredericks, M., and Basili, V. R. Detecting defects in object-oriented designs: using reading techniques to increase software quality. SIGPLAN Not., 1999, 34,10:47–56.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. Jdeodorant: Identification and removal of type-checking bad smells. In Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering, pp. 329–331, Washington, DC, USA, 2008. IEEE Computer Society.

Van Emden, E. and Moonen, L. Java quality assurance by detecting code smells. In Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02), pp. 97–, Washington, DC, USA, 2002. IEEE Computer Society.

Van Rompaey, B., Du Bois, B., Demeyer, S., and Rieger, M. On the detection of test smells: A metrics-based approach for general fixture and eager test. IEEE Trans. Softw. Eng., 33 ,12:800–817, 2007.

Waldinger, R. Achieving several goals simultaneously. In Elcock, E. W. and Michie, D., editors, Machine Intelligence, volume 8, pp. 94–136. Wiley, 1977.

Wuyts, R. Declarative reasoning about the structure of object-oriented systems. In TOOLS (26), pp. 112–124, 1998.

APPENDICES

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX A

EXAMPLES AND DOMAIN THEORIES OF RESEARCH

The training examples and their domain theories of learning mechanism for constructing explanations (proof trees) are shown in this appendix. The Bloaters category flaws is presented in section A.1. The Object-Oriented Abusers category, The Change Preventers category and The Dispensables are presented in section A.2, A.3 and A.4 respectively. Finally, The Couplers category is presented in section A.5.

A.1 The Bloaters Category

The Bloater flaws represent source code that has grown too large to effectively handled. It seems likely that these flaws grow a little bit at a time. Design flaws in this category are Large Class, Primitive Obsession, Long Parameter List and Data Clumps.

A.1.1 The Long Parameter List

The Long Parameter List is a design flaw which trys to unnecessarily increase coupling between classes. Instead of the called class being aware of relationships between classes, the program let the caller locate everything; then the method concentrates on what it is being asked to do with the pieces.

Examples for defining domain theories : Four examples are used to define rules for design flaw detection as followed. Ex.1 and Ex.2 show examples which are not Long Parameter List design flaws. These example is used to determine domain theories. Ex.3 and Ex.4 are Long Parameter List design flaws examples used for Explanation-based mechanism learning.

Ex.1: From `java.swing.CellRendererPane`

```
public void paintComponent(Graphics gr, Component renderer, Container parent, int
x, int y, int width, int height, Boolean shouldValidate)
```

Ex.2: From `java.awt.Graphics`

```
public Boolean drawImage(Image image, int x1Dest, int y1Dest, int x2Dest,
```

```
int y2Dest, int x1Source, int y1Source, int x2Source, int y2Source, Color color,
ImageObserver obs)
```

Ex.3: From `java.swing.DefaultBoundedRangeModel`

```
    public void setRangeProperties(int newValue, int newExtent, int newMin, int
newMax, boolean isAdjusting)
```

Ex.4: From `java.swing.JOptionPane`

```
    public static int showConfirmDialog(String title, int optionType, int messageType)
```

longParameterList(x) is a target concept that uses for learning. Table A.1 shows a target concept and domain theories which are used to construct a detection rule. Horn clause in R8 cannot be used here because it derives a inconsistent rule inconsistency. The sematic in Figure A.1 shows the detection rule covering of Ex3. and Ex4. in domain theories after performing learning algorithm. The detection rule for Long Parameter List flaw is constructed by covering both Ex.3 and Ex.4.

Table A.1: Domain theories and a target concept of Ex.1- Ex.4 *Long Parameter List*

Target concepts and domain theory:
R1: $\forall x \forall y$ method(x) $\wedge$ hasLongParameter(x, y)
$\Rightarrow$ longParameterList(x)
R2: $\forall x$ methodNoReturnType(x) $\Rightarrow$ method(x)
R3: $\forall x$ methodHasReturnType(x) $\Rightarrow$ method(x)
R4: $\forall x$ publicMethod(x) $\Rightarrow$ method(x) .
R5: $\forall x \forall y$ parameterList(y) $\Rightarrow$ hasLongParameter(x, y)
R6: $\forall y \forall y_i$ $((\forall y_i \in y)$ eachParameter$(y_i))$ $\Rightarrow$ parameterList(y)
R7: $\forall y_i$ basicTypeParameter(y_i) $\Rightarrow$ eachParameter(y_i)
R8: $\forall y_i$ classTypeParameter(y_i) $\Rightarrow$ eachParameter(y_i)
R9: $\forall y_i$ genericTypeParameter(y_i) $\Rightarrow$ eachParameter(y_i)

A rule for detecting Long Parameter List flaw can be constructed in form of MESs as:

```
%==============================================================================
% Rules Long Parameter List
longParameterList(MethodID,ClassID,MethodName) :-
    classT(ClassID,_,_,_),
```

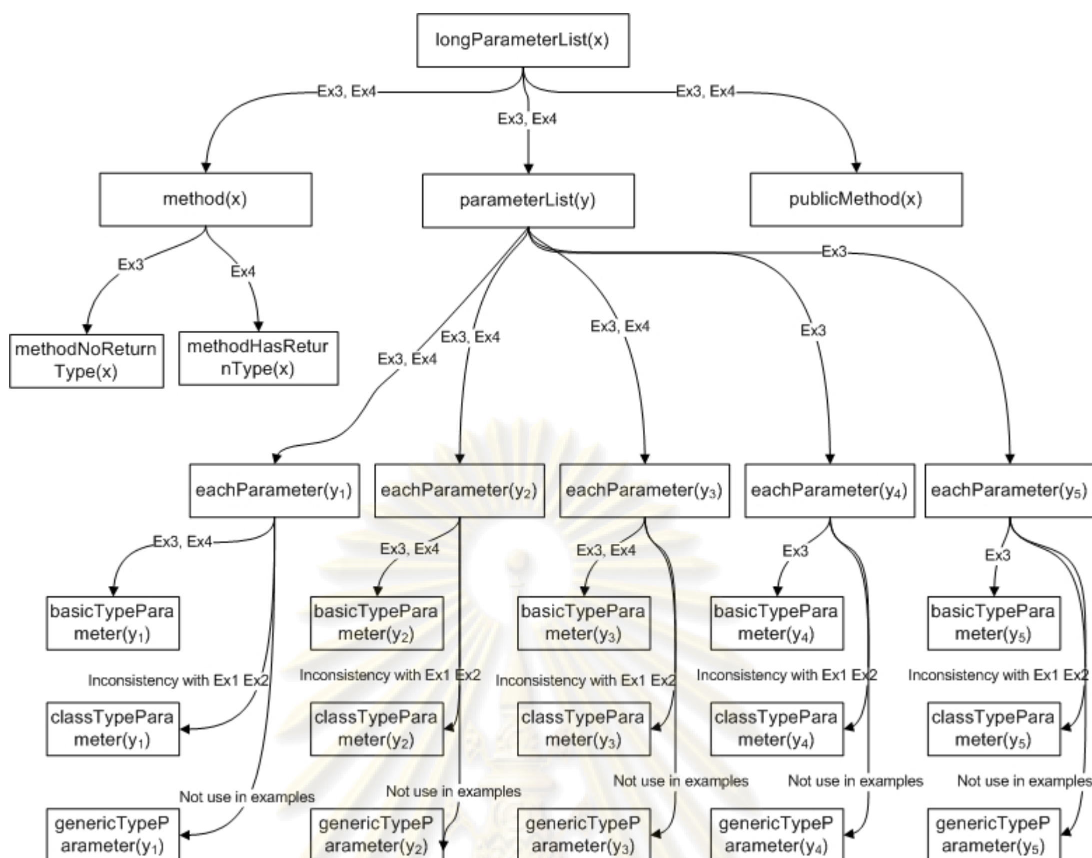


Figure A.1: The sematic net shows covering of Ex3. and Ex4. in domain theories

```
    methodT(MethodID,ClassID,MethodName,ParameterList,_,_,_),
    [first,second,third,rest] = [ParameterList],
    \+parameterbeObject(first),
    \+parameterbeObject(second),
    \+parameterbeObject(third).

parameterbeObject(parameterID) :-
    atom(parameterID),
    sub_atom(parameterID, _, _, _, class).}
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

A.1.2 Large Class

Large Class flaw grows big a little bit in each evolution time. The programmer keeps adding more capabilities to a class until it eventually grows too big. Sometimes the problem is a lack of insight into the parts that make up the whole class. In any case, the class represents too many responsibilities folded together.

Examples for defining domain theories: An example of Java Servlet Front Strategy code is used to define rules for design flaw detection as followed.

```
public class EmployeeController extends HttpServlet {
  // Initializes the servlet.
  public void init(ServletConfig config) throws
    ServletException {
    super.init(config);
  }

  // Destroys the servlet.
  public void destroy() {
  }

  /** Processes requests for both HTTP
   * <code>GET</code> and <code>POST</code> methods.
   * @param request servlet request
   * @param response servlet response
   */
  protected void processRequest(HttpServletRequest
    request, HttpServletResponse response)
    throws ServletException, java.io.IOException {
    String page;

    /**ApplicationResources provides a simple API
     * for retrieving constants and other
     * preconfigured values**/
    ApplicationResources resource =
      ApplicationResources.getInstance();
    try {

      // Use a helper object to gather parameter
      // specific information.
      RequestHelper helper = new
         RequestHelper(request);

      Command cmdHelper= helper.getCommand();

      // Command helper perform custom operation
      page = cmdHelper.execute(request, response);

    }
    catch (Exception e) {
      LogManager.logMessage(
        "EmployeeController:exception : " +
        e.getMessage());
      request.setAttribute(resource.getMessageAttr(),
        "Exception occurred : " + e.getMessage());
      page = resource.getErrorPage(e);
    }
```

```
    // dispatch control to view
    dispatch(request, response, page);
  }

  /** Handles the HTTP <code>GET</code> method.
   * @param request servlet request
   * @param response servlet response
   */
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
      processRequest(request, response);
  }

  /** Handles the HTTP <code>POST</code> method.
   * @param request servlet request
   * @param response servlet response
   */
  protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
        processRequest(request, response);
  }

  /** Returns a short description of the servlet */
  public String getServletInfo() {
    return "Front Controller Pattern" +
      " Servlet Front Strategy Example";
  }

  protected void dispatch(HttpServletRequest request,
    HttpServletResponse response,
    String page)
  throws  javax.servlet.ServletException,
    java.io.IOException {
    RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher(page);
    dispatcher.forward(request, response);
  }
}
```

largeClass(x) is a target concept that uses for learning. Table A.2 shows a target concept and domain theories which are used to construct a detection rule.

All rules for detecting Large Class flaw can be constructed in form of MESs as:

Table A.2: Domain theories and a target concept of *Large Class* design flaw

A target concepts and domain theories:

R1: $\forall c$ class(c) $\wedge$ controllerClass(c) $\Rightarrow$ largeClass(c)

R2: $\forall c \forall m \forall a$ hasMethod(c,m) $\wedge$ method(m) $\wedge$ hasAttribute(c,a) $\wedge$ attribute(a) $\Rightarrow$ class(c)

R3: $\forall c \forall m \forall a$ execMethod(m) $\wedge$ $\neg$hasAttribute(c,a) $\Rightarrow$ controllerClass(c)

```
%===============================================================================
% Rules Large Class
LargeClass(ClassID) :- monopolizeClass(ClassID).
monopolizeClass(ClassID) :-
    accessDataClass(ClassID, ClassID_DataClass),write('Exist From 1. Access Data Class ');
    accessUnusedData(ClassID, ClassID_UnusedData),
        write('Exist From 2. Unused Data in Class ');
    controllerClass(ClassID),write('Exist From 3. Controller Class ');
    unusedElementClass(ClassID),write('Exist From 4. Unused Element in Class ');
    compositeClass(ClassID),write('Exist From 5. CompositClass ').

accessDataClass(ClassID, ClassID_DataClass) :-
    classT(ClassID,_,_,_), classT(ClassID_DataClass,_,_,_),
    ClassID \== ClassID_DataClass,
    classAccess(ClassID,ClassID_DataClass),
    dataClass(ClassID_DataClass).

classAccess(ClassID,ClassID_DataClass) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
blockT(BlockID,MethodID,_,_),
    execT(ExecID,BlockID,_,_),
    callT(_,_,MethodID,_,_,_,MethodID_dataclass),
    classT(ClassID_DataClass,_,_,_),
    methodT(MethodID_dataclass,ClassID_DataClass, _,_,_,_,_).

accessorMethod(AccessorMethodID)  :-
    classT(X,_,_,_),
    fieldT(Y,X,_,_,_),
    methodT(AccessorMethodID,X,_,[],_,_,_),
    blockT(A,AccessorMethodID,_,_),
    returnT(_,A,AccessorMethodID,_),
    getFieldT(_,_,AccessorMethodID,_,_,Y).

mutatorMethod(MutatorMethodID)  :-
    classT(X,_,_,_),
    fieldT(Y,X,_,A,_),
    methodT(MutatorMethodID,X,_,[B],_,_,_),
    paramT(B,MutatorMethodID,_,A),
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
    assignT(_,_,MutatorMethodID,C,_),
    getFieldT(C,_,MutatorMethodID,_,A,Y).

notDataClass(NotDataClassID)  :-
    classT(NotDataClassID,_,_,_),
    methodT(MethodInDataClass,NotDataClassID,_,_,_,_,_),
    \+ mutatorMethod(MethodInDataClass),
    \+ accessorMethod(MethodInDataClass).

dataClass(ClassID_DataClass) :-
    classT(ClassID_DataClass,_,_,_),
\+ notDataClass(ClassID_DataClass).

accessUnusedData(ClassID, ClassID_UnusedData) :-
    classT(ClassID,_,_,_),
    classT(ClassID_UnusedData,_,_,_),
    ClassID \== ClassID_UnusedData,
    accessUnusedData(ClassID,ClassID_UnusedData,MethodID_UnusedData),
    unusedDataClass(ClassID_UnusedData,MethodID_UnusedData).

accessUnusedData(ClassID,ClassID_UnusedData,MethodID_UnusedData) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
blockT(BlockID,MethodID,_,_),
execT(ExecID,BlockID,_,_),
callT(_,_,MethodID,_,_,_,MethodID_UnusedData),
methodT(MethodID_UnusedData,ClassID_UnusedData, _,_,_,_,_).

unusedDataClass(ClassID_UnusedData,MethodID_UnusedData) :-
    \+ usedDataClass(ClassID_UnusedData,MethodID_UnusedData).

usedDataClass(ClassID_UnusedData,MethodID_UnusedData) :-
    mutatorMethod(MethodID_UnusedData);
    accessorMethod(MethodID_UnusedData),
    methodT(MethodID_UnusedData,ClassID_UnusedData,_,_,_,_,_),
    methodT(UseMethod,ClassID_UnusedData,_,_,_,_,_),
    blockT(Temp,UseMethod),
    execT(Temp2, Temp),
    callT(_,Temp2,_,_,_,_,MethodID_UnusedData).

