การใช้โครงสร้างต้นไม้วากยสัมพันธ์แบบนามธรรมเพื่อพัฒนาโมเดลการคำนวณมาตรวัดสนับสนุนกระบวนการรีแฟคทอริงโปรแกรมเชิงวัตถุ

โดย

นายธีรภัทร์ เสถียรพงษ์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
วิทยาศาสตรมหาบัณฑิต (วิทยาการคอมพิวเตอร์)
สาขาวิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยธรรมศาสตร์
ปีการศึกษา 2559

ลิขสิทธิ์ของมหาวิทยาลัยธรรมศาสตร์

Ref. code: 25595509035084XRS

การใช้โครงสร้างต้นไม้วากยสัมพันธ์แบบนามธรรมเพื่อพัฒนาโมเดลการคำนวณมาตรวัดสนับสนุนกระบวนการรีแฟคทอริงโปรแกรมเชิงวัตถุ

โดย

นายธีรภัทร์ เสถียรพงษ์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
วิทยาศาสตรมหาบัณฑิต (วิทยาการคอมพิวเตอร์)
สาขาวิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยธรรมศาสตร์
ปีการศึกษา 2559

ลิขสิทธิ์ของมหาวิทยาลัยธรรมศาสตร์

Ref. code: 25595509035084XRS

USING GENERAL ABSTRACT SYNTAX TREE FOR CALCULATING METRICS SUPPORTING OBJECT-ORIENTED CODE REFACTORING

BY

MR THERAPHAT SATHEANPHONG

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE (COMPUTER SCIENCE)
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCE AND TECHNOLOGY
THAMMASAT UNIVERSITY
ACADEMIC YEAR 2016

COPYRIGHT OF THAMMASAT UNIVERSITY

Ref. code: 25595509035084XRS

มหาวิทยาลัยธรรมศาสตร์
คณะวิทยาศาสตร์และเทคโนโลยี

วิทยานิพนธ์

ของ

นายธีรภัทร์ เสถียรพงษ์

เรื่อง

การใช้โครงสร้างต้นไม้วากยสัมพันธ์แบบนามธรรมเพื่อพัฒนาโมเดลการคำนวณมาตรวัดสนับสนุน
กระบวนการรีแฟคทอริงโปรแกรมเชิงวัตถุ

ได้รับการตรวจสอบและอนุมัติ ให้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
วิทยาศาสตรมหาบัณฑิต (วิทยาการคอมพิวเตอร์)

เมื่อ วันที่ 31 กรกฎาคม พ.ศ. 2560

ประธานกรรมการสอบวิทยานิพนธ์ ______________________
(ดร. มนวรรัตน์ ผ่องไพบูลย์)

กรรมการและอาจารย์ที่ปรึกษาวิทยานิพนธ์ ______________________
(ผู้ช่วยศาสตราจารย์ ดร. ทรงศักดิ์ รองวิริยะพานิช)

กรรมการสอบวิทยานิพนธ์ ______________________
(ดร. ทวีทรัพย์ อภิวัฒนาพงศ์)

กรรมการสอบวิทยานิพนธ์ ______________________
(รองศาสตราจารย์ ปกรณ์ เสริมสุข)

คณบดี ______________________
(รองศาสตราจารย์ ปกรณ์ เสริมสุข)

หัวข้อวิทยานิพนธ์	การใช้โครงสร้างต้นไม้วากยสัมพันธ์แบบนามธรรมเพื่อพัฒนาโมเดลการคำนวณมาตรวัดสนับสนุนกระบวนการรีแฟคทอริงโปรแกรมเชิงวัตถุ
ชื่อผู้เขียน	นายธีรภัทร์ เสถียรพงษ์
ชื่อปริญญา	วิทยาศาสตรมหาบัณฑิต (วิทยาการคอมพิวเตอร์)
สาขาวิชา/คณะ/มหาวิทยาลัย	สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยธรรมศาสตร์
อาจารย์ที่ปรึกษาวิทยานิพนธ์	ผู้ช่วยศาสตราจารย์ ดร. ทรงศักดิ์ รองวิริยะพานิช
ปีการศึกษา	2559

บทคัดย่อ

งานวิจัยนี้นำเสนอวิธีการและเครื่องมือ โดยใช้เทคนิคกระบวนการพัฒนาซอฟต์แวร์แบบ MDA (Model-Driven Architecture) เพื่อวิเคราะห์และคำนวณหาค่ากลุ่มมาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา ในส่วนคุณลักษณะย่อย ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ดจากมาตรฐาน ISO 9126 และกลุ่มมาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสมซึ่งประกอบด้วยมาตรวัด NOS PAR NBD VG LCOM CU DU และ PU วิธีการที่เสนอนั้นสามารถรองรับภาษาโปรแกรมเชิงวัตถุหลายภาษา โดยกระบวนการพัฒนาซอฟต์แวร์แบบเอ็มดีเอ(MDA : Model-Driven Architecture) ที่ใช้ในงานวิจัยนี้ เริ่มต้นกระบวนการโดยนำเครื่องมือ Sissy แปลงซอร์สโค้ดโปรแกรมภาษาเชิงวัตถุให้เป็นแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ (GAST : Generalized Abstract Syntax Tree) จากนั้นทำการแปลงเป็นแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม (PDG : Program Dependency Graph) เพื่อให้ได้ข้อมูลที่จำเป็นสำหรับหาค่ามาตรวัดที่ต้องการ สุดท้ายนำแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมมาทำกระบวนการแปลงแบบจำลองกับแบบจำลองเก็บค่ามาตรวัดโปรแกรมเพื่อคำนวณหาค่ามาตรวัดของงานวิจัย โดยกระบวนการแปลงแบบจำลองนั้นได้ใช้ภาษาเอทีแอล (ATL transformation language) มาทำการแปลงแบบจำลองเพื่อให้ได้ผลลัพธ์ที่ต้องการ ขั้นตอนทั้งหมดที่กล่าวมานั้นถูกนำมาพัฒนาเป็นเครื่องมือที่เป็นส่วนเสริมของโปรแกรมอิคลิปส์ (Eclipse plugins)

คำสำคัญ: มาตรวัด, แบบจำลอง, รีแฟคทอริง, ร่องรอยที่ไม่ดี, เอ็มดีเอ

Thesis Title	Using General Abstract Syntax Tree for Calculating Metrics Supporting Object-Oriented Code Refactoring
Author	Mr. Theraphat Satheanphong
Degree	Master of Science (Computer Science)
Major Field/Faculty/University	Computer Science Faculty of Science and Technology Thammasat University
Thesis Advisor	Asst. Prof. Dr. Songsakdi Rongviriyapanish
Academic Years	2016

ABSTRACT

This research proposes a method and a tool based on Model-Driven Architecture (MDA) software development technique.for analyzing and calculating a set of software quality metrics and metrics used for selecting the appropriate refactoring. The set of metrics consists of NOS , PAR , NBD, VG, LCOM, DU and PU. The proposed method can support various object-oriented programming languages. The transformation process from code to metrics model begins by using the Sissy tool to convert source code to an abstract syntax tree using Generalized abstract syntax tree meta-model (GAST). Then it is transformed to a program dependency graph using the PDG Meta-Model developed in this research. Lastly, the obtained program dependency graph is transformed to a metric model by which the set of metrics we proposed in this research can be computed. We use the ATL transformation language to describe the transformation process. Finally, We implemented our proposed method as an Eclipse plugin.

Keywords: Metric, Model, Refactoring, Code Smell

Ref. code: 25595509035084XRS

กิตติกรรมประกาศ

วิทยานิพนธ์เล่มนี้สำเร็จได้เป็นอย่างดี ข้าพเจ้าขอขอบพระคุณผู้ช่วยศาสตราจารย์ ดร. ทรงศักดิ์ รองวิริยะพานิช ผู้ที่เป็นอาจารย์ที่ปรึกษาวิทยานิพนธ์ที่เมตตาสละเวลาอันล้ำค่าคอยชี้แนะแนวทางการทำวิทยานิพนธ์ให้ถูกต้องตามแนวทางทั้งยังติดตามและแก้ไขให้แก่ข้าพเจ้า จากศิษย์ที่ไม่มีความรู้สู่การทำวิจัยที่ดีได้ และขอขอบพระคุณนางสาวพนิตา เมนะเมตร รุ่นพี่ระดับปริญญาเอกที่คอยประคับประคองข้าพเจ้าสำหรับแนวทางการทำงานวิจัยตลอดจนแนะนำทั้งเครื่องมือและวิธีการสำหรับการทำงานวิจัยให้สะดวกมากยิ่งขึ้น

ขอขอบพระคุณพี่เป้และเจ้าหน้าที่ประจำสาขาวิชาวิทยาการคอมพิวเตอร์ มหาวิทยาลัยธรรมศาสตร์ ที่คอยอำนวยความสะดวกสำหรับพื้นที่และความช่วยเหลือในด้านต่างๆทั้งเรื่องการเรียนและการทำงานวิทยานิพนธ์

ขอขอบคุณเพื่อนๆปริญญาโท สาขาวิชาวิทยาการคอมพิวเตอร์ รหัส 55 มหาวิทยาลัยธรรมศาสตร์ ที่คอยช่วยเหลือทั้งด้านการเรียนและการสนับสนุนในด้านต่างๆจนทำให้วิทยานิพนธ์เล่มนี้สำเร็จได้ด้วยดี และขอบคุณเพื่อนชาวปทุมวิไล ห้อง 6/2 รุ่นที่ 94 สำหรับกำลังใจดีๆที่มีให้กันเสมอมา

สุดท้ายนี้ขอขอบพระคุณแม่แอ๋ว แจ๊ก จ๋า พี่หนิงและบุคคลที่ไม่ได้กล่าวถึงในที่นี้ ที่คอยให้กำลังใจ สนับสนุนและช่วยเหลือในด้านต่างๆให้แก่ข้าพเจ้าเสมอมา

นายธีรภัทร์ เสถียรพงษ์

Ref. code: 25595509035084XRS

สารบัญ

หน้า

บทคัดย่อภาษาไทย (1)

บทคัดย่อภาษาอังกฤษ (2)

กิตติกรรมประกาศ (3)

สารบัญตาราง (7)

สารบัญภาพ (8)

บทที่ 1 บทนำ 1

1.1 ความเป็นมาและความสำคัญของปัญหา 1
1.2 วัตถุประสงค์ของการวิจัย 3
1.3 ขอบเขตของการวิจัย 3
1.4 ประโยชน์ที่คาดว่าจะได้รับ 4
1.5 ข้อจำกัดงานวิจัย 4

บทที่ 2 วรรณกรรมและงานวิจัยที่เกี่ยวข้อง 5

2.1 วรรณกรรมที่เกี่ยวข้อง 5

Ref. code: 25595509035084XRS

2.1.1 ร่องรอยที่ไม่ดีประเภทลองเมธอด (Long Method Bad smell) 5
2.1.2 กระบวนการรีแฟคทอริง (Refactoring) โดยเน้นกำจัดร่องรอยที่ไม่ดีประเภทลองเมธอด 5
2.1.3 การวัดซอฟต์แวร์ (Software Measurement) 10
2.1.4 แนวคิดการพัฒนาแบบ MDA 13
2.1.4.1 การแปลงแบบจำลอง (Model Transformation) 15
2.2 งานวิจัยที่เกี่ยวข้อง 19

บทที่ 3 วิธีการวิจัย 32

3.1 ภาพรวมของวิธีการดำเนินงานวิจัย 32
3.1.1 กระบวนการพัฒนาซอฟต์แวร์แบบเอ็มดีเอ (MDA: Model-driven Architecture) ที่สามารถรองรับการคำนวณมาตรวัดสำหรับโปรแกรมเชิงวัตถุ 32
3.1.2 การคัดเลือกมาตรวัดที่เกี่ยวข้องจากงานวิจัยที่ผ่านมา 33
3.1.3 แนวทางการแปลงแบบจำลอง (Model-to-Model Transformation) 38
3.2 วิธีการสร้างเมตาโมเดลให้รองรับกับการคำนวณมาตรวัด 39
3.2.1 เมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ (General Abstract Syntax Tree Meta-model) 39
3.2.2 การออกแบบเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม (Program Dependency Graph Meta model) 43
3.2.3 การออกแบบเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม (Metrics Meta Model) 45
3.3 เครื่องมือที่ใช้พัฒนาโปรแกรม 47
3.3.1 องค์ประกอบทางด้านฮาร์ดแวร์ 47
3.3.2 องค์ประกอบทางด้านซอฟต์แวร์ 47
3.4 ขั้นตอนการพัฒนา 47
3.4.1 ส่วนของการพัฒนาซอฟต์แวร์แบบ MDA (Model-driven Architecture) 47
3.5 วิธีการวัดผล 48

Ref. code: 25595509035084XRS

บทที่ 4 ผลการวิจัยและอภิปรายผล 49

4.1 นิยามกฎการแปลงแบบจำลอง 49

4.1.1 การแปลงซอร์สโค้ดโปรแกรมเชิงวัตถุให้อยู่ในรูปของแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ 52

4.1.2 การสร้างกฎการแปลงแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ไปสู่แบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม 54

4.1.3 การแปลงแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมไปสู่แบบจำลองเก็บค่ามาตรวัดของโปรแกรม 63