controllerClass(ClassID) :-
    classT(ClassID,_,_,_),
    \+ fieldT(FieldID,ClassID,_,_,_).

unusedElementClass(ClassID) :-
    unusedElementField(ClassID),
    unusedElementMethod(ClassID).

unusedElementField(ClassID) :-
    classT(ClassID,_,_,_),
    fieldT(FieldID,ClassID,_,_,_),
```

```
    methodT(MethodID,ClassID,_,_,_,_,_),
    \+getFieldT(_,_,MethodID,_,_,FieldID).

unusedElementMethod(ClassID) :-
    classT(ClassID,_,_,_),
    methodT(MethodID,ClassID,_,_,_,_,_),
    newClassT(NewClassID,_,_,_,_,ClassID,_,_),
    \+ callT(CallID,_,_,_,_,_,MethodID),
    identT(_,CallID,_,_,LocalID),
    localT(LocalID,_,_,_,_,NewClassID).

connected(X,Y) :-
    edge(X,Y) ; edge(Y,X).

path(A,B,Path) :-
    travel(A,B,[A],Q),
    reverse(Q,Path),!. % cut backtracking

travel(A,B,P,[B|P]) :-
    connected(A,B).

travel(A,B,Visited,Path) :-
    connected(A,C),
    C \== B,
    not(member(C,Visited)),
    travel(C,B,[C|Visited],Path) ;
    Path = [B].

element_of(X,[X|Tail]).
element_of(X,[_|Tail]) :- element_of(X,Tail).

add2end(X,[H|T],[X,H|T]):-add2end(X,T,[X|T]).
add2end(X,[],[X]).

addElement(X,[X]).
addElement(X,[X|Element]) :- addElement(X,Element).

checkPath(PathSourceSink,Source,Sink,A_Temp) :-
    PathSourceSink = [Sink], nl,
    add2end(Sink,A_Temp,A_Temp1), nl,
    fail.

compositeClass(ClassID) :-
    \+ noGodclass(ClassID).

noGodclass(ClassID) :-
    classT(ClassID,_,_,ListElement),fieldT(FieldID,ClassID,_,_,_),
    methodT(MethodID,ClassID,_,_,_,_,_),
    edge(FieldID,MethodID),!,
    methodT(MethodA,ClassID,_,_,_,_,_),
```

```
    methodT(MethodB,ClassID,_,_,_,_,_),
    edge(MethodA,MethodB),!,
    pathTraverse(FieldID,MethodID,ListElement).

edge(FieldID,MethodID) :-
    getFieldT(_,_,MethodID,_,_,FieldID).

edge(MethodA,MethodB) :-
    MethodA \== MethodB,
    callT(_,_,MethodA,_,_,_,MethodB).

pathTraverse(FieldID,MethodID,ListElement) :-
    [A,B|C] = ListElement,
    A_Temp = [A],!,
    element_of(Y,C),
    path(B,Y,Path),
    checkPath(Path,B,Y,A_Temp),nl.
```

A.1.3 Primitive Obsession

Primitive Obsession flaw is actually more of a symptom that causes bloats than a bloat itself. When the Primitive Obsession occurs, there are no small classes for small entities (e.g. phone numbers). Thus, the functionality is added to some other class which increase the class and method size in the software.

Examples for defining domain theories: Two examples of Fowler's workbook are used to define rules for design flaw detection as followed.

```
public class Age {
    private int age;

    public Age(int age) {
        this.age = age; }
    public int toInt() {
        return age; }
}

class Person {
    public static final int O = 0;
    public static final int A = 1;
    public static final int B = 2;
    public static final int AB = 3;
    private int _bloodGroup;

    public Person (int bloodGroup) {
        _bloodGroup = bloodGroup;
    }
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
    public void setBloodGroup(int arg) {
        _bloodGroup = arg;
    }

    public int getBloodGroup() {
        return _bloodGroup;
    }
  }
```

primitiveObsession(x) is a target concept that uses for learning. Table A.3 shows a target concept and domain theories which are used to construct a detection rule.

Table A.3: Domain theories and a target concept of *Primitive Obsession* design flaw

A target concepts and domain theories:
R1: $\forall c$ class(c) $\wedge$ $\neg$objectObsession(c) $\wedge$ $\neg$genericObsession(c) $\Rightarrow$ primitiveObsession(c)
R2: $\forall c \forall a$ hasAttribute(c,a) $\wedge$ attribute(a) $\Rightarrow$ class(c)
R3: $\forall c \forall a$ attributeType(a,'Object') $\Rightarrow$ objectObsession(c)
R4: $\forall c \forall a$ attributeType(a,'Generic') $\Rightarrow$ genericObsession(c)

All rules for detecting Primitive Obsession flaw can be constructed in form of MESs as:

```
%==============================================================================
% Rules Primitive Obsession
primitiveObsession(ClassID,ClassName,Pathsource) :-
    projectRequired(icice,Pathsource,ClassID,ClassName),
    primitiveObsessionRules(ClassID).


primitiveObsessionRules(ClassID) :-
    haveGenericType(ClassID).
    simulatedAccessors(ClassID).


havePrimitiveType(ClassID) :- \+haveGenericType(ClassID).

haveGenericType(ClassID) :-
    fieldT(_,ClassID,Type,_,_),
    \+arg(1,Type,class),
    \+arg(1,Type,generic).
```

```
simulatedAccessors(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    localT(DeclareArrayID,_,MethodID,_,_,_),
    newArrayT(_,DeclareArrayID,MethodID,_,_,_),
    % 1 element insert array
    assignT(AssignID,_,_,_,_),
    indexedT(IndexID,AssignID,_,_,_),
    identT(_,IndexID,_,_,DeclareArrayID),
    % 2 element insert array
    assignT(AssignID2,_,_,_,_),
    indexedT(IndexID2,AssignID2,_,_,_),
    identT(_,IndexID2,_,_,DeclareArrayID),
    AssignID \= AssignID2,!.

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.1.4 Data Clump

With Data Clump flaws, there is a set of primitives that always appear together. Since these data items are not encapsulated in a class. This increases the sizes of methods and classes.

Examples for defining domain theories: A example of Fowler's Refactoring is used to define rules for design flaw detection as followed. A pair of range values is considered in this example.

```
public class Account{
    double getFlowBetween (Date start, Date end){
        double result = 0;
        Enumeration e = _entries.elements();
        while(e.hasMoreElements()){
            Entry each = (Entry) e.nextElement();
            if (each.getDate().equals(start)) ||
                each.getDate().equals(end) ||
                (each.getDate().after(start) && each.getDate.before(end)))
            {
                result += each.getValue(0;
            }
        }
    return result;
}
```

dataClump(x) is a target concept that uses for learning. Table A.4 shows a target concept and domain theories which are used to construct a detection rule.

Table A.4: Domain theories and a target concept of *Data Clump* design flaw

A target concepts and domain theories:
R1: $\forall a_1 \forall a_2$ class(c) $\wedge$ clumpPair(a_1, a_2) $\wedge$ clumpFirst(a_1, a_2)
$\wedge$ clumpSecond(a_1, a_2)
$\Rightarrow$ dataClump(c)
R2: $\forall a_1 \forall a_2$ attribute(a_1) $\wedge$ attribute(a_2) $\wedge$ $(a_1 \neq a_2)$
$\Rightarrow$ clumpPair(a_1, a_2)
R3: hasMethod(c, m_1) $\wedge$ method(m_1) $\Rightarrow$ (c)
R4: hasMethod(c, m_2) $\wedge$ method(m_2) $\Rightarrow$ (c)
R5: hasParameter(m_1, m_{1-para}) $\wedge$ parameter(m_{1-para}) $\Rightarrow$ method(m_1)
R6: hasParameter(m_2, m_{2-para}) $\wedge$ parameter(m_{2-para}) $\Rightarrow$ method(m_2)
R7: parameterIn(a_1, m_{1-para}) $\wedge$ parameterIn(a_2, m_{1-para})
$\Rightarrow$ clumpFirst(a_1, a_2)
R7: parameterIn(a_1, m_{2-para}) $\wedge$ parameterIn(a_2, m_{2-para})
$\Rightarrow$ clumpSecond(a_1, a_2)

All rules for detecting Data Clump can be constructed in form of MESs as:

```
%=============================================================================
% Rule DataClumps
dataClump(Pathsource,ClassID,ClassName) :-
    projectRequired(SourceProgram,Pathsource,ClassID,ClassName),
    dataClumps(ClassID).

dataClumps(ClassID) :-
    dataClumps2Ele(ClassID),
    dataClumps2Get(ClassID),
    dataClumps3Get(ClassID).

dataClumps2Ele(ClassID) :-
    fieldT(FieldID1,_,_,_,_),
    fieldT(FieldID2,_,_,_,_),
    FieldID1 \= FieldID2,

    methodT(Method1ID,ClassID,_,Parameter1,_,_,_),
    methodT(Method2ID,ClassID,_,Parameter2,_,_,_),
    MethodID1 \= MethodID2,

    paramT(Parameter1ID,Method1ID,_,_),
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
    member(Parameter1ID,Parameter1),
    getFieldT(_,_,Method1ID,_,_,FieldID1),
    identT(_,_,MethodID1ID,_,Parameter1ID),

    paramT(Parameter2ID,Method1ID,_,_),
    member(Parameter2ID,Parameter1),
    getFieldT(_,_,Method1ID,_,_,FieldID2),
    identT(_,_,MethodID1ID,_,Parameter2ID),

    paramT(Parameter1ID,Method2ID,_,_),
    member(Parameter1ID,Parameter1),
    getFieldT(_,_,Method2ID,_,_,FieldID1),
    identT(_,_,MethodID2ID,_,Parameter1ID),

    paramT(Parameter2ID,Method2ID,_,_),
    member(Parameter2ID,Parameter1),
    getFieldT(_,_,Method2ID,_,_,FieldID2),
    identT(_,_,MethodID2ID,_,Parameter2ID).


dataClumps2Get(ClassID) :-
    fieldT(FieldID1,_,_,_,_),
    fieldT(FieldID2,_,_,_,_),
    FieldID1 \= FieldID2,

    methodT(Method1ID,ClassID,_,_,_,_,_),
    methodT(Method2ID,ClassID,_,_,_,_,_),
    MethodID1 \= MethodID2,

    accessAttributeFromMethod(Method1ID,AttributeID1),
    accessAttributeFromMethod(Method1ID,AttributeID2),
    accessAttributeFromMethod(Method2ID,AttributeID1),
    accessAttributeFromMethod(Method2ID,AttributeID2).

dataClumps3Get(ClassID) :-
    fieldT(FieldID1,ClassID,_,_,_),
    fieldT(FieldID2,ClassID,_,_,_),
    fieldT(FieldID3,ClassID,_,_,_),
    FieldID1 \= FieldID2,
    FieldID1 \= FieldID3,
    FieldID2 \= FieldID3,
    methodT(MethodID,ClassID,_,_,_,_,_),
    accessAttributeFromMethod(MethodID,FieldID1),
    accessAttributeFromMethod(MethodID,FieldID2),
    accessAttributeFromMethod(MethodID,FieldID3),!.

accessAttributeFromMethod(Method_ID,Attribute_ID) :-
    getFieldT(_,_,Method_ID,_,_,Attribute_ID);
    callT(_,_,Method_ID,_,_,_,Method_Temp_ID),
    getFieldT(_,_,Method_Temp_ID,_,_,Attribute_ID).
```

```
projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.2 The Object-Oriented Abusers Category

The common denominator for the flaws in the Object-Orientation Abuser category is that they represent cases where the solution does not fully exploit the possibilities of object-oriented design. Design flaws in this category are Switch Statements, Temporary Field, Refused Bequest and Alternative Classes with Different Interfaces.

A.2.1 Switch Statements

A Switch Statement might be considered acceptable or even good design in procedural programming, but it is something that should be avoided in object-oriented programming. The situation where switch statements or type codes are needed should be handled by creating sub-classes.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";       break;
            case 2:  monthString = "February";      break;
            case 3:  monthString = "March";         break;
            case 4:  monthString = "April";         break;
            case 5:  monthString = "May";           break;
            case 6:  monthString = "June";          break;
            case 7:  monthString = "July";          break;
            case 8:  monthString = "August";        break;
            case 9:  monthString = "September";     break;
            case 10: monthString = "October";       break;
            case 11: monthString = "November";      break;
            case 12: monthString = "December";      break;
            default: monthString = "Invalid month"; break;
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
        }
        System.out.println(monthString);
    }
}
```

switchStatement(x) is a target concept that uses for learning. Table A.5 shows a target concept and domain theories which are used to construct a detection rule.

Table A.5: Domain theories and a target concept of *Switch Statements* design flaw

Target concepts and domain theory:
R1: $\forall x \forall y$ method(x) $\wedge$ statement(y) $\wedge$ hasSwitchStatements(x, y) $\Rightarrow$ switchStatements(x) .
R2: $\forall y$ hasSwitchOperation(y) $\Rightarrow$ statement(y)

A rule for detecting Switch Statement flaw can be constructed in form of MESs as:

```
%==============================================================================
% Rule Switch Statement
switchStatement(PathProject,ClassID,ClassName) :-
    projectRequired('SourceProgram',PathProject,ClassID,ClassName),
    methodT(MethodID,ClassID,_,_,_,_,_),
    switchT(_,_,MethodID,_,_),
    write(' ######## Switch Statement : Switch Exist'), nl.

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.2.2 Temporary Field

This flaw can happen when one part of an object has an algorithm that passes around information through the fields rather than through parameters – the fields are valid or used only when the algorithm is active. These fields do not suggest which there may be a missing object whose life cycle differs from the object holding it.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class Logo extends Canvas {
    private Image fImage;
    public int fWidth;
    public int fHeight;

    public Logo() {
    fImage= loadImage("logo.gif");
    MediaTracker tracker= new MediaTracker(this);
    tracker.addImage(fImage, 0);
    try {
            tracker.waitForAll();
        } catch (Exception e) {
        }

        if (fImage != null) {
            fWidth= fImage.getWidth(this);
            fHeight= fImage.getHeight(this);
        } else {
            fWidth= 20;
            fHeight= 20;
        }
    setSize(fWidth, fHeight);
}
```

temporaryField(x) is a target concept that uses for learning. Table A.6 shows a target concept and domain theories which are used to construct a detection rule.