4.2 การแสดงผลจากส่วนเสริมโปรแกรมอีคลิปส์ (Eclipse-plugin) 70

4.3 การทดสอบ 71

บทที่ 5 สรุปผลการวิจัยและข้อเสนอแนะ 86

5.1 สรุปผลการวิจัย 86

5.2 ข้อเสนอแนะ 88

รายการอ้างอิง 89

ภาคผนวก

ภาคผนวก ก 115

ประวัติผู้เขียน 127

Ref. code: 25595509035084XRS

สารบัญตาราง

ตารางที่ หน้า

2.1 มาตรวัดโคฮีชั่นในระดับเมธอด 20
2.2 ชุดมาตรวัดของงานวิจัย (Srivisut and Muenchaisri 2007) 22
2.3 มาตรวัดของงานวิจัย (Kaur & Maini , 2016) 26
2.4 ภาพรวมงานวิจัยที่ผ่านมา 29
3.1 แสดงกลุ่มมาตรวัดที่ได้ทำการคัดเลือก 36
3.2 อิลิเม้นของแพคเกจ Statement 40
3.3 แสดงคำอธิบายอิลิเม้นของเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม 45
3.4 อธิบายอีลิเม้นท์ของเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม 46
3.5 รายละเอียดวิธีการวัดผลของงานวิจัย 48
4.1 การจับคู่ระหว่างเมตาโมเดลเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์กับเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม 50
4.2 การจับคู่ระหว่างเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรมกับเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม 51
4.3 สรุปจำนวนกฎการแปลงแบบจำลอง 52
4.4 ผลการทดลองมาตรวัด Number Of Statement In Method (NOS) 74
4.5 ผลการทดลองมาตรวัด Number of parameter in method (PAR) 75
4.6 ผลการทดลองมาตรวัด Nested Block Depth (NBD) 77
4.7 ผลการทดลองมาตรวัด Cyclomatic complexity (VG) 77
4.8 ผลการทดลองมาตรวัด Lack Of Cohesion In Method (LCOM) 84
4.9 ผลการทดลองมาตรวัด CU , PU และ DU 85

Ref. code: 25595509035084XRS

สารบัญภาพ

ภาพที่ หน้า

2.1 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง 6

2.2 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธีเอ็กแทรเมธอด 6

2.3 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง 6

2.4 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Replace Temp With Query 7

2.5 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง 7

2.6 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Replace Method With Method Object 8

2.7 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง 9

2.8 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Decompose Conditional 9

2.9 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง 9

2.10 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Preserve Whole Object 10

2.11 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง 10

2.12 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Introduce Parameter Object 10

2.13 แสดงแนวคิดพื้นฐานของ MDA 14

2.14 ขั้นตอนทำงานของ Model Transformation 16

2.15 แสดงแนวคิดการแปลงแบบจำลองของภาษาเอทีแอล 17

2.16 ตัวอย่างกฏการแปลงแบบจำลองด้วยภาษาเอทีแอล 18

2.17 แสดงตัวอย่างเฮลเปอร์ isFemale() 19

3.1 แสดงภาพรวมของงานวิจัย 33

3.2 แสดงการแปลงแบบจำลองของงานวิจัย 38

3.3 แสดงแพ็คเกจ Statement ของเมทาโมเดลเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ 39

3.4 แสดงภาพรวมของเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ (General abstract syntax tree Meta-model) 42

3.5 เมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม (Program Dependency Graph Meta-model) 44

3.6 เมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม (Metrics Meta-model) 46

4.1 เมธอด statement ที่สร้างจากภาษาจาวา 53

Ref. code: 25595509035084XRS

4.2 แสดงแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ของเมธอด statement 52
4.3 แสดงกฎ ClassToMClass 55
4.4 แสดงการแปลงกฎ MethodToMControlFlowGraph 55
4.5 แสดงการแปลงกฎ ConstructorToMControlFlowGraph 56
4.6 แสดงกฎ StatementToNode 57
4.7 แสดงกฎ SimpleStatementToNode 58
4.8 แสดงกฎ LoopStatementToIterativeNode 59
4.9 แสดงกฎ BranchStatementToConditionNode 60
4.10 แสดงกฎ JumpStatementToNode 61
4.11 แสดงกฎ FormalParamToParam 61
4.12 แสดงกฎ InstantToVar 62
4.13 แสดงกฎ LocalVariableToVar 63
4.14 เฮลเปอร์ getLCOM 64
4.15 เฮลเปอร์ getNumberMethodUseInstant 65
4.16 กฎ MClassToMClass 65
4.17 เฮลเปอร์ getVG 66
4.18 เฮลเปอร์ getNOS 67
4.19 เฮลเปอร์ getNBD 68
4.20 กฎ MMethodToMMethod 68
4.21 กฎ VarToVar 69
4.22 กฎ ParamToParam 69
4.23 ส่วนของการแสดงผลมาตรวัด 70
4.24 เมธอด statement () ในภาษา C++ 72
4.25 เมธอด statement() ในภาษา Java 73
4.26 เมธอด Movie ในภาษา C++ 74
4.27 เมธอด Movie ในภาษา Java 74
4.28 เมธอด maxSongs() ในภาษา C++ 76
4.29 เมธอด maxSongs() ในภาษา Java 76
4.30 คลาส Customer ในภาษา Java 79
4.31 คลาส Customer ในภาษา C++ 80

Ref. code: 25595509035084XRS

4.32 คลาส Movie ในภาษา Java 81
4.33 คลาส Movie ในภาษา C++ 82
4.34 คลาส Rental ในภาษา Java 83
4.35 คลาส Rental ในภาษา C++ 83
4.36 เมธอด getPrice() ในภาษา Java 85
4.37 เมธอด getPrice() ในภาษา C++ 85

Ref. code: 25595509035084XRS

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

กระบวนการบำรุงรักษาซอฟต์แวร์เป็นวิธีการหนึ่งที่ช่วยยืดอายุการใช้งานซอฟต์แวร์ให้มีการทำงานที่ยาวนาน ช่วยลดต้นทุน ลดความผิดพลาดที่อาจเกิดในอนาคตและทำให้ระบบซอฟต์แวร์ทำงานได้อย่างมีประสิทธิภาพมากขึ้น ปัจจัยที่ก่อให้เกิดการบำรุงรักษาซอฟต์แวร์นั้น อาจเกิดจากการพบข้อผิดพลาดของโปรแกรม การเปลี่ยนแปลงเทคโนโลยีใหม่ๆ หรือเกิดจากการปรับเปลี่ยนความต้องการของผู้ใช้งาน ซึ่งเหตุการณ์ต่างๆเหล่านี้อาจจะทิ้งร่องรอยที่ไม่ดี (Bad smell) ไว้ในระบบ เช่น มีซอร์สโค้ดที่มีลักษณะการทำงานที่ซ้ำกันอยู่ในระบบ (Duplicated code) ทำให้เมธอดนั้นมีขนาดยาว (Long method) และทำให้ขนาดของคลาสใหญ่ขึ้น (Large class) ส่งผลทำให้ยากต่อการทำความเข้าใจและการต่อยอดนั่นคือแสดงให้เห็นถึงความสามารถด้านการบำรุงรักษา (Maintainability) เป็นต้น การกำจัดร่องรอยที่ไม่ดีเหล่านี้นิยมใช้เทคนิคการทำรีแฟคทอริง (Refactoring) (Fowler ,1999)

การทำรีแฟคทอริงก็คือการเปลี่ยนแปลงลักษณะโครงสร้างของซอร์สโค้ดเดิมให้มีลักษณะที่ดีขึ้นโดยที่การทำงานของซอฟต์แวร์ยังคงเหมือนเดิม โดยเทคนิครีแฟคทอริงนั้นจะช่วยทำความเข้าใจทั้งในด้านการบำรุงรักษาและการพัฒนาซอฟต์แวร์ในอนาคตอีกทั้งยังช่วยลดลดค่าใช้จ่ายของวงจรการพัฒนาซอฟต์แวร์ (Software life cycle) กระบวนการรีแฟคทอริงมีขั้นตอนที่สำคัญอยู่ 4 ขั้นตอน คือ 1. ขั้นตอนการตรวจจับร่องรอยที่ไม่ดี (Bad smell detection) 2. ขั้นตอนการระบุวิธีการรีแฟคทอริงเพื่อกำจัดร่องรอยที่ไม่ดี (Identify refactoring) 3. ขั้นตอนการใช้งานรีแฟคทอริง (Apply refactoring) และ 4. การประเมินผลหลังจากการทำรีแฟคทอริง (Assessment) โดยแต่ละขั้นตอนการทำรีแฟคทอริงนั้นจำเป็นต้องมีข้อมูลพื้นฐานที่สำคัญ คือ มาตรวัด (Metric) เพื่อช่วยวัดคุณภาพซอฟต์แวร์ เช่น มาตรวัดซีแค (Chidamber & Kemerer ,1992) หรือมาตรวัดที่เสนอโดย ลอเรนซ์และคิดด์ (Lorenz & Kidd ,1995) เป็นต้น ซึ่งงานวิจัยนี้ได้เจาะจงไปที่ขั้นตอนการระบุ ริแฟคทอริงและขั้นตอนการประเมินผลหลังจากการทำรีแฟคทอริงโดยมาตรวัดที่สนใจคือ มาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา (Metrics for Maintainability) ในส่วนคุณลักษณะย่อย (Sub characteristics) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) จากมาตรฐาน ISO 9126 และ มาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม

Ref. code: 25595509035084XRS

(Metrics for refactoring selection) ปัจจุบันมีเครื่องมือที่ใช้คำนวณหามาตรวัดซอฟต์แวร์อยู่หลายเครื่องมือด้วยกันแต่ที่นักวิจัยส่วนใหญ่นิยมใช้กันก็คือ PMD ,Checkstyle (Fontana, Zanoni et al. , 2013) และ JDeodorant (Fokaefs, Tsantalis et al. , 2007) ซึ่งเครื่องมือที่กล่าวมานั้นต่างก็มีความสามารถในการตรวจหาร่องรอยที่ไม่ดีต่างๆ เช่น Duplicated Code Dead Code Large Class Long Method และ Long Parameter List เป็นต้น อย่างไรก็ตามเครื่องมือที่ได้กล่าวมานั้นต่างก็มีข้อบกพร่องที่คล้ายกันคือสนับสนุนซอร์สโค้ดภาษาใดภาษาหนึ่งเท่านั้นทำให้มีความยากลำบากต่อการใช้งาน ในขณะที่ปัจจุบันกระบวนการทำวิศวกรรมย้อนกลับนั้น (Reverse Engineering) ได้มีวิธีการเอ็มดีเอ (MDA: Model-driven Architecture) ที่สามารถนำซอร์สโค้ดมาสร้างแบบจำลองที่เป็นอิสระไม่ขึ้นกับภาษาโปรแกรม

เอ็มดีเอคือกระบวนการสร้างซอฟต์แวร์โดยใช้แบบจำลอง (Software Model) เป็นตัวขับเคลื่อน ซึ่งวิธีดังกล่าวนั้นสามารถแปลงแบบจำลองให้เป็นซอร์สโค้ด (PSM : Platform Specsific Model) และสามารถแปลงซอร์สโค้ดให้อยู่ในรูปแบบของแบบจำลองได้ (PIM : Platform Inependent Model) ซึ่งได้มีผู้ที่สนใจนำเทคนิคเอ็มดีเอ ไปสร้างเป็นเครื่องมือที่ใช้ในการตรวจหาร่องรอยที่ไม่ดี เช่น SISSy (SISSy, 2011) ซึ่งสามารถสแปลงซอร์สโค้ดออกมาให้อยู่ในรูปของโครงสร้างต้นไม้วากยสัมพันธ์ซึ่งเป็นแบบจำลองที่เก็บรายละเอียดต่างๆที่จำเป็นต้องใช้ในการวิเคราะห์หามาตรวัดคุณภาพซอฟต์แวร์และมาตรวัดสำหรับเลือกเงื่อนไขเสนอรีแฟคทอริง

จากปัญหาดังกล่าว งานวิจัยนี้ จึงนำเสนอวิธีการและเครื่องมือที่ใช้ในการวิเคราะห์คำนวณหาค่ามาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา (Metrics for Maintainability) ในส่วนคุณลักษณะย่อย (Sub characteristics) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) จากมาตรฐาน ISO 9126 และ มาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection) วิธีการที่เสนอสามารถรองรับได้หลายภาษาโปรแกรมที่เป็นเชิงวัตถุ โดยใช้กระบวนการพัฒนาซอฟต์แวร์แบบเอ็มดีเอ (MDA : Model-Driven Architecture) ที่นำเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ (GAST : Generalized Abstract Syntax Tree Meta-model) มาสร้างแบบจำลองที่เป็นอิสระจากแพลตฟอร์ม ที่จำเป็นสำหรับการคำนวณหาค่ามาตรวัดดังที่กล่าวเอาไว้ข้างต้น ซึ่งขั้นตอนที่กล่าวจะถูกพัฒนาให้อยู่ในรูปแบบส่วนเสริมของโปรแกรมอีคลิปส์ (Eclipse plugin)

Ref. code: 25595509035084XRS

1.2 วัตถุประสงค์ของการวิจัย

1.2.1 สำรวจและรวบรวมมาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา (Metrics for Maintainability) ในส่วนคุณลักษณะย่อย (*Sub characteristics*) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) จากมาตรฐาน ISO 9126 และ สำรวจ รวบรวมมาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection) จากงานวิจัยต่างๆ

1.2.2 พัฒนาเครื่องมือสำหรับหาค่ามาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา ในส่วนคุณลักษณะย่อย (*Sub characteristics*) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) ตามมาตรฐาน ISO 9126 และ ค่ามาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสมสำหรับกำจัดร่องรอยที่ไม่ดีประเภทลองเมธอด (Metrics for selecting refactoring)

1.2.3 นำเสนอแนวทางสำหรับนักวิจัยที่สนใจในการใช้วิธีการเอ็มดีเอเพื่อพัฒนาโมเดลสำหรับคำนวณค่ามาตรวัดประเภทอื่น

1.3 ขอบเขตของการวิจัย

1.3.1 ครอบคลุมเฉพาะมาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา (Metrics for Maintainability) ในส่วนคุณลักษณะย่อย (*Sub characteristics*) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) ตามมาตรฐาน ISO 9126 และ มาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection) เพื่อกำจัดร่องรอบที่ไม่ดีประเภทลองเมธอดเท่านั้น

1.3.2 โมเดลและเครื่องมือที่พัฒนาขึ้นสามารถคำนวณค่ามาตรวัดที่ใช้สำหรับระบุรีแฟคทอริงเพื่อใช้กำจัดร่องรอยที่ไม่ดีประเภทลองเมธอด และค่ามาตรวัดที่ใช้วัดคุณภาพด้านการบำรุงรักษา (Maintainability) ในส่วนคุณลักษณะย่อย (*Sub characteristics*) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) เท่านั้น และ เครื่องมือใช้ได้กับซอร์สโค้ดเชิงวัตถุเท่านั้น

1.3.3 การทดลองจะครอบคลุมภาษาโปรแกรมเชิงวัตถุ 2 ภาษาคือ JAVA และ C++

Ref. code: 25595509035084XRS

1.4 ประโยชน์ที่คาดว่าจะได้รับ

1.4.1 ทำให้ได้รับเครื่องมือที่ช่วยในการคำนวณหาค่ามาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา (Metrics for Maintainability) ในส่วนคุณลักษณะย่อย (*Sub characteristics*) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) ตามมาตรฐาน ISO 9126

1.4.2 เพื่อเป็นแนวทางสำหรับผู้ที่สนใจวิจัยด้านการคำนวณหาค่ามาตรวัดด้วยแบบจำลอง

1.5 ข้อจำกัดงานวิจัย

1.5.1 เครื่องมือที่พัฒนาสามารถรองรับ Java Version 32 bit

1.5.2 เครื่องมือ Sissy รองรับภาษา Java ได้ถึง Version 5

Ref. code: 25595509035084XRS

บทที่ 2
วรรณกรรมและงานวิจัยที่เกี่ยวข้อง

2.1 วรรณกรรมที่เกี่ยวข้อง

2.1.1 ร่องรอยที่ไม่ดีประเภทลองเมธอด (Long Method Bad Smell)

ร่องรอยที่ไม่ดี (Bad smell) คือ อาการต่างๆที่อยู่ในซอร์สโค้ดโปรแกรมซึ่งจะก่อให้เกิดปัญหาต่างๆของระบบ ร่องรอยที่ไม่ดีนั้นไม่ใช่ข้อผิดพลาดของโปรแกรม (Bugs) แต่เป็นจุดสุ่มเสี่ยงที่ก่อให้เกิดปัญหาในภายหลัง ซึ่งร่องรอยที่ไม่ดีนั้นอาจเกิดจากการออกแบบโมเดลซอฟต์แวร์ที่ไม่ดีหรือโปรแกรมที่มีโครงสร้างการทำงานที่ซับซ้อน ทำความ เข้าใจยาก ส่งผลให้การทำงานของซอฟต์แวร์นั้นไม่มีประสิทธิภาพ โดยร่องรอยที่ไม่ดีแบ่งออกได้ 22 ประเภท (Fowler ,1999)

Long method bad smell คือ เมธอดที่มีขนาดใหญ่ พารามิเตอร์และตัวแปรชั่วคราวที่มาก ส่งผลให้ยากต่อการทำความเข้าใจและการบำรุงรักษา

การกำจัดร่องรอยที่ไม่ดีนั้นจะใช้วิธีการรีแฟคทอริง (Refactoring) มาทำการแก้ไขร่องรอยที่ไม่ดีต่างๆเหล่านี้ เนื่องจากงานวิจัยนี้มุ่งเน้นการกำจัดร่องรอยที่ไม่ดีประเภทลองเมธอด (Long method) ดังนั้น ผู้วิจัยจะกล่าวถึงการทำรีแฟคทอริงประเภทลองเมธอดเท่านั้น

2.1.2 กระบวนการรีแฟคทอริง (Refactoring) โดยเน้นกำจัดร่องรอยที่ไม่ดีประเภทลองเมธอด

รีแฟคทอริง (Refactoring) เป็นกระบวนการปรับปรุงโครงสร้างภายในซอฟต์แวร์โดยไม่มีการเปลี่ยนแปลงต่อพฤติกรรมภายนอก (Behavior) กระบวนการรีแฟคทอริงจะช่วยเพิ่มประสิทธิภาพของซอฟต์แวร์ทั้งความสามารถในด้านการอ่าน (Readability) ความสามารถด้านการบำรุงรักษา (Maintainability) และลดความซับซ้อนของซอร์สโค้ด (Complexity) กระบวนการรีแฟคทอริงนั้นมีทั้งหมด 6 ประเภท 72 วิธี (Fowler ,1999) แต่เนื่องจากงานวิจัยนี้มุ่งเน้นการกำจัดร่องรอยที่ไม่ดีประเภทลองเมธอด (Long method) ดังนั้นกระบวนการรีแฟคทอริงที่เกี่ยวข้องกับการกำจัดร่องรอยที่ไม่ดีประเภทลองเมธอดจะมี 3 ประเภท 6 วิธี โดยสามารถอธิบายได้ดังนี้

1. Composing methods จะเกี่ยวข้องกับเมธอด (Method) ซึ่งอาจมาจากเมธอดที่มีขนาดยาว (Long method) หรือเมธอดอาจมีความซับซ้อน (Complexity) ส่งผลให้เกิดความสับสนและความผิดพลาดของระบบได้ Composing methods นั้นสามารถจัดการด้วยวิธีการรีแฟคทอริงดังนี้

Ref. code: 25595509035084XRS

1.1.Extract Method คือ วิธีการแยกซอร์สโค้ดที่ถูกพิจารณาแล้วว่าควรนำไปสร้างเป็นเมธอดใหม่ โดยเมธอดที่สร้างใหม่นี้จะต้องตั้งชื่อให้สอดคล้องกับลักษณะการทำงานด้วยดังตัวอย่าง

```
void printOwing() {
  printBanner();

  System.out.println ("name:  " + _name);
  System.out.println ("amount " + getOutstanding());
}
```

ภาพที่ 2.1 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + outstanding);

}
```

ภาพที่ 2.2 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธีเอ็กแทรคเมธอด

1.2. Replace Temp With Query คือการนำซอร์สโค้ดที่มีเนื้อหาการประมวลผลแล้วนำผลลัพธ์จัดเก็บไว้รวมอยู่ในตัวแปรชั่วคราวแทนที่ด้วยเมธอดที่สร้างขึ้นใหม่แล้วให้ซอร์สโค้ดส่วนที่ต้องการเรียกใช้ผ่านเมธอดแทนดังตัวอย่าง

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;

}
```

ภาพที่ 2.3 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง

Ref. code: 25595509035084XRS

```
if (basePrice() > 1000)
  return basePrice() * 0.95;
else
  return basePrice() * 0.98;
...
double basePrice() {
  return _quantity * _itemPrice;
}
```

ภาพที่ 2.4 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Replace Temp With Query

1.3. Replace Method With Method Object คือการแทนที่เมธอดที่มีเนื้อหายาวเกินไปแต่ไม่สามารถใช้วิธีเอ็กแทรกเมธอดได้ เนื่องจากในเมธอดนั้นมีการใช้ตัวแปรเฉพาะที่ (Local variable) โดยสามารถแทนที่ได้โดยเปลี่ยนจากเมธอดไปเป็นรูปแบบของวัตถุ (Object) ดังตัวอย่าง

```
class Order {
  //...
  public double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // long computation.
    //...
  }
```

ภาพที่ 2.5 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง

```java
class Order {
  //...
  public double price() {
    return new PriceCalculator(this).compute();
  }
}

class PriceCalculator {
  private double primaryBasePrice;
  private double secondaryBasePrice;
  private double tertiaryBasePrice;

  public PriceCalculator(Order order) {
    // copy relevant information from order object.
    //...
  }

  public double compute() {
    // long computation.
    //...
  }
}
```

ภาพที่ 2.6 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Replace Method With Method Object

2. Simplifying Conditional Expression คือการจัดนิพจน์เงื่อนไขของซอร์สโค้ดที่ซับซ้อน เนื่องจากการแก้ไขทำให้กระทบต่อซอร์สโค้ดส่วนอื่น ดังนั้นการนำกระบวนการรีแฟคทอริงเข้ามาจะช่วยสามารถจัดการได้ง่ายขึ้น โดยกระบวนการรีแฟคทอริงสำหรับประเภท Simplifying Condition Expression มีดังนี้

2.1 Decompose Conditional คือการยุบซอร์สโค้ดที่มีเงื่อนไขยุ่งยากซับซ้อนออกมาเป็นเมธอดแล้วเรียกใช้งานผ่านเมธอด ดังตัวอย่าง

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

ภาพที่ 2.7 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง

```
if (notSummer(date)) {
  charge = winterCharge(quantity);
}
else {
  charge = summerCharge(quantity);
}
```

ภาพที่ 2.8 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Decompose Conditional

3. **Making Method Calls Simpler** คือ การจัดการการเชื่อมต่อของคลาสซึ่งคลาสประกอบไปด้วยการเชื่อมต่อจากคลาสอื่นๆอยู่แล้ว โดยการเชื่อมต่อที่เข้าใจได้ง่ายเป็นหัวใจในการพัฒนาซอฟต์แวร์เชิงวัตถุที่ดี โดยกระบวนการรีแฟคทอริงสำหรับประเภท Making Method Calls Simpler มีดังนี้

3.1. Preserve Whole Object คือการเปลี่ยนจากการส่งค่าอาร์กิวเม้นต์หลายค่าเป็นการส่งวัตถุไปแทนดังตัวอย่าง

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

ภาพที่ 2.9 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง

Ref. code: 25595509035084XRS

```
boolean withinPlan = plan.withinRange(daysTempRange());
```

ภาพที่ 2.10 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Preserve Whole Object

3.2. Introduce Parameter Object คือ การเปลี่ยนกลุ่มของพารามิเตอร์ในเมธอดเป็นวัตถุ ดังตัวอย่าง

```
amountInvoicedIn(Date startDate, Date endDate);
amountReceivedIn(Date startDate, Date endDate);
amountOverdueIn(Date startDate, Date endDate);
```

ภาพที่ 2.11 แสดงตัวอย่างซอร์สโค้ดที่ควรทำรีแฟคทอริง

```
amountInvoicedIn(DateRange dateRange);
amountReceivedIn(DateRange dateRange);
amountOverdueIn(DateRange dateRange);
```

ภาพที่ 2.12 แสดงตัวอย่างซอร์สโค้ดที่ผ่านการรีแฟคทอริงด้วยวิธี Introduce Parameter Object

2.1.3 การวัดซอฟต์แวร์ (Software Measurement)

การวัดซอฟต์แวร์เป็นวิธีการประเมินคุณลักษณะและคุณภาพซอฟต์แวร์โดยแสดงออกมาในรูปแบบของค่าตัวเลขที่ใช้ในการอธิบายความหมายของคุณลักษณะและคุณภาพของซอฟต์แวร์เหล่านั้น (นงเยาว์และคณะ , 2545) การวัดซอฟต์แวร์นั้นสามารถแบ่งได้เป็น 2 วิธีคือ 1. การวัดทางตรง (Direct measurement) เป็นวิธีการวัดเฉพาะคุณลักษณะภายในของสิ่งที่เราสนใจ เช่น การวัดความยาวของซอร์สโค้ดโปรแกรมซึ่งสามารถวัดได้จากการนับจำนวนบรรทัดทั้งหมดของโปรแกรม (Line of code) เป็นต้น

Ref. code: 25595509035084XRS

2. การวัดทางอ้อม (Indirect measurement) เป็นวิธีการที่ใช้วัดคุณลักษณะภายนอก เช่น การวัดความสามารถในการใช้งาน (Usability) ความสามารถในการทดสอบ (Testability)และความสามารถด้านการบำรุงรักษา (Maintainability) เป็นต้น อย่างไรก็ตามการวัดซอฟต์แวร์ทั้ง 2 วิธีนั้น จะต้องใช้มาตรวัดเพื่อวัดคุณลักษณะและคุณภาพของซอฟต์แวร์ที่เราสนใจ โดยมาตรวัดนั้นสามารถแบ่งออกได้ 2 ประเภทคือ

2.1.3.1 มาตรวัดซอฟต์แวร์แบบดั้งเดิม (Traditional Metrics) เช่น มาตรวัดจำนวนบรรทัด (Line of code) หรือ มาตรวัดไซโคลเมติกคอมเพลกซิตี้ (McCabe's cyclomatic complexity measure)

2..1.3.2 มาตรวัดซอฟต์แวร์เชิงวัตถุ (Object oriented software metrics) เช่น มาตรวัดซีเค (CK Metrics) (Chidamber & Kemerer ,1994) หรือมาตรวัดที่นำเสนอโดยลอเรนซ์และคิดด์ (Lorenz & Kidd ,1995)

นอกจากนี้ได้มีมาตรวัดที่น่าสนใจโดยจะขอยกตัวอย่างดังนี้

(1) ชุดมาตรวัดของ Chidamber และทีม (Chidamber & Kemerer ,1994)

ชุดมาตรวัดนี้เป็นมาตรวัดเชิงวัตุที่จะวัดค่าจากคุณสมบัติของคลาสซึ่งประกอบไปด้วย 6 มาตรวัด คือ

- มาตรวัดจำนวนเมธอดต่อคลาส (Weighted method per class) (WMC) คือผลรวมของการคำนวณค่าความซับซ้อนของมาตรวัดไซโคลเมติก(McCabe , Butler ,1989) ทุกเมธอดภายในแต่ละคลาส
- มาตรวัดความรับผิดชอบต่อคลาส (Response for a class) (RFC) คือจำนวนเมธอดที่สามารถตอบสนองกับการเรียกใช้งานที่ได้รับมาจากคลาสอื่น หรือจากเมธอดบางตัวที่อยู่ภายในคลาส
- มาตรวัดระดับความลึกของการสืบทอดคุณสมบัติของคลาส (Depth of inheritance hierarchy) (DIT) คือจำนวนของระดับชั้น (Level) ของการสืบทอดคุณสมบัติของแต่ละคลาสที่พิจารณา
- มาตรวัดจำนวนคลาสลูก (Number of children) (NOC) คือ การนับจำนวนคลาสลูกทั้งหมดที่มีการสืบทอดมาจากคลาสแม่ ซึ่งเป็นคลาสที่พิจารณาอยู่ในขณะนั้น
- มาตรวัดการเข้าคู่กันระหว่างวัตถุ (Coupling between object) (CBO) คือการแสดงความสัมพันธ์ระหว่างวัตถุเมื่อมีการเรียกใช้ตัวแปรหรือเมธอดระหว่างกัน