Table A.6: Domain theories and a target concept of *Temporary Field* design flaw

A target concepts and domain theories:
R1: $\forall x \forall y$ class(y) $\wedge$ attribute(x) $\wedge$ hasAttribute(y, x) $\Rightarrow$ temporaryField(x)
R2: $\forall x$ hasAttrModifierPublic(x) $\Rightarrow$ attribute(x)
R3: $\forall x$ hasAttrModifierPrivate(x) $\Rightarrow$ attribute(x)
R4: $\forall x$ hasAttrModifierProtected(x) $\Rightarrow$ attribute(x)
R5: $\forall x$ isStatic(x) $\Rightarrow$ attribute(x)
R6: $\forall x$ $\neg$isStatic(x) $\Rightarrow$ attribute(x)

All rules for detecting Temporary Field flaw can be constructed in form of MESs as:

```
%==============================================================================
% Rule Temporary Field
temporalField(PathProject,ClassID,ClassName,FieldID) :-
   projectRequired('CommonCLI',PathProject,ClassID,ClassName),
   fieldT(FieldID,ClassID,_,_,_),
   modifierT(FieldID,public),
   write(' ######## 1 Temporal Field : Private Attribute'),nl;

   projectRequired(ieice,PathProject,ClassID,ClassName),
   fieldT(FieldID,ClassID,_,_,_),
   methodT(MethodID,ClassID,_,_,_,_,_),
   assignT(AssignID,_,MethodID,_,_),
   getFieldT(_,AssignID,_,_,_,FieldID),
   identT(_,AssignID,_,_,null),
   write(' ######## 2 Temporal Field : Set Null Attribute & Object'),nl;

    projectRequired(ieice,PathProject,ClassID,ClassName),
    fieldT(FieldID,ClassID,_,_,_),
    identT(_,FieldID,_,_,null),
    write(' ######## 3 Temporal Field : assign Attribute and null value'),nl.

    projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.2.3 Refused Bequest

A class may inherit from another class just for implementation convenience without really intending the class to be substitutable for the parent. Or, there may be a conscious decision to let subclasses deny use of some features to prevent an explosion of types for all feature combinations. This situation in such software systems is Refused Bequest flaw .

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
package junit.framework;

/**
 * A Listener for test progress
 */
public interface TestListener {
/**
   * An error occurred.
```

```
   */
public void addError(Test test, Throwable t);
/**
   * A failure occurred.
   */
  public void addFailure(Test test, AssertionFailedError t);
/**
 * A test ended.
 */
  public void endTest(Test test);
/**
 * A test started.
 */
public void startTest(Test test);
}

class TestSuitePanel extends JPanel implements TestListener {
private JTree fTree;
private JScrollPane fScrollTree;
private TestTreeModel fModel;

static class TestTreeCellRenderer extends DefaultTreeCellRenderer {
private Icon fErrorIcon;
private Icon fOkIcon;
private Icon fFailureIcon;

TestTreeCellRenderer() {
     super();
     loadIcons();
}

void loadIcons() {
fErrorIcon= TestRunner.getIconResource(getClass(), "icons/error.gif");
     fOkIcon= TestRunner.getIconResource(getClass(), "icons/ok.gif");
     fFailureIcon= TestRunner.getIconResource(getClass(), "icons/failure.gif");
}

String stripParenthesis(Object o) {
String text= o.toString ();
     int pos= text.indexOf('(');
     if (pos < 1)
     return text;
     return text.substring (0, pos);
   }

public Component getTreeCellRendererComponent(JTree tree, Object value,
boolean sel, boolean expanded, boolean leaf, int row, boolean hasFocus) {

Component c = super.getTreeCellRendererComponent
                (tree, value, sel, expanded, leaf, row, hasFocus);
```

```
   TreeModel model= tree.getModel();
   if (model instanceof TestTreeModel) {
TestTreeModel testModel= (TestTreeModel)model;
Test t= (Test)value;
String s= "";
     if (testModel.isFailure(t)) {
     if (fFailureIcon != null)
     setIcon(fFailureIcon);
     s= " - Failed";
     }
     else if (testModel.isError(t)) {
     if (fErrorIcon != null)
     setIcon(fErrorIcon);
     s= " - Error";
     }
     else if (testModel.wasRun(t)) {
     if (fOkIcon != null)
     setIcon(fOkIcon);
     s= " - Passed";
     }
     if (c instanceof JComponent)
     ((JComponent)c).setToolTipText(getText()+s);
   }
   setText(stripParenthesis(value));
return c;
}
}

public TestSuitePanel() {
super(new BorderLayout());
setPreferredSize(new Dimension(300, 100));
fTree= new JTree();
fTree.setModel(null);
fTree.setRowHeight(20);
ToolTipManager.sharedInstance().registerComponent(fTree);
fTree.putClientProperty("JTree.lineStyle", "Angled");
fScrollTree= new JScrollPane(fTree);
add(fScrollTree, BorderLayout.CENTER);
}

public void addError(final Test test, final Throwable t) {
  fModel.addError(test);
fireTestChanged(test, true);
}

public void addFailure(final Test test, final AssertionFailedError t) {
  fModel.addFailure(test);
fireTestChanged(test, true);
}
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
/**
   * A test ended.
   */
  public void endTest(Test test) {
fModel.addRunTest(test);
  fireTestChanged(test, false);
  }

/**
   * A test started.
   */
  public void startTest(Test test) {
  }

/**
 * Returns the selected test or null if multiple or none is selected
 */
public Test getSelectedTest() {
TreePath[] paths= fTree.getSelectionPaths();
if (paths != null && paths.length == 1)
return (Test)paths[0].getLastPathComponent();
return null;
}

/**
 * Returns the Tree
 */
public JTree getTree() {
return fTree;
}

/**
 * Shows the test hierarchy starting at the given test
 */
public void showTestTree(Test root) {
        fModel= new TestTreeModel(root);
        fTree.setModel(fModel);
        fTree.setCellRenderer(new TestTreeCellRenderer());
    }

private void fireTestChanged(final Test test, final boolean expand) {
        SwingUtilities.invokeLater(
            new Runnable() {
                public void run() {
                    Vector vpath= new Vector();
                    int index= fModel.findTest(test, (Test)fModel.getRoot(), vpath);
                    if (index >= 0) {
                        Object[] path= new Object[vpath.size()];
                        vpath.copyInto(path);
                        TreePath treePath= new TreePath(path);
```

```
                        fModel.fireNodeChanged(treePath, index);
                        if (expand) {
                            Object[] fullPath= new Object[vpath.size()+1];
                            vpath.copyInto(fullPath);
                            fullPath[vpath.size()] =
                                fModel.getChild(treePath.getLastPathComponent(), index);
                            TreePath fullTreePath= new TreePath(fullPath);
                            fTree.scrollPathToVisible(fullTreePath);
                        }
                    }
                }
            }
        );
    }
}
```

refusedBequest(x) is a target concept that uses for learning. Table A.7 shows a target concept and domain theories which are used to construct a detection rule.

Table A.7: Domain theories and a target concept of *Refused Bequest* design flaw

A target concepts and domain theories:
R1: $\forall x_i$ (i∈1,2)$\forall y \forall z$ class(x_i) ∧ attribute(z) ∧ hasAttribute(x_i, z) ∧ method(y) ∧ hasMethod(x_i, y) ∧ notInheritOperation(x_1,x_2) ⇒ refusedBequest(x_2)
R2: $\forall z$ hasAttrModifierPublic(z) ⇒ attribute(z)
R3: $\forall z$ hasAttrModifierPrivate(z) ⇒ attribute(z)
R4: $\forall z$ hasAttrModifierProtected(z) ⇒ attribute(z).
R5: $\forall z \forall z_{name}$ hasName(z, z_{name}) ⇒ attribute(z)
R6: $\forall y \forall y_{name}$ hasName(y, y_{name}) ⇒ method(z)
R7: $\forall y$ hasMetModifierPublic(y) ⇒ attribute(y)
R8: $\forall y$ hasMetModifierPrivate(y) ⇒ attribute(y)
R9: $\forall y$ hasMetModifierProtected(y) ⇒ attribute(y)
R10: $\forall x_i$ (i∈1,2)$\forall y_{name1} \forall y_{name2}$ notTheSame(y_{name1},y_{name1}) ⇒ notInheritOperation(x_1,x_2)

All rules for detecting Refused Bequest flaw can be constructed in form of MESs as:

```
%===========================================================================
% Rule Refused Bequest
```

```
refuseBequest(Pathsource,ClassID,ClassName) :-
    projectRequired('SourceProgram',Pathsource,ClassID,ClassName),
    refuseBequestAnyForm(ClassID).

refuseBequestAnyForm(ClassID) :-
    refuseBequestClassForm(ClassID).
    refuseBequestClassFormNotImplement(ClassID);
    refuseBequestInterfaceFormImplementClassBlank(ClassID).
    refuseBequestInterfaceFormImplementAbsBlank(ClassID).

% First Form (no inherit method)
refuseBequestClassForm(ClassID) :-
    classT(ClassID,_,_,ElementInClass),
    classT(ClassSuperID,_,_,EleInSuper),
    extendsT(ClassID,ClassSuperID),
    \+globalIds('java.lang.Object',ClassSuperID),
    \+notRefuseBequestClassForm(ClassID,ClassSuperID).

notRefuseBequestClassForm(ClassID,ClassSuperID) :-
    methodT(MethodSuperID,ClassSuperID,MethodSuperName,_,_,_,_),
    methodT(MethodID,ClassID,MethodName,_,_,_,_),
    MethodSuperName = MethodName,!.

% Second Form (class not implement)
refuseBequestClassFormNotImplement(ClassID) :-
    classT(ClassID,_,_,ElementInClass),
    classT(ClassSuperID,_,_,EleInSuper),
    extendsT(ClassID,ClassSuperID),
    \+globalIds('java.lang.Object',ClassSuperID),
refuseBequestCFNI(ClassID,ClassSuperID).

refuseBequestCFNI(ClassID,ClassSuperID) :-
    methodT(MethodSuperID,ClassSuperID,MethodSuperName,_,_,_,_),
    methodT(MethodID,ClassID,MethodName,_,_,_,_),
    MethodSuperName = MethodName,
    blockT(_,MethodID,_,[]),!.

% Third Form (implement interface blank)
refuseBequestInterfaceFormImplementClassBlank(ClassID) :-
classT(ClassID,_,_,ElementInClass),
implementsT(ClassID,ClassSuperID),
    classT(ClassSuperID,_,_,_),
    interfaceT(ClassSuperID),
    \+globalIds('java.lang.Object',ClassSuperID),
refuseBequestIFICB(ClassID,ClassSuperID).

refuseBequestIFICB(ClassID,ClassSuperID) :-
    methodT(MethodSuperID,ClassSuperID,MethodSuperName,_,_,_,_),
    methodT(MethodID,ClassID,MethodName,_,_,_,_),
    MethodSuperName = MethodName,
```

```
    blockT(_,MethodID,_,[]),!.

% Third Abstract Form (implement interface blank)
refuseBequestInterfaceFormImplementAbsBlank(ClassID):-
classT(ClassID,_,_,_),
implementsT(ClassID,ClassSuperID),
modifierT(ClassID,abstract),
    classT(ClassSuperID,_,_,_),
    interfaceT(ClassSuperID),
    \+globalIds('java.lang.Object',ClassSuperID),
refuseBequestIFIAB(ClassID,ClassSuperID).

refuseBequestIFIAB(ClassID,ClassSuperID) :-
    methodT(MethodSuperID,ClassSuperID,MethodSuperName,_,_,_,_),
    methodT(MethodID,ClassID,MethodName,_,_,_,_),
    MethodSuperName = MethodName,
    blockT(_,MethodID,_,[]),!.

% Find Path
projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
projectS(ProjectID,NameProject,_,_,_),
sourceFolderS(SourceID,ProjectID,_),
fileS(FileID,SourceID,Pathsource),
compilationUnitT(ClassCompilationID,_,FileID,_,_),
classT(ClassID,ClassCompilationID,ClassName,_).
```

A.2.4 Alternative Classes with Different Interfaces

The Alternative Classes with Different Interfaces flaw shows that two classes seem to be doing the same thing but are using different class names.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
 public interface TestRunListener {
     /* test status constants*/
     public static final int STATUS_ERROR= 1;
     public static final int STATUS_FAILURE= 2;

     public void testRunStarted(String testSuiteName, int testCount);
     public void testRunEnded(long elapsedTime);
     public void testRunStopped(long elapsedTime);
     public void testStarted(String testName);
     public void testEnded(String testName);
     public void testFailed(int status, String testName, String trace);
}
```

```
public class TestRunner extends BaseTestRunner {
private ResultPrinter fPrinter;

public static final int SUCCESS_EXIT= 0;
public static final int FAILURE_EXIT= 1;
public static final int EXCEPTION_EXIT= 2;

/**
 * Constructs a TestRunner.
 */
public TestRunner() {
this(System.out);
}

/**
 * Constructs a TestRunner using the given stream for all the output
 */
public TestRunner(PrintStream writer) {
this(new ResultPrinter(writer));
}

/**
 * Constructs a TestRunner using the given ResultPrinter all the output
 */
public TestRunner(ResultPrinter printer) {
fPrinter= printer;
}

/**
 * Runs a suite extracted from a TestCase subclass.
 */
static public void run(Class testClass) {
run(new TestSuite(testClass));
}

/**
 * Runs a single test and collects its results.
 * This method can be used to start a test run
 * from your program.
 * <pre>
 * public static void main (String[] args) {
 *     test.textui.TestRunner.run(suite());
 * }
 * </pre>
 */
static public TestResult run(Test test) {
TestRunner runner= new TestRunner();
return runner.doRun(test);
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
}

/**
 * Runs a single test and waits until the user
 * types RETURN.
 */
static public void runAndWait(Test suite) {
TestRunner aTestRunner= new TestRunner();
aTestRunner.doRun(suite, true);
}

/**
 * Always use the StandardTestSuiteLoader. Overridden from
 * BaseTestRunner.
 */
public TestSuiteLoader getLoader() {
return new StandardTestSuiteLoader();
}

public void testFailed(int status, Test test, Throwable t) {
}

public void testStarted(String testName) {
}

public void testEnded(String testName) {
}

/**
 * Creates the TestResult to be used for the test run.
 */
protected TestResult createTestResult() {
return new TestResult();
}

public TestResult doRun(Test test) {
return doRun(test, false);
}

public TestResult doRun(Test suite, boolean wait) {
TestResult result= createTestResult();
result.addListener(fPrinter);
long startTime= System.currentTimeMillis();
suite.run(result);
long endTime= System.currentTimeMillis();
long runTime= endTime-startTime;
fPrinter.print(result, runTime);

pause(wait);
return result;
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
}

protected void pause(boolean wait) {
if (!wait) return;
fPrinter.printWaitPrompt();
try {
System.in.read();
}
catch(Exception e) {
}
}

public static void main(String args[]) {
TestRunner aTestRunner= new TestRunner();
try {
TestResult r= aTestRunner.start(args);
if (!r.wasSuccessful())
System.exit(FAILURE_EXIT);
System.exit(SUCCESS_EXIT);
} catch(Exception e) {
System.err.println(e.getMessage());
System.exit(EXCEPTION_EXIT);
}
}