Ref. code: 25595509035084XRS

- มาตรวัดระดับการขาดการเกาะกันเป็นก้อนของเมธอดภายในคลาส (LCOM : Lack of cohesion of method) การเกาะกันเป็นก้อน (Cohesion) ของคลาสคือ การที่เมธอดนั้นมีความสัมพันธ์กับเมธอดอื่นอย่างไร ซึ่งการที่มีค่าเกาะกันเป็นก้อนสูง แสดงว่าคลาสนั้นมีการออกแบบที่ดี

(2) มาตรวัดที่นำเสนอโดยลอเรนซ์และคิดด์ (Metric proposed by Lorenz and Kidd)

มาตรวัดเชิงวัตถุของลอเรนซ์และคิดด์ (Lorenz & Kidd ,1995) ที่ถูกออกแบบให้วัดค่าโดยตรงจากคลาสซึ่งประกอบไปด้วยมาตรวัดดังนี้

- มาตรวัด Number of public instance method in a class (PIM) คือ จำนวนเมธอดที่ประกาศเป็น public นั้น จะทำให้คลาสอื่นสามารถเข้ามาใช้งานเมธอดนั้นได้
- มาตราวัด Number of instance methods in a class (NIM) คือ จำนวนเมธอดที่ประกาศภายในคลาสนั้น ซึ่งการประกาศเมธอดเป็น public protected และ private
- มาตรวัด Number of instance variable in a class (NIV) คือจำนวนตัวแปรที่ประกาศภายในคลาสนั้น
- มาตรวัด Number of class methods in a class (NCM) คือ จำนวนเมธอดที่เป็นคลาสที่ประกาสภายในคลาสนั้น
- มาตรวัด Number of class variables in a class (NVC) คือ จำนวนตัวแปรที่เป็นคลาสที่ประกาศอยู่ในคลาสนั้น
- มาตรวัด Number of methods overridden (NMO) คือ จำนวนเมธอดที่ทำการ override โดยคลาสลูก
- มาตรวัด Number of methods inherited (NMI) คือ จำนวนเมธอดที่ถูก inherit โดยคลาสลูก
- มาตรวัด Number of methods added (NMA) คือ จำนวนเมธอดที่สร้างขึ้นมาใหม่ภายในคลาสลูก

Ref. code: 25595509035084XRS

- มาตรวัด Specialization index (SIX) คือ มาตรวัดที่เกิดจากการคำนวณจำนวนเมธอดที่ถูก override คูณด้วยจำนวน hierarchy หารด้วยจำนวนเมธอดทั้งหมด
- มาตรวัด Average parameters per method (APPM) คือ อัตราส่วนระหว่างจำนวนพารามิเตอร์ในทุกๆเมธอดกับจำนวนเมธอดทั้งหมด

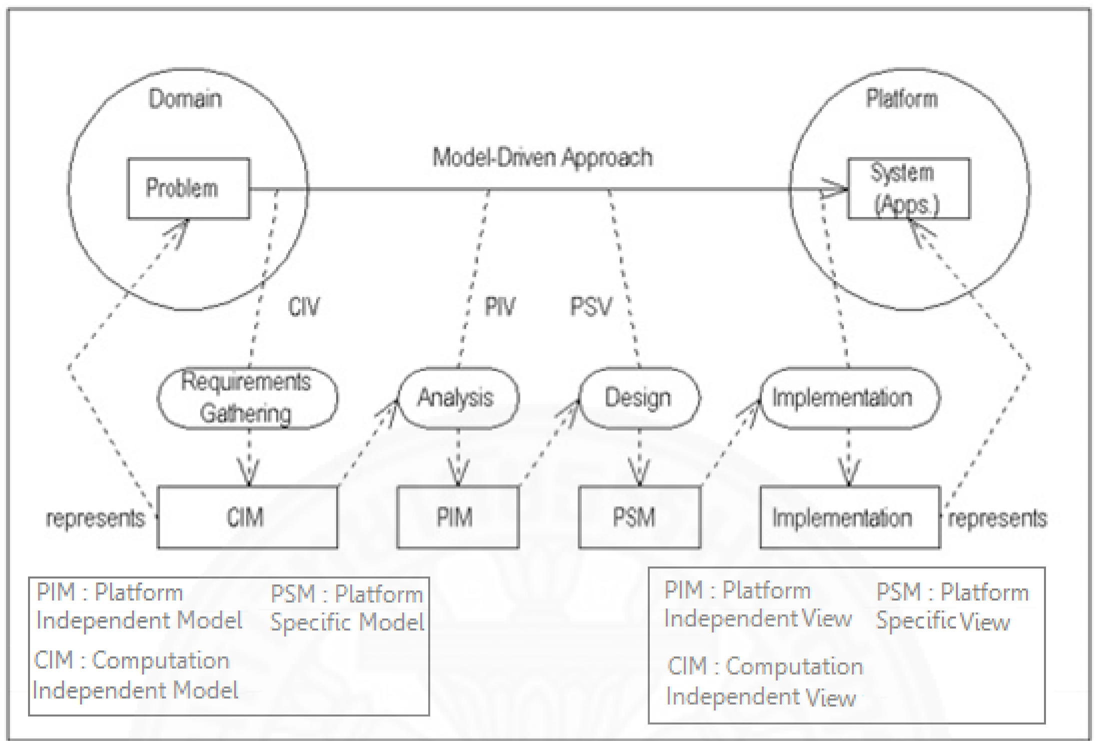
2.1.4 แนวคิดการพัฒนาแบบ MDA

เอ็มดีเอ (MDA : Model-Driven Architecture) คือเฟรมเวิร์ค(Framework) สำหรับกระบวนการพัฒนาซอฟต์แวร์ ถูกกำหนดขึ้นโดยหน่วยงานชื่อ โอเอ็มจี (OMG : Object Management Group)(Miller,2003) หัวใจหลักของเอ็มดีเอคือ โมเดล (Model) หรือแบบจำลองซอฟต์แวร์ที่อยู่ในกระบวนการพัฒนาซอฟต์แวร์ กระบวนการพัฒนาซอฟต์แวร์ด้วยวิธีการเอ็มดีเอนั้นจะขับเคลื่อนโดยการสร้างแบบจำลองการทำงานของระบบซอฟต์แวร์ โดยการแปลงโมเดลอธิบายความต้องการของซอฟต์แวร์ ไปเป็นโมเดลแสดงการทำงานของซอฟต์แวร์และสุดท้ายจะแปลงเป็นซอร์สโค้ดของระบบซอฟต์แวร์

วงจรการพัฒนาซอฟต์แวร์ด้วยวิธีการเอ็มดีเอนั้นไม่ได้แตกต่างจากวงจรการพัฒนาแบบดั้งเดิม กล่าวคือวิธีการของเอ็มดีเอนั้นจะเป็นการสร้างแบบจำลองนั่นเอง และแบบจำลองที่สร้างขึ้นมานั้นจะถูกทำให้คอมพิวเตอร์เข้าใจด้วย โดยเอ็มดีเอ นั้นจะใช้มุมมอง (View point) ของระบบในการสร้างแบบจำลองโดยจะแบ่งออกเป็น 3 มุมมองดังนี้

- Computation Independent Viewpoint หรือ ซีไอวี (CIV) ซึ่งจะเจาะจงไปที่ความต้องการของระบบและสภาพแวดล้อมของระบบซอฟต์แวร์ ซึ่งซีไอวีนั้นจะเป็นมุมมองทางด้านแนวคิด (Concept) เป็นหลัก
- Platform Independent Viewpoint หรือ พีไอวี (PIV) ซึ่งจะมุ่งเน้นไปที่กระบวนการทำงานของระบบ โดยไม่ขึ้นกับแพลตฟอร์มใดๆ
- Platform Specific Viewpoint หรือ พีเอสวี (PSV) ซึ่งจะมุ่งเน้นไปที่กระบวนการของระบบบนแพลตฟอร์มเป้าหมาย โดยมุมมองนี้จะแตกต่างไปตามแพลตฟอร์มเป้าหมาย

Ref. code: 25595509035084XRS



ภาพที่ 2.13 แสดงแนวคิดพื้นฐานของ MDA

จากมุมมองดังกล่าว จะสามารถสร้างแบบจำลองได้ทั้ง 3 ประเภท คือ

(1) Computation Independent Model (CIM)

จากมุมมองซีไอวี จะมีการสร้างแบบจำลองระดับซีไอเอ็มขึ้นมา โดยที่แบบจำลองในระดับซีไอเอ็มจะแสดงรายละเอียดของโดเมน (Domain) และความต้องการของระบบ ซึ่งประกอบด้วยแบบจำลองแสดงรายละเอียดเกี่ยวกับข้อมูลที่ใช้ภายในระบบ (Information Model) โดยแบบจำลองในระดับซีไอเอ็มนั้นจะสอดคล้องกับผลการรวบรวมความต้องการของระบบ (Requirement Gathering) ดังภาพที่ 2.13

(2) Platform Independent Model (PIM)

จากมุมมองพีไอวี จะมีการสร้างแบบจำลองระดับพีไอเอ็มขึ้นมา โดยที่แบบจำลองระดับพีไอเอ็มจะแสดงรายละเอียดกระบวนการทำงานของระบบซึ่งไม่ขึ้นอยู่กับแพลตฟอร์มใดๆ แบบจำลอง

ระดับพีไอเอ็มประกอบด้วยแบบจำลองด้านข้อมูลที่ใช้ในระบบ และแบบจำลองด้านการประมวลผลการทำงานของระบบ (Computation Model) ซึ่งรายละเอียดเกี่ยวกับการประมวลผลของระบบ เป็นอิสระจากแพลตฟอร์มใดๆ โดยโมเดลระดับพีไอเอ็มนั้นจะต้องสอดคล้องกับผลการการวิเคราะห์ระบบ (Analysis) ดังภาพที่ 2.1

(3) Platform Specific Model (PSM)

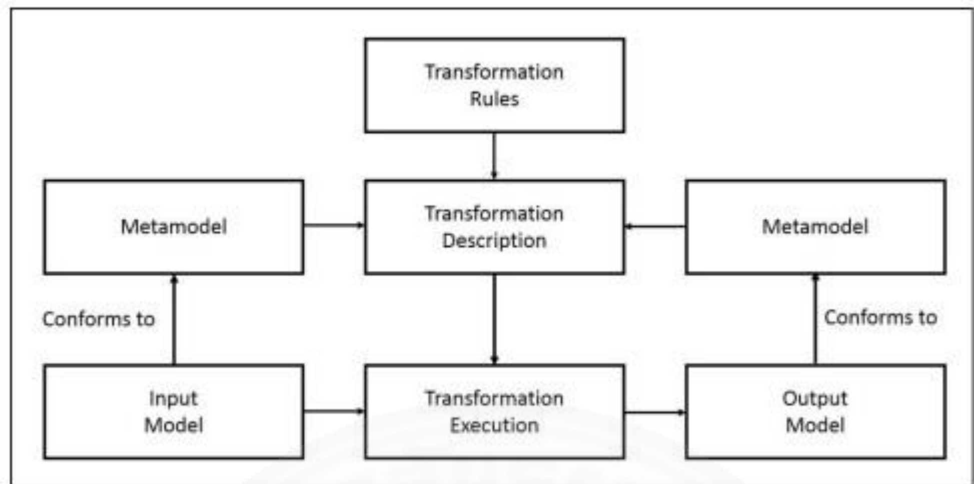
จากมุมมอง พีเอสวีและรายละเอียดของแบบจำลองระดับพีไอเอ็มที่ได้ในขั้นตอนก่อนหน้า จะนำมาออกแบบเพื่อสร้างแบบจำลองระดับพีเอสเอ็ม โดยแบบจำลองระดับพีเอสเอ็มประกอบไปด้วยแบบจำลองด้านข้อมูลของระบบ และแบบจำลองด้านการคำนวณ แบบจำลองระดับพีเอสอ็มจะเจาะจงไปที่แพลตฟอร์มที่กำหนดไว้ โดยแบบจำลองระดับพีเอสเอ็มนั้นจะสอดคล้องกับผลการออกแบบระบบ (Design) ดังรูปที่ 2.1

การที่จะสร้างแบบจำลองระดับพีเอสเอ็มนั้นจะต้องอาศัยกระบวนการหนึ่งของเอ็มดีเอคือขั้นตอนการแปลงแบบจำลองเพื่อให้ได้แบบจำลองตามที่ต้องการ

2.1.4.1 การแปลงแบบจำลอง (Model Transformation)

การแปลงแบบจำลอง (Model Transformation) คือกระบวนการของการแปลงแบบจำลองระดับพีไอเอ็มโดยอาศัยข้อมูลรายละเอียดต่างๆของแบบจำลองเป้าหมายเพื่อให้ได้แบบจำลองระดับพีเอสเอ็มซึ่งขั้นตอนการแปลงแบบจำลองจากพีไอเอ็มไปสู่แบบจำลองระดับพีเอสเอ็มนั้นจะแสดงดังภาพที่ 2.14

Ref. code: 25595509035084XRS



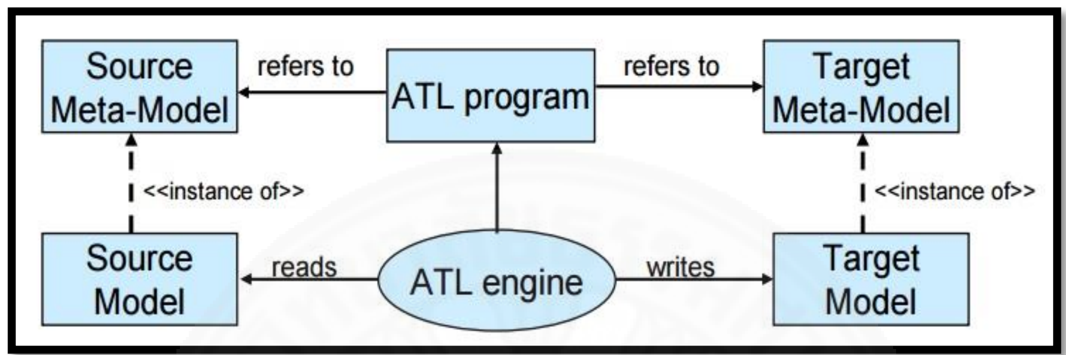
ภาพที่ 2.14 ขั้นตอนทำงานของ Model Transformation

จากภาพที่ 2.14 จะแสดงให้เห็นถึงขั้นตอนการทำ Model Transformation ซึ่งประกอบไปด้วย 3 ส่วนคือ 1. ส่วนของแบบจำลองต้นทาง (Input Model) ในที่นี้คือโมเดลระดับพีไอเอ็มซึ่งมีเมตาโมเดล (Meta-model) ไว้สำหรับอธิบายโครงสร้างของแบบจำลองต้นทางนี้ 2. ส่วนของการแปลงแบบจำลองซึ่งมีกฎการแปลงแบบจำลอง (Transformation Rule) ไว้สำหรับนิยามการแปลงแบบจำลองต้นทางไปเป็นแบบจำลองปลายทางและมีรายละเอียดของการแปลงแบบจำลอง (Transformation Description) ที่ให้ข้อมูลเกี่ยวกับกฎการแปลงแบบจำลองนั้นๆโดย Transformation Execution เป็นการดำเนินการแปลงแบบจำลองต้นทางไปเป็นแบบจำลองปลายทางตามกฎการแปลงแบบจำลอง 3. ส่วนของแบบจำลองปลายทาง (Output Model) ไว้สำหรับอธิบายโครงสร้างของแบบจำลองปลายทางในที่นี้คือโมเดลระดับพีเอสเอ็มซึ่งเป็นผลผลิตจากการทำขั้นตอนทั้ง 2 ก่อนหน้านี้

การทำขั้นตอนการแปลงแบบจำลองนั้นจะต้องใช้ภาษาสำหรับการแปลงแบบจำลอง (Transformation Language) มาดำเนินการในส่วนนี้เช่นภาษาคิววีที (QVT Transformation Language) (OMG , 2005)หรือภาษาเอทีแอล (ATL Transformation Language)(Jouault, Allilaire et al. 2008) ในงานวิจัยนี้เลือกใช้ภาษาเอทีแอลเนื่องจากพัฒนาอยู่บนส่วนเสริมของโปรแกรมอีคลิปส์ซึ่งประกอบไปด้วย Editor, Compiler, Virtual machine และ Debugger อีกทั้งเอทีแอลนั้นยังสนับสนุนการใช้งานทั้ง Declarative และ Imperative ซึ่งขึ้นอยู่กับสถาณการณ์ที่พบต่างจากคิววีทีซึ่งเป็นแบบ Imperative

2.1.4.2 เอทีแอล (Atlas Transformation Language)

เอทีแอล (ATL) (Jouault, Allilaire et al. 2008) เป็นภาษาสำหรับแปลงแบบจำลอง พัฒนาโดย Atlan Mod โดยเอทีแอลนั้นได้นำแนวคิดของการแปลงแบบจำลองมาสร้างเป็นเครื่องมือ โดยแสดงดังภาพที่ 2.15



ภาพที่ 2.15 แสดงแนวคิดการแปลงแบบจำลองของภาษาเอทีแอล

จากภาพที่ 2.15 แสดงแนวคิดการแปลงแบบจำลองของภาษาเอทีแอล ซึ่งมีรายละเอียดดังนี้

- Source Model คือแบบจำลองต้นทางที่ต้องการแปลง
- Source Meta-Model คือเมตาโมเดลที่ใช้สำหรับอธิบายความหมายของแบบจำลองต้นทาง (Source Model)
- Target Model คือแบบจำลองปลายทางที่ต้องการแปลง
- Target Meta-Model คือเมตาโมเดลที่ใช้สำหรับอธิบายความหมายของแบบจำลองปลายทาง
- ATL Program คือการแปลงเซตของแบบจำลองต้นทางไปสู่เซตของแบบจำลองปลายทางซึ่งใน ATL Program จะประกอบไปด้วยกฎ (Rule) ซึ่งใช้กำหนดการแปลง อิลิเม้นของแบบจำลองต้นทางไปสู่แบบจำลองต้นทางอย่างไร และเฮลเปอร์ (Helper) ที่ใช้สำหรับสร้างเงื่อนไขในการแปลงซึ่งจะถูกเรียกใช้โดยกฎการแปลง (Transformation rules)

โดยกฎการแปลงของเอทีแอลนั้นจะแสดงดังภาพที่ 2.16

Ref. code: 25595509035084XRS

```
rule Member2Male {
from
        s: Families!Member (not s.isFemale())
to
        t: Persons!Male (
                fullName <- s.firstName + ' ' + s.familyName()
        )
}
```

ภาพที่ 2.16 ตัวอย่างกฎการแปลงแบบจำลองด้วยภาษาเอทีแอล

จากภาพที่ 2.16 แสดงตัวอย่างกฎการแปลงแบบจำลอง Member2Families ซึ่งเป็นการแปลงเอนทิตี้ Member ของเมตาโมเดล Families ไปสู่เอนทิตี้ Males ของเมตาโมเดล Persons โดยองค์ประกอบของกฎการแปลงนั้นจะขึ้นต้นด้วย rule ซึ่งเป็นการประกาศสร้างกฎการแปลงแบบจำลอง ซึ่งในที่นี้คือ Member2Families โดยในกฎนั้นจะต้องมี from ซึ่งเป็นการประกาศชนิดของเอนทิตี้ต้นทางที่ต้องการแปลงแบบจำลองซึ่งในที่นี้คือ เอนทิตี้ Member ของเมตาโมเดล Families และ to เป็นการประกาศชนิดของเอนทิตี้ปลายทางที่ต้องการแปลงซึ่งในที่นี้คือ เอนทิตี้ Male ของเมตาโมเดล Persons โดยใน to นั้นจะเป็นการแปลงแอตทริบิวต์ fullName ของเอนทิตี้ Male

ภาษาเอทีแอลนั้นนอกจากจะมีกฎการแปลงเพื่อแปลงแบบจำลองต้นทางไปเป็นแบบจำลองปลายทางแล้ว ยังมีเฮลเปอร์ (helper) ซึ่งเปรียบเสมือนฟังก์ชั่นของการแปลงแบบจำลอง ที่ช่วยให้การแปลงแบบจำลองนั้นมีประสิทธิภาพมากยิ่งขึ้นและยังสามารถลดการเขียน code ที่ซับซ้อนและสามารถ re-use ใช้กับกฎการแปลงอื่นๆที่ต้องการใช้งานตามฟังก์ชั่นการทำงานได้อีกด้วย โดยในตัวอย่างกฎ Member2Families มีการเรียกใช้เฮลเปอร์ ชื่อ isFemale() ซึ่งจะทำหน้าที่ในการตรวจสอบสมาชิกที่เป็นเพศหญิงของเอนทิตี้ Member ของเมตาโมเดล Families โดยตัวอย่างของเฮลเปอร์ isFemale() นั้นได้แสดงดังภาพที่ 2.17

Ref. code: 25595509035084XRS

```
helper context Families!Member def: isFemale() : Boolean =
    if not self.familyMother.oclIsUndefined() then
        true
    else
        if not self.familyDaughter.oclIsUndefined() then
            true
        else
            false
```

ภาพที่ 2.17 แสดงตัวอย่างเฮลเปอร์ isFemale()

2.2 งานวิจัยที่เกี่ยวข้อง

งานวิจัย (Meananeatra, Rongviriyapanish et al. 2011) ได้เสนอวิธีการใช้มาตรวัดในการเลือกรีแฟคทอริงเพื่อกำจัดร่องรอยที่ไม่ดีประเภทลองเมธอด โดยแบ่งออกเป็น 4 ขั้นตอน 1. หาค่ามาตรวัดระดับความสามารถในการบำรุงรักษาด้านความสามารถในการวิเคราะห์ (Analysability) จากซอร์สโค้ด โดยใช้มาตรวัด MLOC, MCX ,LCOM และใช้มาตรวัด DU,CU และ PU ที่คำนวณจากกราฟประเภท Control flow graph (CFG) และ Data flow graph (DFG) เพื่อคัดรีแฟคทอริง 2. นำค่ามาตรวัดที่ได้จากขั้นตอนแรกมาเข้าสูตรการคำนวณเพื่อคัดเลือกวิธีรีแฟคทอริง 3. นำวิธีรีแฟคทอริงที่ได้คัดเลือกจากขั้นตอนที่สองนำมาประยุกต์ใช้กับซอร์สโค้ดและคำนวณหาค่า Maintainability 4. ระบุรีแฟคทอริงที่ให้ค่า Maintainability สูงที่สุด

งานวิจัย (Charalampidou, Ampatzoglou et al. 2015) ได้สำรวจและรวบรวมมาตรวัดที่ใช้ทำนายการเกิดร่องรอยที่ไม่ดีประเภทลองเมธอด ซึ่งมาตรวัดดังกล่าวจะเป็นมาตรวัดที่เกี่ยวข้องกับขนาด (Size) และการทำงานร่วมกัน (Cohesion) ดังแสดงในตารางที่ 2.1

Ref. code: 25595509035084XRS

ตารางที่ 2.1

มาตรวัดโคฮีชั่นในระดับเมธอด

Cohesion Metric	Application on method level
LCOM1	LCOM1 = P, where P is the number of pairs of lines that do not share variables.
LCOM2	LCOM2 = P – Q, if P – Q ≥ 0 / otherwise LCOM2 = 0, where P is the number of pairs of lines that do not share variables, and Q is the number of pairs of lines that share variables.
LCOM3	Number of connected components in a graph, where each node represents a line of code and each edge the common use of at least one variable.
LCOM4	Similar to LCOM3. Method calls are treated as edges
LCOM5	LCOM5 = (a - nl) / (l - nl) where n is the number of lines, a is the number of variables used in a line, and l is the total number of variables.
Coh	Coh = 1 – (1 – 1/n) LCOM5 where n is the number of lines
Tight Class Cohesion (TCC)	TCC = NDC / NP where NDC the number of directly connected pairs of lines (i.e. accessing a common variable either within the line or within the body of a method invoked in that line directly or transitively), and NP the maximum possible number of direct connections in a method.
Loose Class Cohesion (LCC)	LCC = (NDC + NIC) / NP where NDC and NP as defined above, and NIC the number of indirectly connected pairs of lines. A pair of lines is indirectly connected, if they access no common variables, but there is a line directly connected to both lines of the pair.
Degree of Cohesion Direct (DC_D)	DC_D =|E_D| / [n * (n – 1) / 2] where E_D the number of edges in a graph connecting directly related lines of code (i.e. as defined for TCC or in cases that the lines directly or transitively invoke the same method), and n the number of lines of a method
Degree of Cohesion Indirect (DC_I)	DC_I =|E_I| / [n * (n – 1) / 2] where EI the number of edges in a graph connecting indirectly related lines of code

	(i.e. as defined for LCC or in cases that the lines directly or transitively invoke the same method), and n the number of lines of a method.
Class Cohesion (CC)	$$CC = \frac{1(n-2)!}{n!} \sum_{i=1}^{\frac{n!}{2(n-2)!}} \frac{\lvert IV\rvert c}{\lvert IV\rvert t}$$ where n the number of lines of a method, $\lvert IV\rvert_t$ is the total number of variables used by two lines and $\lvert IV\rvert_c$ the number of common variables used by both lines
ClassCohesion Metric (SCOM)	$$SCOM = \frac{2}{n(n-1)} \sum_{l=1}^{n=1} \sum_{f=l+1}^{n} \frac{card(Il \cap If) * card(Il \cup If)}{\min\big(card(Il), card(If)\big) * a}$$ where n is the number of lines of a method, $card(Il \cap If)$ = $\lvert IV\rvert_c$ as defined for CC, $card(Il \cup If)$ = $\lvert IV\rvert_t$ as defined for CC, $\min\big(card(Il), card(If)\big)$ is the minimum number of variables accessed between the two lines, and a is the number of variables accessed in the method
Low-level design Similaritybased Class Cohesion (LSCC)	$$LSCC = \int_{\frac{2}{n(n-1)} \sum_{i=1}^{n=1} \sum_{f=l+1}^{n} \cdot}^{0} 1 \quad \begin{matrix} if\ l = 0\ and\ n > 1 \\ if (l > 0\ and\ n = 0) or\ n = 1 \\ otherwise \end{matrix}$$ where n is the number of lines, l the number of variables in the method of interest, and ns the normalized similarity between a pair of lines.