/**
 * Starts a test run. Analyzes the command line arguments
 * and runs the given test suite.
 */
protected TestResult start(String args[]) throws Exception {
String testCase= "";
boolean wait= false;

for (int i= 0; i < args.length; i++) {
if (args[i].equals("-wait"))
wait= true;
else if (args[i].equals("-c"))
testCase= extractClassName(args[++i]);
else if (args[i].equals("-v"))
System.err.println("JUnit "+Version.id()+" by Kent Beck and Erich Gamma");
else
testCase= args[i];
}

if (testCase.equals(""))
throw new Exception("Usage: TestRunner [-wait] testCaseName,
                where name is the name of the TestCase class");

try {
Test suite= getTest(testCase);
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
return doRun(suite, wait);
}
catch(Exception e) {
throw new Exception("Could not create and run test suite: "+e);
}
}

protected void runFailed(String message) {
System.err.println(message);
System.exit(FAILURE_EXIT);
}

public void setPrinter(ResultPrinter printer) {
fPrinter= printer;
}



}
public void testFailed(int status, Test test, Throwable t) {
}
```

alternativeClasses(x) is a target concept that uses for learning. Table A.8 shows a target concept and domain theories which are used to construct a detection rule.

Table A.8: Domain theories and a target concept of *Alternative Classes with Different Interfaces* design flaw

A target concepts and domain theories:

R1: $\forall x_i \ (i \in 1,2) \ \mathrm{class}(x_1) \wedge \mathrm{class}(x_2) \wedge \mathrm{doSame}(x_1, x_2)$
$\Rightarrow \mathrm{alternativeClasses}(x_i)$

R2: $\forall x \forall y \ \mathrm{hasMethod}(x, y) \wedge \mathrm{method}(y) \Rightarrow \mathrm{class}(x)$

R3: $\forall y \forall y_{para} \forall y_{type} \forall y_{name} \ \mathrm{hasParameter}(y, y_{para}) \wedge \mathrm{hasReturnType}(y, y_{type}).$
$\mathrm{hasReturnType}(y, y_{name}) \Rightarrow \mathrm{method}(y)$

R4: $\forall_{para} \forall y_{type} \forall y_{name} \ \mathrm{theSamePara}(y_{parax1}, y_{parax2}) \wedge$
$\mathrm{theSameName}(y_{namex1}, y_{namex2}) \wedge \mathrm{theSameType}(y_{typex1}, y_{typex2})$
$\Rightarrow \mathrm{doSame}(x_1, x_2)$

All rules for detecting Alternative Classes with Different Interfaces flaw can be constructed in form of MESs as:

```
%=============================================================================
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
% Rule Alternative Classes with Different Interfaces
alternativeClass(PathProject,ClassID,ClassName,MethodID,MethodName,
    ClassID2,ClassID2Name,MethodID2,MethodName2) :-
    projectRequired(SourceProgram,PathProject,ClassID,ClassName),
    projectRequired(SourceProgram,_,ClassID2,_),
    methodT(MethodID,ClassID,MethodName,ParameterID1,ReturnTypeID1,_,_),
    methodT(MethodID2,ClassID2,MethodName2,ParameterID2,ReturnTypeID2,_,_),
    classT(ClassID2,_,ClassID2Name,_),
    MethodName == MethodName2,
    ClassID \= ClassID2,
    length(ParameterID1,LengthID1) = length(ParameterID1,LengthID2),
    LengthID1 = LengthID2,
    ClassName \= ClassID2Name,
    arg(1,ReturnTypeID1,TypeMethod1),
    arg(1,ReturnTypeID2,TypeMethod2),
    TypeMethod1 = TypeMethod2.

subAtomEqu(ParameterID1,ParameterID2) :-
    arg(1,ParameterID1,basic),
    arg(1,ParameterID2,basic);
    arg(1,ParameterID1,class),
    arg(1,ParameterID2,class);
    arg(1,ParameterID1,generic),
    arg(1,ParameterID2,generic).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFoldersS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.3 The Change Preventers Category

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

The Change Preventers flaws are flaws that hinder changing or further developing the software. These flaws violate rules – suggested by Fowler and Beck – which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, while changes to calculation formulas only affect the other class. Design flaws in this category are Divergent Change, Shotgun Surgery and Parallel Inheritance Hierarchies.

A.3.1 Divergent Change

This flaw shows that a class picks up more responsibilities as it evolves, with no one noticing that two different types of decisions are involved.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class CsvWriter {
    public CsvWriter() {}

    public void write(String[][] lines) {
        for (int i = 0; i < lines.length; i++)
            writeLine(lines[i]);
    }

    private void writeLine(String[] fields) {
        if (fields.length == 0)
            System.out.println();
        else {
            writeField(fields[0]);

            for (int i = 1; i < fields.length; i++) {
                System.out.print(",");
                writeField(fields[i]);
            }
            System.out.println();
        }
    }

    private void writeField(String field) {
        if (field.indexOf(',') != -1 || field.indexOf('\"') != -1)
            writeQuoted(field);
        else
            System.out.print(field);
    }

    private void writeQuoted(String field) {
        System.out.print('\"');
        for (int i = 0; i < field.length(); i++) {
            char c = field.charAt(i);
            if (c == '\"')
                System.out.print("\"\"");
            else
                System.out.print(c);
        }
        System.out.print('\"');
    }
}
```

divergentChange(x) is a target concept that uses for learning. Table A.9 shows a target concept and domain theories which are used to construct a detection rule.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table A.9: Domain theories and a target concept of *Divergent Change* design flaw

A target concepts and domain theories:

R1: $\forall x$ class(x) $\wedge$ inUse(x) $\Rightarrow$ divergentChange(x)

R2: $\forall c_1 \forall a_1$ attribute(a_1) $\wedge$ hasAttribute(c_1,a_1) $\Rightarrow$ class(c_1)

R3: $\forall c_2 \forall m_2$ method(m_2) $\wedge$ hasMethod(c_2,m_2) $\wedge$ hasName(m_2,m_{2-name}) $\Rightarrow$ class(c_2) .

R4: $\forall c_3 \forall m_3$ method(m_3) $\wedge$ hasMethod(c_3,m_3) $\wedge$ hasName(m_3,m_{3-name}) $\Rightarrow$ class(c_3)

R5: $\forall c_1 \forall m_2 \forall m_3 \forall a_1$ invocation(m_2,a_1) $\wedge$ invocation(m_3,a_1) $\wedge$ ($m_{2-name} \neq m_{3-name}$) $\Rightarrow$ inUse(c_1)

All rules for detecting Divergent Change flaw can be constructed in form of MESs as:

```
%==============================================================================
% Rule Divergent Change
divergentChange(ClassID,ClassName,Pathsource) :-
    projectRequired(SourceProgram,Pathsource,ClassID,ClassName),
    divergentChangeRules(ClassID).

divergentChangeRules(ClassID) :-
     divergentChangeInAttribute(ClassID),!;
     divergentChangeInMethod(ClassID),!;
     divergentChangeInAM(ClassID).

 divergentChangeInAttribute(ClassID) :-
     fieldT(FieldID,ClassID,_,_,_),
     methodT(MethodID1,ClassID,_,_,_,_,_),
     methodT(MethodID2,ClassID,_,_,_,_,_),
     getFieldT(_,_,MethodID1,_,_,FieldID),
     getFieldT(_,_,MethodID2,_,_,FieldID),
     MethodID1 \= MethodID2.

divergentChangeInMethod(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    methodT(MethodID1,ClassID,_,_,_,_,_),
    methodT(MethodID2,ClassID,_,_,_,_,_),
    callT(_,_,MethodID1,_,_,_,MethodID),
    callT(_,_,MethodID2,_,_,_,MethodID),
    MethodID1 \= MethodID2.

 divergentChangeInAM(ClassID) :-
     fieldT(FieldID,ClassID,_,_,_),
```

```
     methodT(MethodID,ClassID,_,_,_,_,_),
     methodT(MethodID1,ClassID,_,_,_,_,_),
     getFieldT(_,_,MethodID,_,_,FieldID),
     callT(_,_,MethodID1,_,_,_,MethodID).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.3.2 Shotgun Surgery

Shotgun Surgery flaw occurs when making a simple change requires programmers to change several classes. The cause of this flaw is that one responsibility is split among several classes. There may be a missing class that would understand the whole responsibility (and which would get a cluster of changes). Moreover this can happen through an overzealous attempts to eliminate Divergent Change.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class MsgLog {
    protected static String defaultLogFile = "c:\\msglog.txt";

        public static void write(String s) throws IOException {
        write(defaultLogFile, s);
    }

        public static void write(String f, String s) throws IOException {
        TimeZone tz = TimeZone.getTimeZone("EST"); // or PST, MID, etc ...
        Date now = new Date();
        DateFormat df = new SimpleDateFormat ("yyyy.mm.dd hh:mm:ss ");
        df.setTimeZone(tz);
        String currentTime = df.format(now);

        FileWriter aWriter = new FileWriter(f, true);
        aWriter.write(currentTime + " " + s + "\n");
        aWriter.flush();
        aWriter.close();
    }
}
```

shotgunSurgery(x) is a target concept that uses for learning. Table A.10 shows a target concept and domain theories which are used to construct a detection rule.

Table A.10: Domain theories and a target concept of *Shotgun Surgery* design flaw

A target concepts and domain theories:
R1: $\forall c_1$ class(c_1) $\wedge$ parallelCall(c_1) $\Rightarrow$ shotgunSurgery(c_1)
R2: $\forall c_1 \forall m_1$ method(m_1) $\wedge$ hasMethod(c_1,m_1) $\wedge$ hasName(c_1,c_{1-name}) $\Rightarrow$ class(c_1)
R3: $\forall c_1 \forall c_2 \forall c_3$ calledClass(c_1,c_2) $\wedge$ callingClass(c_1,c_3) $\Rightarrow$ parallelCall(c_1)
R4: $\forall c_2 \forall m_2$ method(m_2) $\wedge$ hasMethod(c_2,m_2) $\wedge$ hasName(c_2,c_{2-name}) $\Rightarrow$ class(c_2)
R5: $\forall c_3 \forall m_3$ method(m_3) $\wedge$ hasMethod(c_3,m_3) $\wedge$ hasName(c_3,c_{3-name}) $\Rightarrow$ class(c_3)
R6: $\forall c_1 \forall c_2 \forall m_1 \forall m_2$ invocation(m_2,m_1) $\wedge$ ($c_{2-name} \neq c_{1-name}$) $\Rightarrow$ calledClass(c_1,c_2)
R7: $\forall c_1 \forall c_3 \forall m_1 \forall m_3$ invocation(m_1,m_3) $\wedge$ ($c_{1-name} \neq c_{3-name}$) $\Rightarrow$ calledClass(c_1,c_3)

All rules for detecting Shotgun Surgery flaw can be constructed in form of MESs as:

```
%=============================================================================
%================ Shotgun Surgery Rules
shotGunSurgery(ClassID,ClassName,Pathsource) :-
    projectRequired(SourceProgram,Pathsource,ClassID,ClassName),
    shotGunSurgeryRules(ClassID).

shotGunSurgeryRules(ClassID) :-
     shotGunSurgeryMMM(ClassID),write('Cast MMM'),!;
     shotGunSurgeryMM2(ClassID),write('Cast MM2'),!;
     shotGunSurgeryMmMA(ClassID),write('Cast MmMA'),!;
     shotGunSurgeryInline(ClassID),write('Cast Inline'),!.

shotGunSurgeryMMM(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    callT(_,_,MethodID,_,_,_,MethodCallingID),
    methodT(MethodCallingID,ClassCallingID,_,_,_,_,_),
    \+methodT(MethodCallingID,ClassID,_,_,_,_,_),
    callT(_,_,MethodCallerID,_,_,_,MethodID),
    methodT(MethodCallerID,ClassCallerID,_,_,_,_,_),
    \+methodT(MethodCallerID,ClassID,_,_,_,_,_).
```

```
shotGunSurgeryMM2(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    callT(_,_,MethodID1,_,_,_,MethodID),
    callT(_,_,MethodID2,_,_,_,MethodID),
    MethodID1 \= MethodID2,
    \+methodT(MethodID1,ClassID,_,_,_,_,_),
    \+getFieldT(_,_,MethodID,_,_,FieldID),
    fieldT(FieldID,ClassID,_,_,_).

shotGunSurgeryMmMA(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    callT(_,_,MethodID1,_,_,_,MethodID),
    callT(_,_,MethodID2,_,_,_,MethodID),
    MethodID1 \= MethodID2,
    \+methodT(MethodID1,ClassID,_,_,_,_,_),
    getFieldT(_,_,MethodID,_,_,FieldID),
    fieldT(FieldID,ClassID,_,_,_).

shotGunSurgeryInline(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    blockT(BlockID,MethodID,_,[ReturnID]),
    returnT(ReturnID,BlockID,_,ExpressID),
    callT(ExpressID,ReturnID,MethodID,_,_,_,MethodCallingID),
    MethodID \= MethodCallingID,
    ClassCallingID \= ClassID,
    methodT(MethodCallingID,ClassCallingID,_,_,_,_,_),
    getFieldT(_,_,MethodCallingID,_,_,FieldCallingID),
    fieldT(FieldCallingID,MethodCallingID,_,_,_).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.3.3 Parallel Inheritance Hierarchies

The hierarchies of structure programs probably grow in parallel. That is a class and its pair being needed at the same time. As usual, it probably is not bad at first but after two or more pairs get introduced. This becomes too complicated structure to change one thing. (Often both classes embody different aspects of the same decision.) This situation introduces Parallel Inheritance Hierarchies flaw in software system.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
class Memento {
   private String state;

   public Memento(String stateToSave) { state = stateToSave; }
   public String getSavedState() { return state; }
}

class Originator {
   private String state;
   /* lots of memory consumptive private data that is not necessary to define the
    * state and should thus not be saved. Hence the small memento object. */

   public void set(String state) {
       System.out.println("Originator: Setting state to "+state);
       this.state = state;
   }

   public Memento saveToMemento() {
       System.out.println("Originator: Saving to Memento.");
       return new Memento(state);
   }
   public void restoreFromMemento(Memento m) {
       state = m.getSavedState();
       System.out.println("Originator: State after restoring from Memento: "+state);
   }
}

class Caretaker {
   private ArrayList<Memento> savedStates = new ArrayList<Memento>();

   public void addMemento(Memento m) { savedStates.add(m); }
   public Memento getMemento(int index) { return savedStates.get(index); }
}

class MementoExample {
   public static void main(String[] args) {
       Caretaker caretaker = new Caretaker();

       Originator originator = new Originator();
       originator.set("State1");
       originator.set("State2");
       caretaker.addMemento( originator.saveToMemento() );
       originator.set("State3");
       caretaker.addMemento( originator.saveToMemento() );
       originator.set("State4");

       originator.restoreFromMemento( caretaker.getMemento(1) );
   }
}
```

ศูนย์วิทยทรัพยากร จุฬาลงกรณ์มหาวิทยาลัย

parallelInheritanceHierarchies(x) is a target concept that uses for learning. Table A.11 shows a target concept and domain theories which are used to construct a detection rule.

Table A.11: Domain theories and a target concept of *Parallel Inheritance Hierarchies* design flaw

A target concepts and domain theories:
R1: $\forall c_1 \forall c_2$ class(c_1) $\wedge$ class(c_2) $\wedge$ parallelInherit(c_1,c_2) $\Rightarrow$ parallelInheritanceHierarchies(c_1)
R2: $\forall c_1 \forall c_2$ inheritOriginate(c_1) $\wedge$ inheritCareTaker(c_2) $\wedge$ chaingParallel(c_1) $\Rightarrow$ parallelInherit(c_1,c_2)
R3: $\forall c_1 \forall a_1$ attribute(a_1) $\wedge$ hasAttribute(c_1,a_1) $\wedge$ hasName(a_1,a_{1-name}) $\Rightarrow$ class(c_1)
R4: $\forall c_2 \forall a_2$ attribute(a_2) $\wedge$ hasAttribute(c_2,a_2) $\wedge$ hasName(a_2,a_{2-name}) $\Rightarrow$ class(c_2)
R5: $\forall c_{11} \forall a_{11}$ attribute(a_{11}) $\wedge$ hasAttribute(c_{11},a_{11}) $\wedge$. hasName(a_{11},$a_{11-name}$) $\Rightarrow$ class(c_{11})
R6: $\forall c_{22} \forall a_{22}$ attribute(a_{22}) $\wedge$ hasAttribute(c_{22},a_{22}) $\wedge$ hasName(a_{22},$a_{22-name}$) $\Rightarrow$ class(c_{22})
R7: $\forall c_1 \forall c_{11}$ inherit((c_1,c_{11})) $\wedge$ (a_{1-name} = $a_{11-name}$) $\Rightarrow$ inheritOriginate(c_1)
R8: $\forall c_2 \forall c_{22}$ inherit((c_2,c_{22})) $\wedge$ (a_{2-name} = $a_{22-name}$) $\Rightarrow$ inheritCareTaker(c_2)
R9: $\forall a_2 \forall c_2$ mutateValue(a_2) $\Rightarrow$ chaingParallel(c_2)
R10: $\forall a_1 \forall a_2$ mutateValue(a_1) $\Rightarrow$ mutateValue(a_2)

All rules for detecting Parallel Inheritance Hierarchies flaw can be constructed in form of MESs as:

```
%=========================================================================
%================ Rules Parallel Inheritance Hierachy
parallelIH(ClassID,ClassName,Pathsource) :-
    projectRequired(SourceProgram,Pathsource,ClassID,ClassName),
    parallelIHRules(ClassID,ClassName).

parallelIHRules(ClassHostID,ClassName) :-
    classT(ClassHostID,_,ClassName,_),
    methodT(MethodHostID,ClassHostID,_,_,_,_,_),
    localT(ObjCareTakerID,_,MethodHostID,_,_,InitialCareTakerID),
    newClassT(InitialCareTakerID,_,MethodHostID,_,_,CareTakerID,_,_),
    localT(ObjOriginatorID,_,MethodHostID,_,_,InitialOriginatorID),
    newClassT(InitialOriginatorID,_,MethodHostID,_,_,OriginatorID,_,_),
    classT(MementoID,_,_,_),
```

```
    classT(OriginatorID,_,_,_),
    classT(CareTakerID,_,_,_),
    fieldT(_,MementoID,_,StateMementoID,_),
    fieldT(StateOrgIDID,OriginatorID,_,StateOrgID,_),
    StateMementoID = StateOrgID,

    % originator Update State normal
    callT(CallMethodOriginatorsetID,_,MethodHostID,_,_,_,MethodOriginatorSetID),
    identT(_,CallMethodOriginatorsetID,_,_,ObjOriginatorID),
    % and in site Set method in Ori
    assignT(AssignID,_,MethodOriginatorSetID,_,_),
    getFieldT(_,AssignID,MethodOriginatorSetID,_,_,StateOrgIDID),

    % caretaker.addMemento [originator.saveToMemento()]
    callT(CallAddMomento,_,MethodHostID,_,_,_,Method_care_addMementorID),
    callT(_,CallAddMomento,MethodHostID,_,_,_,Method_Ori_saveToMementorID),
    identT(_,CallAddMomento,MethodHostID,_,ObjCareTakerID),

    % care taker.addMemento() update state backup
    getFieldT(_,_,Method_care_addMementorID,_,_,FiledUpdateID),
    fieldT(FiledUpdateID,CareTakerID,_,_,_),

    % originator.saveToMemento()
    newClassT(_,_,Method_Ori_saveToMementorID,_,_,MementoID,_,_),!.

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.4 The Dispensables Category

The common thing for the Dispensables category is that they all represent something unnecessary that should be removed from the software system. This group contains two types of flaws (dispensable classes and dispensable code) but since they violate the same principle (programmers can look at them together). If a class does not do enough works, it needs to be removed or its responsibility needs to be increased. This is the case with the Lazy class and the Data class flaws. Code is not used or is redundant that needs to be removed. This is the case with Duplicate Code and Dead Code flaws.

A.4.1 Lazy Class

The Lazy Class is a flaw that a class does not do enough operations its parents, children, or callers seem to be doing all the associated work. There is not enough behavior which left in the class to justify its continued existence.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class AssertionFailedError extends Error {

    public AssertionFailedError () {
    }

    public AssertionFailedError (String message) {
        super (message);
    }
}
```

lazyClass(x) is a target concept that uses for learning. Table A.12 shows a target concept and domain theories which are used to construct a detection rule.

Table A.12: Domain theories and a target concept of *Lazy Class* design flaw

A target concepts and domain theories:
R1: $\forall c_1 \forall c_2$ class(c_1) $\wedge$ class(c_2) $\wedge$ directInheritHierarchy(c_2,c_1) $\Rightarrow$ lazyClass(c_2)
R2: $\forall c_1 \forall m_1$ method(m_1) $\wedge$ hasMethod(c_1,m_1) $\wedge$ attribute(a_1) $\wedge$ hasAttribute(c_1,a_1) $\Rightarrow$ class(c_1)
R3: $\forall c_2 \forall m_2$ method(m_2) $\wedge$ hasMethod(c_2,m_2) $\wedge$ attribute(a_2) $\wedge$ hasAttribute(c_2,a_2) $\Rightarrow$ class(c_2) .
R4: $\forall c_1 \forall c_2 \forall a_1 \forall a_2 \forall m_1 \forall m_2$ hasName(m_1,m_{1-name}) $\wedge$ hasName(m_2,m_{2-name}) $\wedge$ hasName(a_1,a_{1-name}) $\wedge$ hasName(a_2,a_{2-name}) $\wedge$ theSame(m_{1-name},m_{2-name}) $\wedge$ theSame(a_{1-name},a_{2-name}) $\Rightarrow$ directInheritHierarchy(c_2,c_1)

All rules for detecting Lazy Class flaw can be constructed in form of MESs as:

```
%============================================================================
```

```
%================ Rules Lazy Class
lazyClass(ClassID,Name,Pathsource,ClassID) :-
    projectSource(SourceProgram,Name,Pathsource,ClassID),
    extendsT(ClassID,ClassParentID),
    constructorT(_,ClassID,_,_,_),
    \+ fieldT(_,ClassID,_,_,_),
    \+ methodT(_,ClassID,_,_,_,_,_),
    write('1 LazyClass : inherit pure'),nl;

    projectWanted('GanttProject',Name,Pathsource,ClassID),
    classT(ClassInnerID,ClassID,_,_),
    write('2 LazyClass : inner Class'),nl;

    projectWanted('GanttProject',Name,Pathsource,ClassID),
methodT(MethodID,ClassID,_,_,_,_,_),
callT(CallID,_,MethodID,_,_,_,MethodInlineID),
getFieldT(_,CallID,_,_,_,AttriID),
fieldT(AttriID,MethodID,_,_,_),
returnT(_,_,MethodID,CallID),

methodT(MethodInlineID,ClassInlineID,_,_,_,_,_),
    getFieldT(_,_,MethodInlineID,_,_,FieldInlineID),
    fieldT(FieldInlineID,ClassInlineID,_,_,_),
    returnT(_,_,MethodInlineID,_),
    write('3 LazyClass : inline class'),nl,!.

projectSource(NameProject,Name,Pathsource,ClassID) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,Name,_)
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

A.4.2 Data class

The Data class flaw is a flaw that a class consists only of public data members or of simple getting and setting methods. This lets clients depend on the mutability and representation of the class. This is common for classes to begin like this: programmers realize that some data is part of an independent object, so programmers extract it. But objects are about the commonality of behavior. These objects are not developed enough as yet to have much behavior.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class FilterMap implements Serializable {
```

```
    ...
    private String filterName = null;
    public String getFilterName() {
        return (this.filterName);
    }
    public void setFilterName(String filterName) {
        this.filterName = filterName;
    }

    private String servletName = null;
    return (this.servletName);
    }
    public void setServletName(String servletName) {
        this.servletName = servletName;
    }
...
}
```

dataClass(x) is a target concept that uses for learning. Table A.13 shows a target concept and domain theories which are used to construct a detection rule.

Table A.13: Domain theories and a target concept of *Data Class* design flaw

Target concepts and domain theory:
R1: $\forall x$ class(x) $\wedge$ $\neg$not−dataclass(x) $\Rightarrow$ dataclass(x)
R2: $\forall x$ $\forall y$ not−dataclass(y) $\wedge$ $\neg$ is(x,y) $\Rightarrow$ $\neg$not−dataclass(x)
R3: $\forall x$ $\forall y$ hasMethod(x,y) $\wedge$ method−operation(y) $\Rightarrow$ not−dataclass(x)
R4: $\forall x$ $\neg$mutator−method(x) $\wedge$ $\neg$accessor−method(x) $\Rightarrow$ method−operation(x)
R5: $\forall x$ $\forall y$ accessor−method(y) $\wedge$ $\neg$is(x,y) $\Rightarrow$ $\neg$accessor−method(x)) .
R6: $\forall x$ $\forall y$ mutator−method(y) $\wedge$ $\neg$is(x,y) $\Rightarrow$ $\neg$mutator−method(x))
R7: $\forall x$ $\forall y$ $\forall z$ has−attribute(z,y) $\wedge$ has−method(z,x) $\wedge$ method−returntype(*x,[VOID,NULL]*) $\wedge$ method−parameter(x,[{y,_}]) $\Rightarrow$ mutator−method(x)
R8: $\forall x$ $\forall y$ $\forall z$ has−attribute(z,y) $\wedge$ has−method(z,x) $\wedge$ method−returntype(x,[y,_]) $\wedge$ method−parameter(x,[{NULL,NULL}]) $\Rightarrow$ accessor−method(x)

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

All rules for detecting Data Class flaw can be constructed in form of MESs as:

```
%==============================================================================
%================ Rules Data Class
dataClass(ClassID_DataClass,ClassName,PathProject) :-
    projectRequired(SourceProgram,PathProject,ClassID_DataClass,ClassName),
    methodT(_,ClassID_DataClass,_,_,_,_,_),
    fieldT(_,ClassID_DataClass,_,_,_),
    \+notDataClass(ClassID_DataClass),
    write('Data Class in case 1: Set-Get Method');

dataClass1(ClassID_DataClass,ClassName,PathProject) :-
    projectRequired(_,PathProject,ClassID_DataClass,ClassName),
noMethodBeing(ClassID_DataClass),
    fieldT(_,ClassID_DataClass,_,_,_),
    \+methodT(_,ClassID_DataClass,_,_,_,_,_),
    write('Data Class in case 2: No Method Exist')

accessorMethod(AccessorMethodID)  :-
    classT(X,_,_,_),
    fieldT(FieldID,ClassID,_,_,_),
    methodT(AccessorMethodID,ClassID,_,_,_,_,_),
    blockT(A,AccessorMethodID,_,_),
    returnT(ReturnID,_,AccessorMethodID,_),
    getFieldT(_,ReturnID,AccessorMethodID,_,_,FieldID).

mutatorMethod(MutatorMethodID) :-
    classT(X,_,_,_),
    fieldT(FieldID,ClassID,_,_,_),
    methodT(MutatorMethodID,ClassID,_,ParameterList,_,_,_),
    paramT(EachParameter,MutatorMethodID,_,_),
    member(EachParameter,ParameterList),
    assignT(_,_,MutatorMethodID,GetID,IdentID),
    getFieldT(GetID,_,MutatorMethodID,_,_,FieldID),
    identT(IdentID,_,ClassID,_,EachParameter).

notDataClass(NotDataClassID)  :-
    classT(NotDataClassID,_,_,_),
    methodT(MethodInDataClass,NotDataClassID,_,_,_,_,_),
    \+ mutatorMethod(MethodInDataClass),
    \+ accessorMethod(MethodInDataClass).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.4.3 Duplicate Code

Duplicate Code flaws is a code where the same code structure is presented in more than one place. This duplication can be syntactic or semantic. In the manner of duplication, methods do the same thing with a different algorithm. *Forming template methods* by using substitute algorithm are required in detecting this flaw.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++){
        if (people[i].equals ("Don")){ return "Don"; }
        if (people[i].equals ("John")){ return "John"; }
        .....
        if (people[i].equals ("Kent")){ return "Kent"; }
    }
    return "";
 }

 void printOwing()
{
    printBanner();

    //print details
    System.out.println (&quot;name: &quot; + _name);
    System.out.println (&quot;amount &quot; + getOutstanding());
}
```

duplicateCode(x) is a target concept that uses for learning. Table A.14 shows a target concept and domain theories which are used to construct a detection rule.

All rules for detecting Duplicate Code flaw can be constructed in form of MESs as:

```
%==============================================================================
% Divergent Change Rules
duplicateCode(ClassID,ClassName,Pathsource) :-
    projectRequired('CommonCLI',Pathsource,ClassID,ClassName),
    duplicateCodeRules(ClassID).
```

```
duplicateCodeRules(ClassID) :-
    duplicateCodeSwitch(ClassID),!;
    duplicateCodePrintLN(ClassID),!;
    duplicateCodeForIf(ClassID),!,
    duplicateCodeWhileIf(ClassID),!;
    duplicateCodeDoIf(ClassID).