หลังจากคัดเลือกมาตรวัดทั้ง 13 มาตรวัดแล้วได้ทำการสร้างกรณีศึกษาขึ้นมาเพื่อสังเกตการณ์ โดยผลลัพธ์ที่ได้จากการใช้กลุ่มมาตรวัดขนาดเช่น LOC เป็นต้น และกลุ่มมาตรวัดโคฮีชั่นเช่น LCOM1, LCOM2 และ COH เป็นต้น ซึ่งค่าที่ได้รับนั้นโดดเด่นสอดคล้องกับช่วงค่ามาตรวัดตามที่ได้ทำนายเอาไว้ ดังนั้น ผลการศึกษาแสดงให้เห็นว่ามาตรวัดดังกล่าวมีความสามารถในการอธิบายลักษณะความต้องการสำหรับการแก้ปัญหาร่องรอยที่ไม่ดีประเภทลองเมธอด

งานวิจัย (Srivisut & Muenchaisri 2007) ได้เสนอชุดมาตรวัดซอฟต์แวร์สำหรับ Aspect-Oriented (AO) เพื่อระบุร่องรอยที่ไม่ดีที่ซ่อนอยู่ในซอฟต์แวร์ ซึ่งชุดมาตรวัดแสดงดังตารางที่ 2.2

Ref. code: 25595509035084XRS

ตารางที่ 2.2

แสดงชุดมาตรวัดของงานวิจัย (Srivisut ,Muenchaisri 2007)

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

การตรวจสอบมาตรวัดที่นำเสนอมานั้นจะใช้โดยการตรวจจับร่องรอยที่ไม่ดีกับตัวอย่างซอร์สโค้ดด้าน Aspect-Oriented 4 ตัวอย่าง โดยผลการทดสอบนั้นแสดงให้เห็นว่ามาตรวัดดังกล่าวสามารถระบุร่องรอยที่ไม่ดีที่ซ่อนอยู่ในซอฟต์แวร์ได้

Ref. code: 25595509035084XRS

งานวิจัย (Rani & Kaur ,2014) ได้สร้างเครื่องมือสำหรับตรวจสอบร่องรอยที่ไม่ดีชนิด Lazy Class, Long method, Comment lines และ Large class ซึ่งมาตรวัดที่ใช้ในเครื่องมือนี้ประกอบไปด้วย DIT Coupling WCM มาตรวัด Halstead Cyclomatic complexity NLOC และ LOC

งานวิจัย (Sreenu & Rao , 2014) เสนอวิธีการรีแฟคทอริงและมาตรวัดสำหรับร่องรอยที่ไม่ดีชนิด Lazy Class และ Temporary Field ซึ่งมาตรวัดนั้นประกอบไปด้วย Number of method (NOM) ,Instance Variables per Method in Class (IVMC) Depth of inherit (DIT) และ LOC ซึ่งค่าที่ได้จากมาตรวัดเหล่านี้สามารถนำไปใช้กับรีแฟคทอริงดังกล่าวโดยสามารถลดจำนวนของบรรทัด (LOC) และเพิ่มประสิทธิภาพของซอร์สโค้ด

งานวิจัย (Higo, Kamiya et al. , 2005)ได้เสนอวิธีการสำหรับผสานโค้ดโคลน (Code clone) ซึ่งวิธีการที่สำคัญคือการวัดโดยใช้มาตรวัด โดยแบ่งออกเป็น 2 เทคนิค คือ Locational Relationship in Class Hierarchy ซึ่งเป็นเทคนิคการรวมโค้ดโคลนโดยขึ้นอยู่กับที่ตั้งของลำดับความสัมพันธ์ในคลาสโดยได้แบ่งเป็น 3 กรณีได้แก่ 1. ถ้าโค้ดโคลนอยู่ในคลาสเดียวกันให้ extract ออกมาเป็นเมธอดใหม่ที่อยู่ในคลาสเดียวกัน 2. ถ้าโค้ดโคลนมีมากกว่า 2 คลาสโดยที่สืบทอดมาจากคลาสแม่เดียวกัน ให้ทำการดึงโค้ดโคลนไปไว้ที่คลาสแม่ 3. ถ้าโค้ดโคลนปรากฏในคลาสที่แตกต่างกันให้นำโคล้ดโคลนสร้างเป็นคลาสขึ้นมาใหม่ ซึ่งเทคนิคนี้ได้ใช้มาตรวัด DCH (Dispersion in the Class Hierarchy) เทคนิค Coupling between a Code Clone and its Surrounding Code เป็นเทคนิคการย้ายโค้ดที่ซ้ำซ้อนไปยังพื้นที่อื่น โดยใช้มาตรวัด 2 ชนิดในการคำนวณสำหรับการเคลื่อนย้ายโค้ดที่ซ้ำซ้อน คือ NRV (Number of Referenced Variables) และ NAV (Number of Assigned Variables) จากนั้นได้พัฒนาเครื่องมือที่ชื่อว่า ARIES เพื่อใช้ในกระบวนการรีแฟคทอริง หลังจากนั้นได้ทำการทดลองกับ Apache Ant 1.6.0 ซึ่ง ARIES นั้นให้ผลลัพธ์ที่น่าพอใจ สามารถสนับสนุนกระบวนการบำรุงรักษาซอฟต์แวร์อย่างมีประสิทธิภาพ

งานวิจัย (Zhao & Hayes ,2006) เสนอวิธีการทำนายคลาสที่ต้องการทำรีแฟคทอริง วิธีการคือใช้กลุ่มมาตรวัดความซับซ้อนและขนาด ซึ่งประกอบไปด้วย มาตรวัด Halstead เพื่อใช้หาค่าจาก operators และ operands ที่อยู่ในโค้ด และมาตรวัด WMC (Weight Methods Per Class) วิธีการทำนายนี้จะใช้การพิจารณาค่าที่ได้จากการคำนวณที่ได้จากมาตรวัด วิธีการทดลองโดยเปรียบเทียบกับเครื่อง มืออื่นๆ พบว่าเครื่องมือช่วยรีแฟคทอริงนี้สามารถช่วยสนับสนุนให้กับนักพัฒนาวอฟต์แวร์เป็นอย่างดี

งานวิจัย (Higo, Kusumoto et al. , 2008) ได้เสนอเทคนิคในการระบุรีแฟคทอริงในโปรแกรมภาษาเชิงวัตถุโดยใช้มาตรวัด ซึ่งมีวิธีการคล้ายกับงานวิจัยก่อนหน้านี้ในส่วนของการตรวจสอบโค้ด

Ref. code: 25595509035084XRS

โคลน (Higo, Kusumoto et al. 2008) อย่างไรก็ตาม เทคนิคการรวมโค้ดนั้นได้รับการแนะนำว่าให้ทำผ่านขั้นตอนการสกัดและการขั้นตอนการตรวจจับ ในส่วนของการวัดนั้นจะเป็นการใช้มาตรวัด DCH, NRV และ NAV ของงานวิจัย (Higo, Kusumoto et al. , 2008) เพื่อเป็นการแนะนำการรีแฟคทอริ่งที่เหมาะสมด้วยมาตรวัดเหล่านี้ ด้วยวิธีการนี้สามารถระบุการรีแฟคทอริงเช่น Super class, extract class, extract method, form template method, pull up method, move method, pull up constructor และ parameterize method เป็นต้น การรวมโค้ดโคลนนั้นได้ถูกดำเนินการในเครื่องมือ Aries ซึ่งใช้มาตรวัดในการระบุบางรีแฟคทอริงได้ แต่ไม่แนะนำให้ดำเนินการทำรีแฟคทอริง โดยจากการทดลองเทคนิคนี้กับ Apache Ant 1.6.0 นั้นผลลัพธ์ที่ได้สามารถรวมโค้ดโคลนได้อย่างมีประสิทธิภาพ

งานวิจัย (Singh & Kahlon , 2012) ได้ทำการทดลองนำมาตรวัด คือ NOC,DIT,LCOM,LCOM4,WMC,Puf,EncF,CBO,RFC,NOD เพื่อนำมาช่วยหาคลาสที่เกิดข้อผิดพลาดและค้นหาร่องรอยที่ไม่ดีของโปรแกรม โดยได้แบ่งเป็น 2 ขั้นตอน ขั้นตอนแรก ทดลองว่ามาตรวัดซอฟต์แวร์นั้นสามารถที่จะทำนายได้ว่าเกิดร่องรอยที่ไม่ดีและมีข้อผิดพลาดเกิดขึ้นภายในซอร์สโค้ด ขั้นตอนที่สองได้เจาะจงไปที่ว่ามาตรวัดซอฟต์แวร์สามารถที่จะทำนายร่องรอยที่ไม่ดีและข้อผิดพลาดของคลาสและความน่าจะเป็นภายใต้หมวดหมู่ทั้ง 6 ระดับที่ถูกกำหนดโดย Mäntylä (Mäntylä & Lassenius ,2006) หลังจากนั้นได้ทำการทดสอบโดยใช้ Mozilla Firefox โดยผลลัพธ์ที่ได้คือมาตรวัดซอฟต์แวร์สามารถพยากรณ์ร่องรอยที่ไม่ดีและข้อผิดพลาดได้อย่างดี แต่ภายใต้ของโมเดลหมวดหมู่ทั้ง 6 ระดับนั้นสามารถระบุร่องรอยที่ไม่ดีได้แค่บางหมวดหมู่เท่านั้น

งานวิจัย (Dallal & Briand , 2012) ได้เสนอมาตรวัดโคฮีชั่นที่ขึ้นอยู่กับมาตรวัดโคฮีชั่นของคลาสที่มีอยู่ก่อนแล้ว ซึ่งส่วนใหญ่จะอาศัยอยู่ใน method-method interaction (MMI) มาตรวัดที่ระบุว่าโค้ดตรงส่วนใหนบ้างที่ต้องทำรีแฟคทอริง ซึ่งมาตรวัดนี้มีชื่อว่า Low-level design Similarity-based Class Cohesion (LSCC) ซึ่งถูกนำมาใช้ในการหาจำนวนของการเชื่อต่อกันระหว่างเมธอดและคลาสซึ่งเก็บรวบรวมแอตทริบิวต์ทั่วไปที่มีอยู่ระหว่างเมธอดในคลาสและใช้เพื่อหาระดับจำนวนที่คล้ายคลึงกัน กระบวนการรีแฟคทอริงที่ถูกตรวจพบโดย LSCC คือการเคลื่อนย้ายเมธอดออกมาและทำการรีแฟคทอริงคลาส. การทดลองใช้มาตรวัด LSCC นั้นจะใช้กับกรณีตัวอย่างโปรแกรมที่พัฒนาโดยภาษา Java จำนวน 4 ตัวอย่าง โดยมีเป้าหมายเพื่อค้นหาความเกี่ยวของกันระหว่าง LSCC ความแต่ต่างของมาตรวัดโคฮีชั่น และหาความผิดพลาดของคลาส ผลลัพธ์ที่ได้นั้นให้ผลที่ดีมากในด้านการรีแฟคทอริงระดับคลาส อย่างไรก็ตาม LSCC มีข้อจำกัดคือไม่สามารถจำแนกคลาสแอตทริบิวต์และเมธอดที่มีระดับของการเข้า ถึงที่หลากหลาย (Access level modifiers)

Ref. code: 25595509035084XRS

งานวิจัย (Fontana, Braione et al. , 2012) ได้สำรวจและรวบรวมเครื่องมือและมาตรวัดสำหรับการตรวจสอบร่องรอยที่ไม่ดีประเภทต่างๆ โดยได้แบ่งตามประเภทของร่องรอยที่ไม่ดี 6 ประภท ดังนี้

- Duplicated Code ใช้เครื่องมือในการตรวจสอบโดยได้ให้ความเห็นว่าเครื่องมือ iPlasma และ infusion ใช้เทคนิคการตรวจสอบผ่านทางชุดของมาตรวัดที่เกี่ยวข้องกับการแยกโค้ดที่ซ้ำซ้อน ส่วนเครื่องมือ Checkstyle จะใช้เทคนิคนับจำนวนบรรทัด จำนวน 12 บรรทัดของโค้ดซ้ำกันติดต่อกันของโปรแกรม และ เครื่องมือ PMD จะพิจารณาการเกิดขึ้นของโค้ดซ้ำซ้อนอย่างน้อย 1 ครั้งและอย่างน้อยประกอบไปด้วย 25 โทเค็น

- Feature Envy ใช้มาตรวัดในการคำนวณ ซึ่งมาตรวัดประกอบไปด้วย AFTD(Access To Foreign Data) , LAA (Locality of Attribute Accesses) โดยมีสมการดังนี้

$$FDP \leq FEW \land ATFD > LAA < \frac{1}{3} \quad (2.1)$$

โดยที่ FEW จะถูกพิจารณาจากผู้เขียน ซึ่งเครื่องมือที่สนับสนุนคือ JDeodorant

- God Class ใช้มาตรวัดการการตรวจซึ่งประกอบไปด้วย WMC (Weighted Method Count) , TCC(Tight Class Cohesion) และ ATFD (Access to Foreign) โดยมีสมการคือ

$$WMC \geq VERY_HIGH \land AFTD > FEW \land TCC < \frac{1}{3} \quad (2.2)$$

โดยที่ VERY_HIGH และ FEW จะถูกพิจารณาจากผู้เขียน โดยมีเครื่องมือที่สนับสนุนคือ iPlasma , inFusion และ JDeodorant

- Large Class จะพิจารณาจากมาตรวัด NLOC (Number of Line Of Code) โดยมีเครื่องมือที่สนับสนุนคือ PMD และ Check Style

- Long Method พิจารณาโดยใช้มาตรวัด NLOC (Number of Line Of Code) ,CC (Cyclomatic Complexity) และ มาตรวัด Halstead โดยมีเครื่องมือที่สนับสนุนคือ CheckStyle และ PMD

- Long Parameter List พิจารณาโดยนับจำนวนของพารามิเตอร์โดยมีเครื่องมือที่สนับสนุนคือ PMD และ Checkstyle

Ref. code: 25595509035084XRS

หลังจากสำรวจเครื่องมือที่ใช้สำหรับตรวจหาร่องรอยที่ไม่ดีประเภทต่างๆแล้ว ได้ทำการทดสอบกับซอร์สโค้ดโปรเจค GanttProject ซึ่งถูกพัฒนาด้วยภาษา Java โดยมีทั้ง 6 เวอร์ชั่น โดยผลการทดลองนั้นให้ผลเป็นที่น่าพอใจ

งานวิจัย (Kaur & Maini , 2016) ได้สำรวจและรวบรวมมาตรวัดสำหรับตรวจสอบร่องรอยที่ไม่ดีในซอร์สโค้ด ซึ่งมาตรวัดประกอบไปด้วย WMC (Weighted Methods per Class) , DIT(Depth of Inheritance), Class Coupling, CBO (Coupling between Object), RFC (Response for class), NPM (Number of Public Methods), LCOM(Lack of Cohesion in Methods), Number of Parameters, Number of Accessor Methods , Function Point, Halsted Complexity, Tight Class Coupling และ Loose Class Coupling หลังจากนั้นได้นำมาตรวัดที่สำรวจได้มาสร้างกฎในการตรวจสอบร่องรอยที่ไม่ดี ดังแสดงในตารางที่ 2.3

ตารางที่ 2.3

แสดงมาตรวัดของงานวิจัย (Kaur & Maini , 2016)

Rules	Bad smell
a. if Number of source lines of code (NLOC) >50 and declared variables are not used. b. if Cyclomatic complexity >5 c. if Halstead effort E=D*V>15 If any above rule is/are true then bad smell is detected.	Long method
a. if Number of lines of code >300 and more than 5 long methods b. if Depth of inheritance tree (DIT)>5 c. if Class coupling > 10 If any above rule is/are true then bad smell is detected.	Large class
a. if Coupling between objects >5 b. if Lack of cohesion in methods>2 If any above rule is/are true then bad smell is detected.	Feature Envy
a. if Number of parameters of a method > 5 If above rule is true then bad smell is detected.	Long parameter list
a. if Lack of cohesion in methods>2 b. if Number of accessor methods >10 If any above rule is/are true then bad smell is detected.	Data class
a. if Number of methods =0 b. if LOC=100 and WMC/NOM<=2 If any above rule is/are true then bad smell is detected.	Lazy class