duplicateCodeSwitch(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    switchT(SwitchID,_,MethodID,_,_),
    caseT(CaseID_one,SwitchID,_,_),
    ExecNumberID1 is CaseID_one +1,
    ExecNumberID2 is CaseID_one +2,
    execT(ExecNumberID1,SwitchID,_,_),
    execT(ExecNumberID2,SwitchID,_,_),
    CaseID_two \= CaseID_one,
    caseT(CaseID_two,SwitchID,_,_),
    ExecNumberID1_two is CaseID_two +1,
    ExecNumberID2_two is CaseID_two +2,
    execT(ExecNumberID1_two,SwitchID,_,_),
    execT(ExecNumberID2_two,SwitchID,_,_).

duplicateCodePrintLN(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    execT(ExecID1,_,MethodID,_),
    callT(CallID1,ExecID1,MethodID,_,_,_,PrintLNID),
    getFieldT(_,CallID1,MethodID,_,out,_),
    methodT(PrintLNID,_,println,_,_,_,_),
    ExecID1 \= ExecID2,

    execT(ExecID2,_,MethodID,_),
    callT(CallID2,ExecID2,MethodID,_,_,_,PrintLNID),
    getFieldT(_,CallID2,MethodID,_,out,_),
    methodT(PrintLNID,_,println,_,_,_,_).

duplicateCodeForIf(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    forT(ForLoopID,_,MethodID_,_,_,_,ForBodyID),
    blockT(ForBodyID,ForLoopID,_,_),
```

Table A.14: Domain theories and a target concept of *Duplicate Code* design flaw

Target concepts and domain theory:
R1: $\forall c$ class(c) $\wedge$ duplicatePrintLN(c) $\Rightarrow$ duplicateCode(c)
R2: $\forall c \forall m$ hasMethod(c,m) $\wedge$ method(m) $\Rightarrow$ class(c) .
R3: $\forall c \forall m \forall s_1 \forall s_2$ hasStatementPN(s_1,m) $\wedge$ hasStatementPN(s_2,m) $\Rightarrow$ duplicatePrintLN(c)

```
    ifT(IFID_one,ForBodyID,_,_,_,_),
    ifT(IFID_two,ForBodyID,_,_,_,_),
    IFID_one \= IFID_two.

duplicateCodeWhileIf(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    whileT(WhileLoopID,_,MethodID,_,BlockWhileID),
    blockT(BlockWhileID,WhileLoopID,_,_),
    ifT(IfID_one,BlockWhileID,_,_,_,_),
    ifT(IfID_two,BlockWhileID,_,_,_,_),
    IfID_one \= IfID_two.

duplicateCodeDoIf(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    doWhileT(DoWhileID,_,MethodID,_,BlockDoID),
    blockT(BlockDoID,DoWhileID,_,_),
    ifT(IfID_one,BlockDoID,_,_,_,_),
    ifT(IfID_two,BlockDoID,_,_,_,_),
    IfID_one \= IfID_two.

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.4.4 Dead Code

In software evolution, software requirements are changed or new approaches are introduced without adequate cleanup. Complicated logic results in some combinations of conditions that cannot actually happen. A variable, parameter, field, code fragment, method, or class is not used anywhere. These situations lead to Dead Code flaws in software systems.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class DeadCode{
    public void g(){
        System.out.println("Hello");
        return;
        System.out.println("World!");
    }

    public int f (int x, int y){
        int z=x+y;
```

```
        return x*y;
    }
}
```

deadCode(x) is a target concept that uses for learning. Table A.15 shows a target concept and domain theories which are used to construct a detection rule.

Table A.15: Domain theories and a target concept of *Dead Code* design flaw

A target concepts and domain theories:
R1: $\forall a \forall c$ attribute(a) $\wedge$ unUsed(a,c) $\Rightarrow$ deadCode(a)
R2: $\forall c \forall m$ method(m) $\wedge$ hasMethod(c,m) $\wedge$ $\Rightarrow$ class(c) .
R3: $\forall c \forall a$ class(c) $\wedge$ hasAttribute(c,a) $\Rightarrow$ attribute(a)
R4: $\forall c \forall a \forall m$ callAttribute(m,a) $\Rightarrow$ unUsed(a,c)

All rules for detecting Dead Code flaw can be constructed in form of MESs as:

```
%===============================================================================
% Rules Dead Code
deadCode(ClassID,ClassName,Pathsource) :-
    projectRequired('CommonCLI',Pathsource,ClassID,ClassName),
    deadCodeRules(ClassID).

deadCodeRules(ClassID) :-
    deadCodeBlankImplement(ClassID),!;
    deadCodeRulesMethod(ClassID),!;
    deadCodeRulesAttribute(ClassID).

deadCodeRulesAttribute(ClassID) :-
    fieldT(FieldID,ClassID,_,_,_),
    \+getFieldT(AnyCallID,_,_,_,_,FieldID),!.

deadCodeRulesMethod(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    \+callT(AnyClassID,_,_,_,_,_,ClassID),
    \+callT(_,_,ClassID,_,_,_,AnyClassID),
    \+getFieldT(_,_,ClassID,_,_,AnyFieldID),!.

deadCodeBlankImplement(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    blockT(_,ClassID,_,[]),
    extendsT(ClassID,AnyClassExtendID),
```

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

```
    \+classT(AnyClassExtendID,_,'Object',_),!;
    methodT(MethodID,ClassID,_,_,_,_,_),
    \+blockT(AnyBlockID,ClassID,_,_),
    \+interfaceT(ClassID).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.5 The Couplers Category

This group has four coupling-related flaws – Feature Envy, Inappropriate Intimacy, Message Chains and Middle Man. One design principle that has been around for decades is low coupling. This group has 3 flaws that represent high coupling. Middle Man flaw, on the other hand, represents a problem that might be created when it tries to avoid high coupling with constant delegation. Middle Man is a class that is doing too much simple delegation instead of really contributing to the application.

A.5.1 Feature Envy

Feature Envy flaws is a method of such classes that seems to be focused on manipulating the data of other classes rather than its own.

Examples for defining domain theories: An example (`org.apache.catalina.core.ApplicationFilterFactory` class) is used to define rules for design flaw detection as followed.

```
public final class ApplicationFilterFactory {
    private boolean matchFiltersServlet(FilterMap filterMap, String servletName) {
        if (servletName == null) {
            return false;
        } else {
            if (servletName.equals(
                filterMap.getServletName())){
                return true;
            } else {
                return false;
            }
        }
    }
}
```

featureEnvy(x) is a target concept that uses for learning. Table A.16 shows a target concept and domain theories which are used to construct a detection rule.

Table A.16: Domain theories and a target concept of *Feature Envy* design flaw

A target concepts and domain theories:
R1: $\forall c$ class(c) $\wedge$ manipulatingClass(c) $\Rightarrow$ featureEnvy(c)
R2: $\forall c \forall ma$ method(m) $\wedge$ hasMethod(c,m) $\wedge$ attribute(a) $\wedge$ invocate(m,a) $\Rightarrow$ class(c)
R3: $\forall m \forall a$ hasGetField(m,a) $\wedge$ returnParameter(m,a) $\Rightarrow$ method(m)

All rules for detecting Feature Envy flaw can be constructed in form of MESs as:

```
%==============================================================================
% Rules Feature Envy
fetureEnvy(ClassID,ClassName,Pathsource) :-
    projectRequired('CommonCLI',Pathsource,ClassID,ClassName),
    fetureEnvyRules(ClassID,ClassName).

fetureEnvyRules(ClassID,ClassName) :-
    returnParameterData(ClassID,ClassName);
    returnDataDefineobject(ClassID,ClassName).

returnParameterData(ClassID,ClassName) :-
    classT(ClassID,_,ClassName,_),
    methodT(MethodID,ClassID,_,_,_,_,_),
    paramT(_,MethodID,ParameterList,FieldDataName),
    fieldT(FieldDataID,_,_,FieldDataName,_),
    callT(_,_,MethodID,_,_,_,MethodDataID),
    methodT(MethodDataID,ClassDataID,_,_,_,_,_),
    getFieldT(_,_,MethodDataID,_,_,FieldDataID),
    fieldT(FieldDataID,ClassDataID,_,_,_),
    ClassID \= ClassDataID,
    checkIsClass(ParameterList),!.

returnDataDefineobject(ClassID,ClassName) :-
    classT(ClassID,_,ClassName,_),
    methodT(MethodID,ClassID,_,_,_,_,_),
    localT(LocalID,_,MethodID,_,_,_),
    newClassT(_,LocalID,MethodID,_,_,_,_,_),
    returnT(_,_,MethodID,ExpressID),
    callT(CallID,_,MethodID,_,_,_,MethodDataID),
```

```
    identT(_,CallID,MethodID,_,LocalID),
    methodT(MethodDataID,ClassDataID,_,_,_,_,_),
    fieldT(FieldDataID,ClassDataID,_,_,_),
    getFieldT(_,_,MethodDataID,_,_,FieldDataID),!.

checkIsClass(ParameterList) :-
    atom(ParameterList),
    sub_atom(ParameterList, _, _, _,class).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.5.2 Inappropriate Intimacy

Inappropriate Intimacy flaw means that two classes are coupled tightly to each other. It shows that two classes probably became intertwined a little at a time. One class accesses internal (should-be-private) parts of another class (There also is a related form of Inappropriate Intimacy between a subclass and its parents).

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
class Point2d {
    /* The X and Y coordinates of the point--instance variables */
    private double x;
    private double y;
    public boolean debug; // A trick to help with debugging

    public Point2d (double px, double py) { // Constructor
x = px;
y = py;

debug = false; // turn off debugging
    }

    public Point2d () { // Default constructor
this (0.0, 0.0);        // Invokes 2 parameter Point2D constructor
    }
    // Note that a this() invocation must be the BEGINNING of
    // statement body of constructor

    public Point2d (Point2d pt) { // Another consructor
```

```
x = pt.getX();
y = pt.getY();

// a better method would be to replace the above code with
//    this (pt.getX(), pt.getY());
// especially since the above code does not initialize the
// variable debug.  This way we are reusing code that is already
// working.
    }

    public void dprint (String s) {
// print the debugging string only if the "debug"
// data member is true
if (debug)
    System.out.println("Debug: " + s);
    }

    public void setDebug (boolean b) {
debug = b;
    }

    public void setX(double px) {
dprint ("setX(): Changing value of X from " + x + " to " + px );
x = px;
    }

    public double getX() {
return x;
    }
    ......
}

public class Point2dTest {
  public static void main(String[] args) {
    Point2d r1 = new Point2d(1, 1);
    Point2d r2 = new Point2d(2, 3);
    int u = r1.getX();              // Invoke Rect methods
    r1.debug = false;

    if (r1.getX() > 5))      // Use fields and invoke a method
      System.out.println("(" + r1.getX() + "," + r2.getX()+ ") is inside the union");
    }
    ......
}
```

inappropriateIntimacy(x) is a target concept that uses for learning. Table A.17 shows a target concept and domain theories which are used to construct a detection rule.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table A.17: Domain theories and a target concept of *Inappropriate Intimacy* design flaw

A target concepts and domain theories:

R1: $\forall c_1 \forall c_2$ class(c_1) $\wedge$ class(c_2) $\wedge$ $\neg$likewise(c_1, c_2)
$\wedge$ inappropriateCall(c_1, c_2) $\Rightarrow$ inappropriateIntimacy(c_1, c_2) .
R2: $\forall c_1 \forall m_1$ method(m_1) $\wedge$ hasMethod(c_1, m_1) $\Rightarrow$ class(c_1)
R3: $\forall c_2 \forall a_2$ attribute(a_2) $\wedge$ hasAttribute(c_2, a_2) $\Rightarrow$ class(c_2)
R4: $\forall c_1 \forall c_2 \forall m_1 \forall a_2$ getField(m_1, a_2) $\Rightarrow$ inappropriateCall(c_1, c_2)

All rules for detecting Inappropriate Intimacy flaw can be constructed in form of MESs as:

```
%==============================================================================
% Rules Inappropriate Intimacy
inApproIntimacy(Pathsource,ClassID,ClassName) :-
    projectRequired(SourceProgram,Pathsource,ClassID,ClassName),
    inappropriateInimacy(ClassID).

inappropriateInimacy(ClassID) :-
    inAppropriateGen(ClassID);
    inAppropriateInher(ClassID).

inAppropriateGen(ClassID) :-
    methodT(MethodID,ClassID,_,_,_,_,_),
    fieldT(FieldID,Class2ID,_,_,_),
    ClassID \= Class2ID,
    getFieldT(_,_,ClassID,_,_,FieldID).

inAppropriateInher(ClassID) :-
    extendsT(ClassID,ClassExtendID),
    methodT(MethodID,ClassID,_,_,_,_,_),
    fieldT(FieldID,ClassExtendID,_,_,_),
    getFieldT(_,_,ClassID,_,_,FieldID).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.5.3 Message Chains

Message Chains flaw is a flaw that an object must cooperate with other objects to get things done. So that is a good operation. The problem is that this couples both the objects and the path to get to them. This sort of coupling goes against two maxims of object-oriented programming: *Tell, Don't Ask* and the *Law of Demeter*. Tell, Don't Ask shows that instead of asking for objects so that programmers can manipulate them, programmers simply tell them to do the manipulation for him/her. It is phrased even more clearly in the Law of Demeter: A method should not talk to strangers. It is inferred that it should talk only to itself, its arguments, its own fields, or the objects it creates.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class TestRunner extends BaseTestRunner implements TestRunContext {
    private static final int GAP= 4;
    private static final int HISTORY_LENGTH= 5;

    protected JFrame fFrame;
    private Thread fRunner;
    private TestResult fTestResult;

    private JComboBox fSuiteCombo;
    private ProgressBar fProgressIndicator;
    private DefaultListModel fFailures;
private JLabel fLogo;
private CounterPanel fCounterPanel;
private JButton fRun;
private JButton fQuitButton;
private JButton fRerunButton;
private StatusLine fStatusLine;
private FailureDetailView fFailureView;
private JTabbedPane fTestViewTab;
private JCheckBox fUseLoadingRunner;
    private Vector fTestRunViews= new Vector();
    //view associated with tab in tabbed pane

    private static final String TESTCOLLECTOR_KEY= "TestCollectorClass";
    private static final String FAILUREDETAILVIEW_KEY= "FailureViewClass";

public TestRunner() {
}

public static void main(String[] args) {
new TestRunner().start(args);
```

```
}

    public static void run(Class test) {
        String args[]= { test.getName() };
        main(args);
    }

    protected JComboBox createSuiteCombo() {
        JComboBox combo= new JComboBox();
        combo.setEditable(true);
        combo.setLightWeightPopupEnabled(false);

        combo.getEditor().getEditorComponent().addKeyListener(
            new KeyAdapter() {
                public void keyTyped(KeyEvent e) {
                    textChanged();
                        if (e.getKeyChar() == KeyEvent.VK_ENTER)
                            runSuite();
                    }
}
);
    try {
            loadHistory(combo);
        } catch (IOException e) {
            // fails the first time
        }
        combo.addItemListener(
            new ItemListener() {
                public void itemStateChanged(ItemEvent event) {
                    if (event.getStateChange() == ItemEvent.SELECTED) {
                        textChanged();
                    }
                }
            }
        );
        return combo;
    }
    ......
}
```

messageChains(x) is a target concept that uses for learning. Table A.18 shows a target concept and domain theories which are used to construct a detection rule.