นอกจากมีการนำมาตรวัดซอฟต์แวร์ไปใช้ในด้านการระบุและรีแฟคทอริงร่องรอยที่ไม่ดีชนิดต่างๆแล้ว ยังมีการนำมาตรวัดซอฟต์แวร์ไปใช้ในด้านโมเดลคุณภาพซอฟต์แวร์อีกด้วย เช่น

งานวิจัย (Meananeatra, Rattanaleadnusorn et al. 2013) ได้นำเสนอโมเดลวัดคุณภาพด้านความสามารถด้านการวิเคราะห์หาข้อผิดพลาด (Analyzability) ที่ระดับเมธอดสำหรับ J2EE Application โดยโมเดลได้ประยุกต์ใช้โครงสร้างความสัมพันธ์ของ SIG Quality model (Heitlager, Kuipers et al., 2007) โครงสร้างคุณภาพด้านการวิเคราะห์ข้อผิดพลาดนั้นประกอบไปด้วยหลายแอททริบิวต์ที่มีคุณลักษณะที่สอดคล้องกับคุณภาพด้านการวิเคราะห์ข้อผิดพลาดและสามารถวัดค่าจากมาตรวัดที่สอดคล้องกับแอททริบิวต์ที่ใช้ได้แก่ Volume และ Complexity โดยมาตรวัดที่ใช้ได้แก่ MLOC, PAR, NBD และ VG

งานวิจัย (Chawla & Chhabra , 2015) ได้นำเสนอโมเดลคุณภาพซอฟต์แวร์ซึ่งมีความพร้อมต่อการใช้สูตรทางคณิตศาสตร์ที่จะหาจำนวนของคุณลักษณะคุณภาพทั้ง 4 คือ Analyzability Changeability Stability และ Testability ซึ่งมาตรวัดด้าน Analyzability ประกอบไปด้วย VG, TLOC, Comment, WMC, MFA โดยผลรวมนั้นจะเป็นเซ็ตของมาตรวัดซอฟต์แวร์ คุณลักษณะเหล่านี้ต่อไปจะทำหน้าที่เป็นปัจจัยในการประเมินด้าน Maintainability ที่เป็นคุณลักษณะเฉพาะของซอฟต์แวร์ที่สอดคล้องตามมาตรฐานที่ ISO 9126 กำหนด และการสูตรการปรับปรุงเพื่อคำนวณด้าน Maintainability นั้นจะสอดคล้องกับตามมาตรฐานตามที่ ISO 25010 กำหนดเช่นกัน

งานวิจัย (Juthamart, 2015) ได้เสนอโมเดลในการประเมินคุณภาพแวร์ด้านความสามารถในการวิเคราะห์เพื่อหาข้อผิดพลาดสำหรับซอร์สโค้ดโปรแกรมภาษาจาวาระดับคลาส ในการสร้างโมเดลนั้นได้ทำการสำรวจหามาตรวัดที่ส่งผลต่อคุณภาพของซอร์สโค้ดโปรแกรมด้านการวิเคราะห์หาสาเหตุข้อผิดพลาดและตำแหน่งของโปรแกรมที่ต้องแก้ไข และใช้วิธีการทางสถิติการถดถอยโลจิสติกแบบอันดับซึ่งมีมาตรวัดที่ได้รวบรวมไว้ทั้งสิ้น 11 ตัวด้วยกัน คือ WMC,APARM, NOC,DAM, LCOM,CBO,I,CBO ,NBD,DI, NOP และ CLOC เพื่อค้นหามาตรวัดที่มีความสัมพันธ์กับระดับของคุณภาพซอร์สโค้ดด้านการวิเคราะห์หาข้อผิดพลาดและตำแหน่งที่ต้องแก้ไขซึ่งแบ่งไว้ทั้งสิ้น 3 ระดับคือ 1.ควรปรับปรุง 2.พอใช้ 3.ดี จากนั้นได้ทำการทดสอบวัดความถูกต้องแม่นยำของโมเดลที่ได้พัฒนาขึ้น ในระดับความเชื่อมั่นที่ 95% ซึ่งทดสอบโดยสร้างโมเดลและทดสอบโมเดลกับกลุ่มตัวอย่างของคลาสจาวาจากโปรแกรม jEdit จำนวน 121 คลาส โดยมีกลุ่มคลาสที่เลือกมาใช้ในงานวิจัยนี้มี ได้การบันทึกว่ามีข้อผิดพลาดและให้ผู้เชี่ยวชาญวิเคราะห์หาจุดที่จะต้องแก้ไขและให้ค่าระดับคุณภาพของคลาส โดยจากการทดสอบพบว่ามาตรวัดที่มีผลต่อคุณภาพของซอร์สโค้ดด้านความสามารถในการวิเคราะห์เพื่อหาข้อผิดพลาด คือ Weighted Methods Per Class (WMC) และ Class Line of Code (CLOC)

Ref. code: 25595509035084XRS

งานวิจัยนี้ได้ทำการสำรวจและรวบรวมกลุ่มมาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา ในส่วนคุณลักษณะย่อย ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ดจากมาตรฐาน ISO 9126 และกลุ่มมาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection) ซึ่งมีรายละเอียดดังตารางที่ 2.4 ซึ่งแสดงภาพรวมงานวิจัยที่ผ่านมา

Ref. code: 25595509035084XRS

ตารางที่ 2.4

แสดงภาพรวมงานวิจัยที่ผ่านมา

งานวิจัย	จุดประสงค์	ชนิดร่องรอยที่ไม่ดี	มาตรวัด
(Meananeatra, Rongviriyapanish et al. 2011)	เสนอมาตรวัดเพื่อค้นหาร่องรอยที่ไม่ดี	Long method	MLOC, MCX , LCOM, CU, DU, PU
(Dallal and Briand 2012)	เสนอมาตรวัดโคฮีชั่น	Class Cohesion	LSCC
(Fontana, Braione et al. 2012)	รวจและรวบรวมเครื่องมือและมาตรวัดสำหรับการตรวจสอบร่องรอยที่ไม่ดีประเภทต่างๆ	Duplicated Code, Feature Envy, God Class, Large Class, Long Method, Long Parameter List	AFTD, LAA,WMC, TCC, NLOC,CC, Halstead
(Charalampidou, Ampatzoglou et al. 2015)	สำรวจและรวบรวมมาตรวัดเพื่อทำนายร่องรอยที่ไม่ดี	Long method	LCOM1, LCOM2, LCOM3, LCOM4, LCOM5, Coh, TCC, LCC, DC_D, DC_I, CC, SCOM, LSCC
(Zhao and Hayes 2006)	ทำนายคลาสที่ต้องทำรีแฟคทอริง	-	Halstead, WMC
(Srivisut and Muenchaisri 2007)	เสนอมาตรวัดสำหรับค้นหาร่องรอยที่ไม่ดีสำหรับ Aspect-Oriented	-	NAdP, NAdAsP, NSAdP, NNSAdP, SJP, NOASP, NPAs, NNPAs, SCT, NAMA, NIAs, NAdAs, SNOAsP, NPC
(Rani and Kaur 2014)	สร้างเครื่องมือสำหรับตรวจสอบร่องรอยที่ไม่ดี	Lazy Class, Long method, Comment lines , Large class	DIT, Coupling, WCM, Halstead , Cyclomatic complexity , NLOC , LOC

Ref. code: 25595509035084XRS

(Sreenu and Rao)	เสนอวิธีการรีแฟคทอริงและมาตรวัดสำหรับร่องรอยที่ไม่ดี	Lazy Class , Temporary Field	NOM, IVMC, DIT, LOC
(Kaur and Maini)	สำรวจและรวบรวมมาตรวัดสำหรับตรวจสอบร่องรอยที่ไม่ดี	Long method, Large class, Feature Envy, Long parameter list, Data class, Lazy class	WMC , DIT, Class Coupling, CBO RFC NPM LCOM, Number of Parameters, Number of Accessor Methods , Function Point, Halsted Complexity, Tight Class Coupling , Loose Class Coupling
(Higo, Kamiya et al. 2005)	เสนอเครื่องมือช่วยรีแฟคทอริงชื่อ Aries ในการบอกลักษณะของร่องรอยที่ไม่ดี	Code Clone	DCH, NRV, NAV
(Higo, Kusumoto et al. 2008)	เทคนิคในการระบุรีแฟคทอริงในโปรแกรมภาษาเชิงวัตถุโดยใช้มาตรวัด	Code Clone	DCH, NRV, NAV
(Singh and Kahlon 2012)	ใช้มาตรวัดเพื่อนำมาช่วยหาคลาสที่เกิดข้อผิดพลาดและค้นหาร่องรอยที่ไม่ดีของโปรแกรม	-	NOC, DIT, LCOM, LCOM4, WMC, Puf, EncF, CBO, RFC, NOD
(Meananeatra, Rattanaleadnusorn et al. 2013)	ได้นำเสนอโมเดลวัดคุณภาพด้านความสามารถด้านการวิเคราะห์หาข้อผิดพลาด (Analyzability) ระดับเมธอด	-	MLOC, PAR, NBD, VG

Ref. code: 25595509035084XRS

(Juthamart 2015)	เสนอโมเดลประเมินคุณภาพของซอร์สโค้ดด้านความสามารถในการวิเคราะห์เพื่อหาข้อผิดพลาด (Analyzability) ระดับคลาส	-	CLOC, DAM, NOP,APARM,CBO,I,LCOM,NBD,DI,NOC
(Chawla and Chhabra 2015)	เสนอโมเดลคุณภาพซอฟต์แวร์ซึ่งหาจำนวนของคุณลักษณะคุณภาพทั้ง 4 คือ Analyzability Changeability Stability และ Testability		VG, TLOC, Comment, WMC, MFA, NBD, CBO, LCOM, DIT, NORM.NOM, NSC,RFC, CE
(SISSy 2011)	เครื่องมือสำหรับวิเคราะห์โปรแกรมภาษาเชิงวัตถุ		ATFD,AMW,NOM,NOAM,NOPA,TCC,WMC,WOC,CDISP,CINT, CYCLO,FDP,LAA,MAXNESTING
งานวิจัยนี้	เสนอเครื่องมือที่ใช้คำนวณมาตรวัดและระบุรีแฟคทอริง	Long method	NOS, MCX , LCOM, PAR, NBD, CU, DU, PU

Ref. code: 25595509035084XRS

บทที่ 3
วิธีการวิจัย

ในบทนี้จะกล่าวถึงวิธีการดำเนินการวิจัยของวิทยานิพนธ์ ซึ่งประกอบไปด้วยภาพรวมของวิธีการดำเนินงานวิจัย วีธีการสร้างเมตาโมเดลให้รองรับกับการคำนวณมาตรวัด การพัฒนาส่วนเสริมโปรแกรมอิคลิปส์(Eclipse Plugin) เครื่องมือที่ใช้พัฒนา ขั้นตอนการพัฒนา และวิธีการทดลอง

3.1 ภาพรวมของวิธีการดำเนินงานวิจัย

ในส่วนนี้กล่าวถึงวิธีการดำเนินงานวิจัย โดยเน้นภาพรวมของกระบวนการพัฒนาซอฟต์แวร์แบบเอ็มดีเอ (MDA: Model-driven Architecture) ที่สามารถรองรับการคำนวณมาตรวัดสำหรับโปรแกรมเชิงวัตถุ การคัดเลือกมาตรวัดที่เกี่ยวข้องจากงานวิจัยที่ผ่านมา และแนวทางการแปลงแบบจำลอง

3.1.1 กระบวนการพัฒนาซอฟต์แวร์แบบเอ็มดีเอ (MDA: Model-driven Architecture) ที่สามารถรองรับการคำนวณมาตรวัดสำหรับโปรแกรมเชิงวัตถุ

งานวิจัยนี้ได้ประยุกต์ใช้ กระบวนการพัฒนาซอฟต์แวร์แบบเอ็มดีเอ (MDA: Model-driven Architecture) เพื่อเป็นแนวทางในการแปลงซอร์สโค้ดภาษาโปรแกรมเชิงวัตถุให้เป็นแบบจำลองโดยที่ไม่ขึ้นกับภาษาใดๆ เพื่อช่วยให้นักวิเคราะห์โครงสร้างโปรแกรมสะดวกในการวิเคราะห์โครงสร้างโปรแกรมเชิงวัตถุ โดยงานวิจัยนี้ได้เสนอวิธีการและพัฒนาเครื่องมือที่ช่วยในการวิเคราะห์โครงสร้างโปรแกรมเชิงวัตถุที่สามารถรองรับได้มากกว่า 1 ภาษาคือ JAVA , C++ ซึ่งเครื่องมือนั้นจะเป็นส่วนเสริมของอีคลิปส์ (Eclipse plugin) เนื่องจากอีคลิปส์นั้นเป็นโอเพนซอร์ส (Open source) และเป็นเครื่องมือสำหรับพัฒนาโปรแกรมที่มีความนิยมอย่างมากจึงมีความเหมาะสมที่จะใช้งานและพัฒนาต่อยอดและเพื่อให้นักพัฒนาซอฟต์แวร์หรือนักวิจัยที่สนใจสามารถนำไปใช้งานหรือพัฒนาต่อยอดได้ โดยภาพรวมในการพัฒนาส่วนเสริมอีคลิปส์นั้นจะแสดงดังภาพที่ 3.1 โดยขั้นตอนแรกจะเป็นการนำซอร์สโค้ดโปรแกรมภาษาโปรแกรมเชิงวัตถุมาทำการสกัดแบบจำลอง (Model Extractor) ให้อยู่ในรูปแบบจำลองเอกซ์เอ็มไอ(XMI: XML Metadata Interchange) ที่ไม่ขึ้นกับแพลตฟอร์มใดๆโดยแบบจำลองนี้จะเป็นแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ ซึ่ง การแปลงซอร์สโค้ดโปรแกรมให้อยู่ในรูปของแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ นี้จะทำให้แสดงให้เห็นถึงโครงสร้างทั้งหมดของซอร์สโค้ด หลังจากได้แบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ แล้วจะนำแบบจำลองไปเข้าสู่ขั้นตอนต่อไปคือขั้นตอนการแปลงแบบจำลอง (Model Transformation Process) ซึ่งจะทำทั้งสิ้น 2 ขั้นตอนด้วยกันคือการแปลงแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ ไปสู่แบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม (PDG: Program Dependency Graph) และการแปลงแบบจำลองกราฟแสดงความสัมพันธ์ของ