All rules for detecting Message Chains flaw can be constructed in form of MESs as:

```
%==============================================================================
```

Table A.18: Domain theories and a target concept of *Message Chains* design flaw

A target concepts and domain theories:
R1: $\forall c_1 \forall c_2 \forall c_3$ class(c_1) $\wedge$ class(c_2) $\wedge$ class(c_3)
$\wedge$ firstCall(c_1,c_2) $\wedge$ secondCall(c_2,c_3) $\Rightarrow$ messageChains(c_1)
R2: $\forall c_1 \forall m_1$ method(m_1) $\wedge$ hasMethod(c_1,m_1) $\Rightarrow$ class(c_1)
R3: $\forall c_2 \forall m_2$ method(m_2) $\wedge$ hasMethod(c_2,m_2) $\Rightarrow$ class(c_2)
R4: $\forall c_3 \forall a_3$ attribute(a_3) $\wedge$ hasAttribute(c_3,a_3)
$\wedge$ method(m_3) $\wedge$ hasMethod(c_3,m_3) $\Rightarrow$ class(c_3) .
R5: $\forall c_1 \forall c_2$ invocation(c_1,c_2) $\wedge$ $\neg$likewise(c_1,c_2)
$\Rightarrow$ firstCall(c_1,c_2)
R6: $\forall c_2 \forall c_3$ invocation(m_2,m_3) $\wedge$ $\neg$likewise(c_2,c_3)
$\Rightarrow$ secondCall(c_2,c_3)
R7: $\forall c_2 \forall c_3$ getField(m_2,a_3) $\wedge$ $\neg$likewise(c_2,c_3)
$\Rightarrow$ secondCall(c_2,c_3)

```
% Rules Message Chains
classMessageChain(ClassID,ClassName,PathProject,CallID1,CallID2) :-
    projectRequired(SourceProgram,PathProject,ClassID,ClassName),
    methodT(MethodID,ClassID,_,_,_,_,_),
    localT(LocalCallID,_,MethodID,_,_,_),
    callT(CallID1,_,MethodID,CallID2,_,_,_),
    callT(CallID2,_,_,IdentID,_,_,_),
    identT(IdentID,_,_,_,LocalCallID).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    projectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_).
```

A.5.4 Middle Man

Middle Man flaw represents a problem that might be created when trying to avoid high coupling with constant delegation. Middle Man is a class that is doing too much simple delegation instead of really contributing to the application.

Examples for defining domain theories: An example is used to define rules for design flaw detection as followed.

```
public class TestHierarchyRunView implements TestRunView {
    TestSuitePanel fTreeBrowser;
```

```
TestRunContext fTestContext;

public TestHierarchyRunView(TestRunContext context) {
    fTestContext= context;
    fTreeBrowser= new TestSuitePanel();
    fTreeBrowser.getTree().addTreeSelectionListener(
        new TreeSelectionListener() {
            public void valueChanged(TreeSelectionEvent e) {
                testSelected();
            }
        }
    );
}

public void addTab(JTabbedPane pane) {
    Icon treeIcon= TestRunner.getIconResource(getClass(), "icons/hierarchy.gif");
    pane.addTab("Test Hierarchy", treeIcon, fTreeBrowser, "The test hierarchy");
}

public Test getSelectedTest() {
    return fTreeBrowser.getSelectedTest();
}

public void activate() {
    testSelected();
}

public void revealFailure(Test failure) {
    JTree tree= fTreeBrowser.getTree();
    TestTreeModel model= (TestTreeModel)tree.getModel();
    Vector vpath= new Vector();
    int index= model.findTest(failure, (Test)model.getRoot(), vpath);
    if (index >= 0) {
        Object[] path= new Object[vpath.size()+1];
        vpath.copyInto(path);
        Object last= path[vpath.size()-1];
        path[vpath.size()]= model.getChild(last, index);
        TreePath selectionPath= new TreePath(path);
        tree.setSelectionPath(selectionPath);
        tree.makeVisible(selectionPath);
    }
}

public void aboutToStart(Test suite, TestResult result) {
    fTreeBrowser.showTestTree(suite);
    result.addListener(fTreeBrowser);
}

public void runFinished(Test suite, TestResult result) {
    result.removeListener(fTreeBrowser);
```

```
    }

    protected void testSelected() {
        fTestContext.handleTestSelected(getSelectedTest());
    }
}
```

middleMan(x) is a target concept that uses for learning. Table A.19 shows a target concept and domain theories which are used to construct a detection rule.

Table A.19: Domain theories and a target concept of *Middle Man* design flaw

A target concepts and domain theories:
R1: $\forall c_1 \forall c_2 \forall c_3$ class(c_1) $\wedge$ class(c_2) $\wedge$ class(c_3)
$\wedge$ firstCallMM(c_1,c_2) $\wedge$ secondCallMM(c_2,c_3) $\Rightarrow$ middleMan(c_2)
R2: $\forall c_1 \forall m_1$ method(m_1) $\wedge$ hasMethod(c_1,m_1) $\Rightarrow$ class(c_1)
R3: $\forall c_2 \forall m_2$ method(m_2) $\wedge$ hasMethod(c_2,m_2) $\Rightarrow$ class(c_2)
R4: $\forall c_3 \forall a_3$ attribute(a_3) $\wedge$ hasAttribute(c_3,a_3) $\wedge$ method(m_3) .
$\wedge$ hasMethod(c_3,m_3) $\Rightarrow$ class(c_3)
R5: $\forall c_1 \forall c_2$ invocation(c_1,c_2) $\wedge$ $\neg$likewise(c_1,c_2)
$\Rightarrow$ firstCallMM(c_1,c_2)
R6: $\forall c_3 \forall m_2 \forall m_3 \forall a_3$ invocation(m_2,m_3) $\wedge$ getField(m_2,a_3) $\wedge$
return(m_2,a_3) $\wedge$ $\neg$likewise(c_2,c_3) $\Rightarrow$ secondCallMM(c_2,c_3)

All rules for detecting Middle Man flaw can be constructed in form of MESs as:

```
%===============================================================================
% Rules Middle Man
classMiddleMan(ClassID,ClassName,PathProject,MethodName) :-
    projectRequired(SourceProgram,PathProject,ClassID,ClassName),
    methodT(MethodID,ClassID,MethodName,_,_,_,_),
    returnT(_,_,MethodID,ExpressID),
    callT(ExpressID,_,MethodID,Express2ID,_,_,_),
    getFieldT(Express2ID,ExpressID,MethodID,_,_,_).

projectRequired(NameProject,Pathsource,ClassID,ClassName) :-
    ProjectS(ProjectID,NameProject,_,_,_),
    sourceFolderS(SourceID,ProjectID,_),
    fileS(FileID,SourceID,Pathsource),
    compilationUnitT(ClassCompilationID,_,FileID,_,_),
    classT(ClassID,ClassCompilationID,ClassName,_)
```

APPENDIX B

META ELEMENT SPECIFICATIONS

In this appendix, all Meta Element Specifications (MESs) expressed for describing the meta program in this dissertation are shown as followed. The structure and semantics of each MESs type are described in details of their arguments.

B.1 Sourcefolders

Table B.1: packageT

packageT(#id, ‘fullname’)	
arguments	
#id: id	the unique ID assigned to this fact.
‘fullname’: atom	package name of this package declaration, as an atom.

Table B.2: compilationUnitT

compilationUnitT(#id, #package, #fileS, [#import1, ...], [#def1, ...])	
arguments	
#id: id	the unique ID assigned to this fact.
#package: packageT, ‘defaultPackage’	Id of the containing package or ‘defaultPackage’.
#file: fileS, ‘dummyFile’	Id of the file that contains this compilation unit or ‘dummyFile’for Bytecode classes.
[#import1, ...]: importT	List of IDs of import declarations contained in the compilation unit. The order in the list corresponds to the textual order in the file.
[#def1,]: classT	List of IDs of type declarations contained in the compilation unit. The order in the list corresponds to the textual order in the file.

Table B.3: importT

importT(#id, #parent, #import)	
arguments	
#id: id	the unique ID assigned to this fact.
#parent: compilationUnitT	ID of the compilation unit that contains this import declaration.
#import: packageT, classT, methodT, fieldT	ID of the import.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

B.2 Building-Blocks Level

Table B.4: classT

classT(#id, #parent, 'name', [#def1,...])	
Description : Represents the class or interface declaration. Every interface declaration is accompanied by the additional fact: interfaceT(#id). Every annotation declaration is accompanied by an interfaceT(#id) and an annotationT(#id) fact. If it is an abstract class, there will be an additional modifierT(Class, 'abstract') fact.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: execT, compilationUnitT, classT, newClassT	ID of the element that contains this class declaration.
'name': atom	the class name, without a package. In the case of anonymous classes, a globally unique name: ANONYMOUS⟨UN⟩, where ⟨UN⟩ is a unique number.
[#def1,]: annotationMemberT, classT, fieldT, methodT, constructorT, classInitializerT	list of IDs for other facts representing the methods or fields, and inner classes. These fields and methods are the members (not necessarily public!) of the class.

Table B.5: methodT

methodT(#id, #class, 'name', [#param1, ...], TYPE, [#exception1,...], #body)	
Description : Represents the declaration of a method. Static methods have an additional fact "modifierT(ID, 'static')", where ID is the #id value of the method. Note that constructors are represented separately by constructorT facts and static initializers are represented by classInitializerT facts.	
Arguments	
#id: id	the unique ID of this method.
#class: classT	the ID of the class containing this method.
'name': atom	the name of the declared method.
[#param1, . . .]: paramT	the list of IDs of the method parameters.
TYPE: a typeterm	the return type of the method.
[#exception1, . . .]: classT	list of IDs of checked exceptions thrown by this method.
#body: blockT or 'null'	ID of the block containing the method body.

Table B.6: constructorT

constructorT(#id, #classT, [#param1, . . .], [#exception1, . . .], #body)	
Description : Represents the constructor declaration.	
Arguments	
#id: id	the unique ID assigned to this fact.
#class: classT	ID of the parent/declaring class.
[#param1, . . .]: paramT	list of IDs of the method parameters.
[#exception1, . . .]: classT	list of IDs of the thrown exceptions.
#body: blockT, null	ID of the block.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.7: classInitializerT

classInitializerT(#id, #classT, #body)	
Description : Represents the initializer declaration.	
Arguments	
#id: id	the unique ID assigned to this fact.
#class: classT	ID of the parent/declaring class.
#body: blockT	ID of the block.

Table B.8: enumConstantT

enumConstantT(#id, #parent, #encl, 'name', #args)	
Description : Represents the field declaration in enumeration.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: enumT	the id of the parent enumT.
#encl: id	ID of the enclosing MES.
'name': atom	the name of the field.
#args expression	list of arguments for the enum constant.

Table B.9: fieldT

fieldT(#id, #class, TYPE, 'name', #init)	
Description : Represents the field declaration.	
Arguments	
#id: id	the unique ID assigned to this fact.
#class: classT	ID of the enclosing/declaring class.
TYPE: type term	the type of the field.
'name': atom	the name of the field.
#init: expression, null	ID of the initializer of this variable declaration.

Table B.10: paramT

paramT(#id, #parent, TYPE, 'name')	
Description : Represents the program element (method, catch clause, foreach loop) parametrized by this parameter declaration.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: methodT , constructorT , catchT , foreachT	ID of the enclosing MES.
TYPE: type term	the type of the parameter.
'name': atom	the name of the parameter.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.11: typeParamT

typeParamT(#id, #parent, ‘name’, ‘kind’, [#bound1, . . .])	
Description : Represents a type parameter of a generic class. Type parameters are referenced from the rest of the code via type(typevar, #id, arrayDim) terms, where #id is the ID of the referenced typeParamT element.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: classT	the declaring generic class or interface.
‘name’: atom	the name of the type variable.
‘kind’: atom	‘super ’or ‘extends’.
[#bound1,]: classT, typeParamT	

Table B.12: annotationT

annotationT(#id, #parent, #encl, #annotationType, [#keyValue1, . . .])	
Description : Represents an annotation (expression).	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	the ID of the fact that represents the parent of this fact in the prolog AST. Either the ID of the annotated element or membervaluet fact in case of a nested annotation. e.g. Ann2 in Ann(value = 1, ann1 = Ann2(id=1)).
#encl: id	the enclosing annotated declaration, expression or statement.
annotationType: classT	the referenced annotation type.
[#keyValue1, . . .]:memberValueT	list of IDs of the member value pairs.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.13: memberValueT

memberValueT(#id, #parent, #annotationMember, #valueLiteral)	
Description : Represents a member value pair in an annotationT expression. For example queries see the annotationT description.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: annotationT	the parent element. Either the ID of the annotated element or memberValueT fact in case of a nested annotation. e.g. @Ann2 in @Ann(value = 1, ann1 = @Ann2(id=1)).
#annotationMember : annotationMemberT, null	the referenced annotation member.
#valueLiteral: AnnotationExpression, null	

Table B.14: annotationMemberT

annotationMemberT(#id, #parent, TYPE, ‘name’, #default)	
Description : Represents an annotation member construct.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: classT	the parent annotation class attributed with annotationTypeT.
TYPE: type term	the type of the member.
‘name’: atom	the name of the construct.
#default: AnnotationExpression, null	the default expression is optional.

Table B.15: annotatedT

annotatedT(#annotated, #annotation)	
Description : Represents the annotation of a syntax element.	
Arguments	
#annotated: id	the id of the syntax element.
#annotation: annotationT	the annotation.

Table B.16: commentT

commentT(#id, #parent, ‘type’)	
Description : Represents a comment.	
Arguments	
#id: id	the unique ID assigned to this comment.
#parent: id	the ID of the fact that represents the element this comment is referring to. The parent element is determined based on a set of Parent Identification Heuristics.
#type: atom	One of these atoms.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

B.3 Interface-Level Attributes

Table B.17: interfaceT

interfaceT(#class)	
Description : Represents the interface declaration.	
Arguments	
#class:classT	ID of the fact declared to be an interface (this is NOT a unique ID, but a reference to another fact, like a modifierT).

Table B.18: externT

externT(#id)	
Description : It expresses that the class with identity #id is only available as byte-code.	
Arguments	
#id: classT	ID of the byte-code class. This ID is NOT the ID of the externT fact but just a reference to the fact representing the byte-code class.

Table B.19: enumT

enumT(#class)	
Description : Expresses that the class with identity #class is an enum declaration (Enumeration).	
Arguments	
#class: classT	ID of the classT fact declared to be an enumeration (this is NOT the own ID of the enumT fact, but a reference to a classT fact).

Table B.20: annotationTypeT

annotationTypeT(#id)	
Description : Attribute for classT facts. This attribute makes the class an annotation declaration. The class is a subtype (extendsT) of java.lang.annotation.Annotation.	
Arguments	
#id: classT	The ID of the class that is marked as an annotation declaration

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.21: markerAnnotationT

markerAnnotationT(#id)	
Description : An attribute for annotationT facts. Just to maintain the original syntax of annotations without arguments. Annotation is a marker annotation if the parenthesis's are omitted. @marker instead of the normal annotation @marker().	
Arguments	
#id: annotationT	ID of the corresponding annotationT

Table B.22: modifierT

modifierT(#id, 'modifier')	
Description : -	
Arguments	
#id: (classT, fieldT, methodT, constructorT, classInitializerT)	ID form parent MES. Note: modifierT have no own ID. They are referenced over the ID from the corresponding parent
'modifier': atom	one of serveral atoms: public, private, package, protected, static, strictfp, synchronized, transient, native, volatile, abstract and final.

Table B.23: implementsT

implementsT(#class, #interface)	
Description : Represents the implementation of an interface by a class.	
Arguments	
#class: classT	ID of the class.
#interface: classT	ID of an interface implemented by the class.

Table B.24: extendsT

extendsT(#class, #extendedClass)	
Description : Represents the immediate subtype/supertype relation. Transitive super-/subtyping is expressed by the predicate subtype(#subClass, #superClass).	
Arguments	
#class: classT	ID of an object type (class or interface).
#extendedClass: classT	ID of the direct supertype (superclass or super-interface). Note: Interfaces that have no explicit super-interface in the source code have the class java.lang.Object as direct supertype.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.25: assertT

assertT(#id, #parent, #enclMethod, #condition, #msg)	
Description : Represents the assert statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the father node in the AST, typically a block.
#enclMethod: methodT, constructorT, classInitializerT	ID of the enclosing method declaration.
#condition: expression	boolean expression's ID.
#msg: expression, null	ID of the expression that is to be given to assertionerror.

Table B.26: assignT

assignT(#id, #parent, #encl, #lhs, #expr)	
Description : Represents the assignment expression.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	the ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT	the ID of the fact that represents the enclosing element.
#lhs: getFieldT, identT, indexedT	ID of the left hand side of this assignment expression.
#exprt: expression	ID of the right hand side expression

Table B.27: blockT

blockT(#id, #parent, #enclMethod, [#statement1, . . .])	
Description : Represents the block statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#enclMethod: methodT, constructorT, classInitializerT	the ID of the fact that represents the enclosing element.
[#statement1, . . .]: statement	List of the statements in this block.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.28: callT

callT(#id, #parent, #encl, #expr, 'name', [#arg1, ...], #method)	
Description : Represents a method invocation.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	the ID of the fact that represents the parent of this fact in the prolog AST.
#encl: methodT, constructorT, classInitializerT, fieldT	the ID of the fact that represents the enclosing element.
#expr:expression	the ID of the fact representing the receiver expression on which the method is invoked. If the receiver is this the value of #expr is null. In case of static method call #expr is a typeRefT.
'name': atom	the simple (i.e. not fully qualified) name of the called method set into simple quotes.
[#arg1, ...]: expression	list of IDs of other facts representing the arguments of this method invocation.
#method: methodT, constructorT	ID of the methodT or constructorT fact that represents the declaration of the invoked method in the static type of the receiver expression. Because of dynamic binding the method actually invoked at run-time might be another one.

Table B.29: caseT

caseT(#id, #parent, #encl, #label)	
Description : Represents the case statement within the switch statement.	
Arguments	
#id:	the unique ID assigned to this fact.
#parent:	ID of the switch statement being used for the node. (which incidentally is the parent within the AST).
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.
#label: expression, null	ID of the reference to a label. For the default case (default:) this is 'null'.

Table B.30: conditionalT

conditionalT (#id, #parent, #encl, #condition, #thenPart, #elsePart)	
Description : Represents the conditional expression: (condition) ? then : else.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT , classT, annotationMemberT	ID of the fact that represents the enclosing element.
#condition: expression	ID of the expression in this conditional expression.
#thenPart: expression	ID of the "then"part of this conditional expression.
#elsePart: expression	ID of the "else "part of this conditional expression.

Table B.31: doWhileT

doWhileT(#id, #parent, #enclMethod, #condition, #body)	
Description : Represents the do statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.
#condition: expression	ID of the expression of the loop condition of this do statement.
#body statement	ID of the body of this do statement.

Table B.32: execT

execT(#id, #parent, #encl, #expr)	
Description : Represents the execution of an expression in a block. Converts an expression to a statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.
#expr: expression, classT	ID of the expression to be executed. In the case of a local class the ID of that class.

Table B.33: forT

forT(#id, #parent, #encl, [#init1, . . .], #condition, [#step1, . . .], #body)	
Description : Represents the for statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.
[#init1, . . .]: expression, localT	list of IDs of the initializer expressions in this for statement.
#condition: expression, null	ID of the expression in this for statement.
[#step1, . . .]: expression	list of IDs of the update expressions in this for statement.
#body: statement	ID of the body of this for statement.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.34: foreachT

foreachT(#id, #parent, #encl, #param, #expression, #body)	
Description : Represents the for each statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.
#param, paramT	ID of the local parameter of this for statement.
#expression expression	ID of the expression (of type list or array) in this for statement.
#body: statement	ID of the body of this for statement.

Table B.35: getFieldT

getFieldT(#id, #parent, #encl, #receiver, 'name', #field)	
Description : Represents a field access expression (read access and write access).	
Arguments	
#id: id	the unique ID of this field access.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT, annotationMemberT	ID of the fact that represents the enclosing element.
#receiver: expression, typeRefT, 'null'	ID of the expression on which the field is accessed or 'null'for the implicit field access. In case of an access to a static field #expr is a typeRefT.
'name': atom	name of the accessed field.
#field: fieldT, null	ID of the accessed field. null is only valid in case of the "length"field of array types.

Table B.36: identT

identT(#id, #parent, #encl, 'name', #symbol)	
Description : Represents an access to (1) a simple name (local variable or parameter) or (2) this or (3) super or (4) null.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT	ID of the fact that represents the method, constructor, class initializer or field definition that contains this (pseudo-)variable access.
'name': atom	this, super, null or any other legal identifier name
#symbol: 'null', localT, paramT, classT	'null'or ID of the referenced local variable, parameter or class. The access to 'super'references as symbol the ID of the superclass, the access to 'this'references the ID of the class of which 'this'is an instance. The access to the 'null'literal references the symbol 'null'.

Table B.37: indexedT

indexedT(#id, #parent, #encl, #index, #indexed)	
Description : represents the array access expression.	
Arguments	
#id: id	the unique id assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT	ID of the fact that represents the enclosing element.
#index: expression	ID of the index expression of this array access expression.
#indexed: expression	ID of the array expression of this array access expression.

Table B.38: labelT

labelT (#id, #parent, #encl, #body, 'name')	
Description : Represents the labeled statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT	ID of the enclosing element declaration.
#body: statement	ID of the body of this labeled statement.
'name': atom	the name of this label.

Table B.39: literalT

literalT(#id, #parent, #encl, TYPE, 'value')	
Description : Represents the literal node (boolean literal, character literal, number literal, string literal, type literal)	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT, classT, annotationMemberT	ID of the fact that represents the enclosing element.
TYPE: typeterm	type of the literal.
'value': atom	the value of this literal.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.40: localT

localT(#id, #parent, #encl, TYPE, ‘name’, #init)	
Description : Represents local variable declaration.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.
TYPE: typeterm	type of the variable.
‘name’: atom	variable name.
#init: expression, null	ID of the initializer of this variable declaration.

Table B.41: newArrayT

newArrayT(#id, #parent, #encl, [#dim1, . . .], [#elem1, . . .], TYPE)	
Description : Represents the array creation expression.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl:methodT, constructorT, classInitializerT, fieldT, classT, annotationMemberT, annotationT	ID of the fact that represents the enclosing element.
[#dim1, . . .]: expression	list of dimension expressions.
[#elem1, ldots]: expression, annotationExpressionType	list of initial elements of this array.
TYPE: typeterm	type of this array.

Table B.42: newClassT

newClassT (#id, #parent, #encl, #constructor, [#arg1, . . .], #ref, #anonClDef, #encltype)	
Description : Represents the class instance creation expression.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl:methodT, constructorT, classInitializerT, fieldT	ID of the fact that represents the enclosing element.
#constructor: constructorT, null	ID of the constructor invoked by this expression. If the referenced constructor is a anonymous class constructor #constructor is ‘null’.
[#arg1, . . .]: expression	list of argument expressions in this class instance creation expression.
#ref: classT	The class instantiated by this constructor call.
#anonClDef: classT, null	the anonymous class declaration introduced by this class instance creation expression, if it has one.
#encltype: classT, newClassT, null	ID of the inner or member class constructor or ID of the anonymous class.

Table B.43: nopT

nopT(#id, #parent, #encl)	
Description : Represents the no operation node, signifying a lone semicolon.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.

Table B.44: operationT

operationT(#id, #parent, #encl, [#arg1, . . .], 'operatorName', pos)	
Description : Represents the infix expression, postfix expression and prefix expression.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl:methodT, constructorT, classInitializerT, fieldT, classT	ID of the fact that represents the enclosing element. (classT if used in a memberValueT).
[#arg1, . . .]: expression	list of the operands of this expression.
operatorName: atom	the operator of this expression (!,+,-,/,*,?, . . .).
pos: number	postfix expression: 1, infix expression: 0, prefix expression: -1.

Table B.45: precedenceT

precedenceT(#id, #parent, #encl, #expr)	
Description : Represents an expression in parentheses () in the source code.	
Arguments	
#id: id	the unique ID given to this node.
#parent: id	ID if the parent node in the AST.
#encl: methodT, constructorT, classInitializerT, fieldT	ID of the fact that represents the enclosing element.
#expr: expression	ID of the expression within the parenthesis.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.46: returnT

returnT(#id, #parent, #encl, #expr)	
Description : Represents the return statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing method declaration.
#expr: expression, null	ID of the expression of this return statement, or null if there is none.

Table B.47: selectT

selectT(#id, #parent, #encl, ‘name’, ENCLOSING TYPE, #selectedType)	
Description : Represents the access to enclosing instances in inner and anonymous classes.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT, annotationMemberT, annotationT	ID of the method declaration that contains this statement or of the field declaration whose initializer contains this statement.
‘name’: atom	either this or super or class.
selected: typeterm	the enclosing type.

Table B.48: switchT

switchT(#id, #parent, #enclMethod, #expr, [#statement1, . . .])	
Description : Represents the switch statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing method declaration.
#expr: expression	ID of the selection expression (the variable used to switch).
[#statement1, . . .]: statement	list of the statements in the switch. Cases (caseT) are a special kind of statement here, much like labels. The default statement is a case statement with the label ‘null’.

Table B.49: synchronizedT

synchronizedT(#id, #parent, #encl, #lock, #body)	
Description : Represents the synchronized statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing method declaration.
#lock: expression	ID of the expression of this synchronized statement.
#body: blockT	ID of the body of this synchronized statement.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.50: throwT

throwT (#id, #parent, #encl, #expr)	
Description : Represents the throw statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.
#expr: expression	ID of the exception.

Table B.51: tryT

tryT(#id, #parent, #encl, #body, [#catcher1, . . .], #finalizer)	
Description : Represents the try statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing method declaration.
#body: blockT	ID of the body, guarded by the try-catch statement.
[#catcher1, . . .]: catchT	list of the exception catchers.
#finalizer: blockT,null	ID of the block containing the statements of the finally part.

Table B.52: typeCastT

typeCastT(#id, #parent, #encl, TYPE, #expr)	
Description : Represents the cast expression.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT, fieldT, annotationMemberT	ID of the fact that represents the enclosing element.
TYPE: typeterm	the target type
#expr: expression	ID of the expression of this cast expression.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.53: typeTestT

typeTestT(#id, #parent, #encl, #condition, #expression)	
Description : Represents the instance of expression.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the fact that represents the enclosing method.
#condition: typeTerm	right operand of this instanceof expression
#expression: expression	left operand of this instanceof expression

Table B.54: typeRefT

typeRefT(#id, #parent, #encl, #type)	
Description : Represents a (qualified) class name in static method calls or static field access.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: callT, getFieldT	ID of the static method call or field access performed on this type.
#encl: methodT, constructorT, classInitializerT, fieldT, classT, annotationMemberT	ID of the enclosing declaration.
#type: classT	The type on which the static field access or method invocation is performed.

Table B.55: whileT

whileT(#id, #parent, #encl, #condition, #body)	
Description : Represents the while statement.	
Arguments	
#id: id	the unique ID assigned to this fact.
#parent: id	ID of the parent node.
#encl: methodT, constructorT, classInitializerT	ID of the enclosing element.
#condition: expressionType	ID of the expression of this while statement.
#body statementType	ID of the body of this while statement.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

B.4 Body Level Attributes

Table B.56: omitArrayDeclarationT

omitArrayDeclarationT(#id)	
Description :This fact is only used for preserving the original appearance of the source code after transformations. It declares that an array initialization (represented by a newarrayt fact) omits the explicit array instantiation in the original source code.	
Arguments	
#id: newArrayT	ID of the newArrayT element.

Table B.57: inlineDeclarationT

inlineDeclarationT(#firstField, [#otherField1, . . .])	
Description : This fact is only used for preserving the original appearance of the source code after transformations. It declares that multiple variables are declared in a single declaration.	
Arguments	
#firstField: fieldT, localT	ID of the first variable.
[#otherField1, . . .]: fieldT, localT	list of IDs of other variables. These facts are marked with an inlinedT fact.

Table B.58: inlinedT

inlinedT(#id, #reference)	
Description : This fact is only used for preserving the original appearance of the source code after transformations. It declares that the declaration of a variable belongs to an inline declaration.	
Arguments	
#id: fieldT, localT	ID of the variable.
#reference: fieldT, localT	Reference to the first variable in the inline declaration.

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย

Table B.59: variableArgumentT

variableArgumentT(#id)	
Description : This fact marks a parameter in a method as a variable argument. The type of this parameter is always an array.	
#id: paramT	ID of the parameter.

Biography

Name : Sakorn Mekruksavanich

Sex : Male

Date of Birth : April 14,1977

Place of Birth : Lampang

ศูนย์วิทยทรัพยากร
จุฬาลงกรณ์มหาวิทยาลัย