Ref. code: 25595509035084XRS

โปรแกรมไปสู่แบบจำลองเก็บค่ามาตรวัดของโปรแกรม หลังจากผ่านขั้นตอนกระบวนการแปลงแบบจำลองแล้วจะได้แบบจำลองเก็บค่ามาตรของโปรแกรม ที่สามารถนำมาคำนวณหาค่ามาตรวัดซอฟแวร์ 2 กลุ่มคือ มาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา (Metrics for Maintainability) ในส่วนคุณลักษณะย่อย (Sub characteristics) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) จากมาตรฐาน ISO 9126 และ มาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection)

ภาพที่ 3.1

แสดงภาพรวมของงานวิจัย

3.1.2 การคัดเลือกมาตรวัดที่เกี่ยวข้องจากงานวิจัยที่ผ่านมา

งานวิจัยนี้นำเสนอเครื่องมือเพื่อหาค่ามาตรวัดซอฟแวร์ 2 กลุ่มคือ มาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา ในส่วนคุณลักษณะย่อย ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด จากมาตรฐาน ISO 9126 และ มาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม โดยในส่วนของมาตรวัดกลุ่มแรก

Ref. code: 25595509035084XRS

นั้นจะมาจากการสำรวจจากงานวิจัยที่ผ่านมา ส่วนมาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสมนั้นผู้วิจัยได้นำมาจากงานวิจัย (Meananeatra & Rongviriyapanish et al. , 2011) ซึ่งผู้วิจัยนั้นได้นำมาพัฒนาต่อยอดเป็นเครื่องมือสำหรับคัดเลือกวิธีรีแฟคทอริง

สำหรับการคัดเลือกมาตรวัดที่เกี่ยวข้องจากงานวิจัยที่ผ่านมานั้น เมื่อพิจารณาจากตารางที่ 2.4 ผู้วิจัยได้พิจารณามาตรวัดที่ส่งผลทางด้าน ความสามารถในการบำรุงรักษาในส่วนคุณลักษณะย่อยด้านการวิเคราะห์ โดยเน้นจากกลุ่มมาตรวัดที่เกี่ยวข้องกับเมธอดและมีความถี่ในการใช้งานค่อนข้างมาก ซึ่งผู้วิจัยพบว่าเมื่อทำการแบ่งตามคุณลักษณะของมาตรวัดเช่น วัดขนาดของระบบ (Volume) พบว่า งานวิจัย (Meananeatra, Rongviriyapanish et al. 2011)ได้ใช้มาตรวัด Number Of Statement In Method (NOS) เพื่อวัดจำนวนฟังก์ชั่นหรือ Statement ของเมธอด ซึ่งต่างจากงานวิจัย (Fontana, Braione et al. 2012) และงานวิจัย (Rani and Kaur 2014) ที่ได้ใช้มาตรวัด Number of source line of code (NLOC) วัดโดยการนับจำนวนบรรทัดของเมธอดเพื่อหาขนาดของเมธอด ซึ่งผู้วิจัยมองว่าการวัดขนาดของเมธอดโดยการนับจำนวนบรรทัดหรือ Statement นั้นเป็นสิ่งที่จำเป็นเพราะว่าหากเมธอดนั้นมีขนาดที่ใหญ่เกินไปอาจส่งผลทางด้านการบำรุงรักษาได้ นอกจากการนับจำนวนบรรทัดหรือจำนวน Statement แล้วยังได้มีบางงานวิจัยที่ได้ใช้มาตรวัดประเภทอื่นๆเพื่อวัดขนาดเมธอด เช่น งานวิจัย (Kaur & Maini , 2016) และ งานวิจัย (Meananeatra, Rattanaleadnusorn et al. , 2013) ได้นำมาตรวัด Number of parameter (PAR) ซึ่งเป็นการวัดจำนวนพารามิเตอร์ของเมธอด มาช่วยในการวัดขนาดของเมธอด โดยผู้วิจัยมองว่าเมธอดใดที่มีจำนวนพารามิเตอร์ที่มากเกินไปอาจส่งผลให้ยากต่อการแก้ไขในอนาคตได้ โดยนอกจากการวัดคุณสมบัติทางด้านขนาดของระบบแล้วยังมีบางงานวิจัยที่ใช้มาตรวัดในการวัดความซับซ้อน (Complexity) ของเมธอดด้วย เช่น งานวิจัย (Meananeatra, Rongviriyapanish et al. ,2011) , (Rani & Kaur , 2014) , (Meananeatra, Rattanaleadnusorn et al. ,2013) และ (Chawla & Chhabra ,2015) ได้ใช้มาตรวัด McCabe Cyclomatic Complexity (VG) ซึ่งเป็นมาตรวัดที่ใช้วัดความซับซ้อน มาทำการวัดความซับซ้อนของเมธอด ซึ่งต่างจากงานวิจัย (Fontana, Braione et al. , 2012) , (Zhao & Hayes , 2006) และ (Kaur & Maini , 2016) ได้ใช้มาตรวัด Halstead complexity มาใช้ในการในการวัดความซับซ้อนของเมธอด นอกจากนี้มีบางงานวิจัยได้ใช้มาตรวัดที่นอกเหนือจากนี้มาใช้ในการวัดความซับซ้อนอีกด้วย เช่น งานวิจัย (Meananeatra, Rattanaleadnusorn et al. ,2013), (Juthamart 2015) และ (Chawla & Chhabra ,2015) ได้ใช้มาตรวัด Nested block depth (NBD) ที่วัดความลึกของลำดับชั้นของปีกกาของ Statement ประเภทเงื่อนไข เช่น if/then switch for while เป็นต้น มาช่วยในการวัดความซับซ้อนของเมธอด โดยผู้วิจัยเล็งเห็นว่าการที่เมธอดมีความซับซ้อนสูงนั้นอาจส่งผลต่อการทำความเข้าใจระบบได้ นอกจากนี้อย่างมีมาตรวัด

ที่น่าสนใจ เช่น งานวิจัย (Charalampidou, Ampatzoglou et al., 2015), (Kaur & Maini, 2016), (Singh & Kahlon ,2012) , (Juthamart ,2015) และ (Chawla & Chhabra ,2015) ได้ใช้มาตรวัด Lack of cohesion in method (LCOM) เพื่อวัดการเกาะเกี่ยวกันภายในคลาส

นอกจากการคัดเลือกมาตรวัดจากงานวิจัยที่ผ่านมาแล้ว ในส่วนของมาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection) ซึ่งทางผู้วิจัยได้พัฒนางานวิจัยต่อยอดจาก (Meananeatra, Rongviriyapanish et al. ,2011) ซึ่งเป็นงานวิจัยที่เกี่ยวกับการระบุวิธีรีแฟคทอริงของร่องรอยที่ไม่ดีประเภทลองเมธอดโดยใช้กลุ่มมาตรวัด ซึ่งกลุ่มมาตรวัดนั้นประกอบไปด้วย Count of computation of v in path (CU) , Count of assignment of v in path (DU) , Count of predicate of v in path (PU) ดังนั้นจากการสำรวจงานวิจัยที่ผ่านมาสามารถสรุปมาตรวัดและสูตรการคำนวณดังตารางที่ 3.1

ตารางที่ 3.1

สรุปมาตรวัดจากงานวิจัยที่ผ่านมา

มาตรวัด	สูตรการคำนวณค่ามาตรวัด	วัตถุประสงค์ของมาตรวัด	คุณลักษณะย่อย	อ้างอิง
Number Of Statement In Method (NOS)	NOS = จำนวนฟังก์ชั่นในเมธอด โดยไม่ครอบคลุม Java Ternary operator	วัดขนาดของเมธอด	Analyzability	(Meananeatra, Rattanaleadnusorn et al. 2013)
Number of parameter in method (PAR)	PAR = จำนวนของพารามิเตอร์ในเมธอด	วัดจำนวนพารามิเตอร์ของเมธอด		(Kaur and Maini) และ งานวิจัย (Meananeatra, Rattanaleadnusorn et al. 2013),(Juthamart 2015)
Nest block depth (NBD)	NBD = จำนวนความลึกของชั้นปีกกา	วัดจำนวนความลึกของชั้นปีกกา		(Meananeatra, Rattanaleadnusorn et al. 2013), (Juthamart 2015) และ (Chawla and Chhabra 2015)
McCabe Cyclomatic Complexity (VG)	VG = P+1 เมื่อ P = โหนดเงื่อนไขทางเลือก	วัดความซับซ้อนของเมธอด		(Meananeatra, Rongviriyapanish et al. 2011) , (Rani and Kaur 2014) , (Meananeatra, Rattanaleadnusorn et al. 2013) และ (Chawla and Chhabra 2015)
Lack Of Cohesion In Method (LCOM)	$\frac{\left[\frac{1}{v}\sum_{i=1}^{v} m(v_i)\right] - m}{1 - m}$ m = จำนวนเมธอด v = จำนวนแอททริบิวต์ m(v_i) = เมธอดที่เรียกใช้แอททริบิวต์ v_i	วัดการเกาะเกี่ยวกันในคลาส		(Charalampidou, Ampatzoglou et al. 2015), (Kaur and Maini), (Singh and Kahlon 2012) , (Juthamart 2015) , (Chawla and Chhabra 2015)

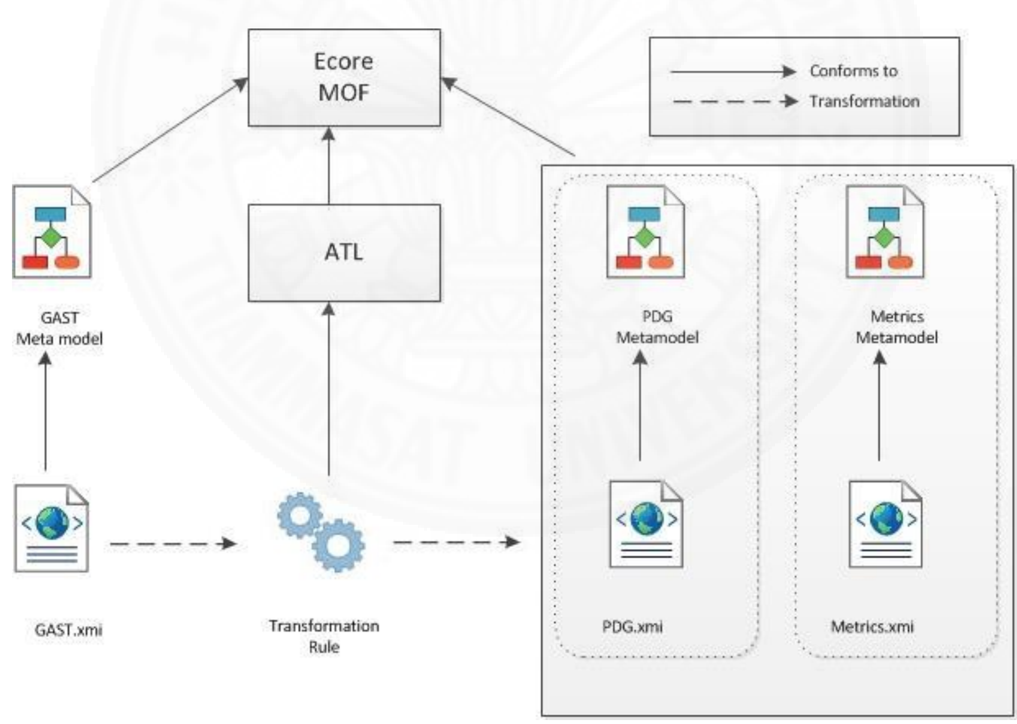
Ref. code: 25595509035084XRS

CU(V)	Count of computation of v in path	Computed of a variable	-	(Meananeatra, Rongviriyapanish et al. 2011)
DU(V)	Count of assignment of v in path	Defined of a variable	-	
PU(V)	Count of predicate of v in path	Decided of a variable	-	

Ref. code: 25595509035084XRS

3.1.3 แนวทางการแปลงแบบจำลอง (Model-to-Model Transformation)

งานวิจัยนี้เลือกใช้วิธีการแปลงแบบจำลองแบบสร้างเมตาโมเดลด้วยภาษา MOF (Meta-Object Facility) ดังนั้นแนวทางการแปลงแบบจำลองในงานวิจัยนี้คือวิธีการแปลงแบบจำลองแบบเมตาโมเดล (Meta-model Transformation) ซึ่งวิธีกานี้แบบจำลองต้นทาง (Source Model) และแบบจำลองปลายทาง (Target Model) ที่ออกแบบจะต้องสอดคล้อง (Conform) กับเมตาโมเดล (Meta-model) ของแต่ละแบบจำลอง โดยแบบจำลองต้นทางในที่นี้คือแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ (GAST) ที่มาจากการทำขั้นตอน การแปลงซอร์สโค้ดให้อยู่ในรูปของแบบจำลอง (Model Extractor) ซึ่งเป็นแบบจำลองที่ออกแบบโดยใช้เมตาโมเดลของแบบจำลองต้นทาง และแบบจำลองปลายทางคือ แบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม (PDG) และแบบจำลองเก็บค่ามาตรของโปรแกรม (Metrics) ตามลำดับโดยมีกฎการแปลง (Transformation Rule) ใช้สำหรับนิยามการแปลงแบบจำลอง ซึ่งแสดงดังภาพที่ 3.2



ภาพที่ 3.2 แสดงการแปลงแบบจำลองของงานวิจัย

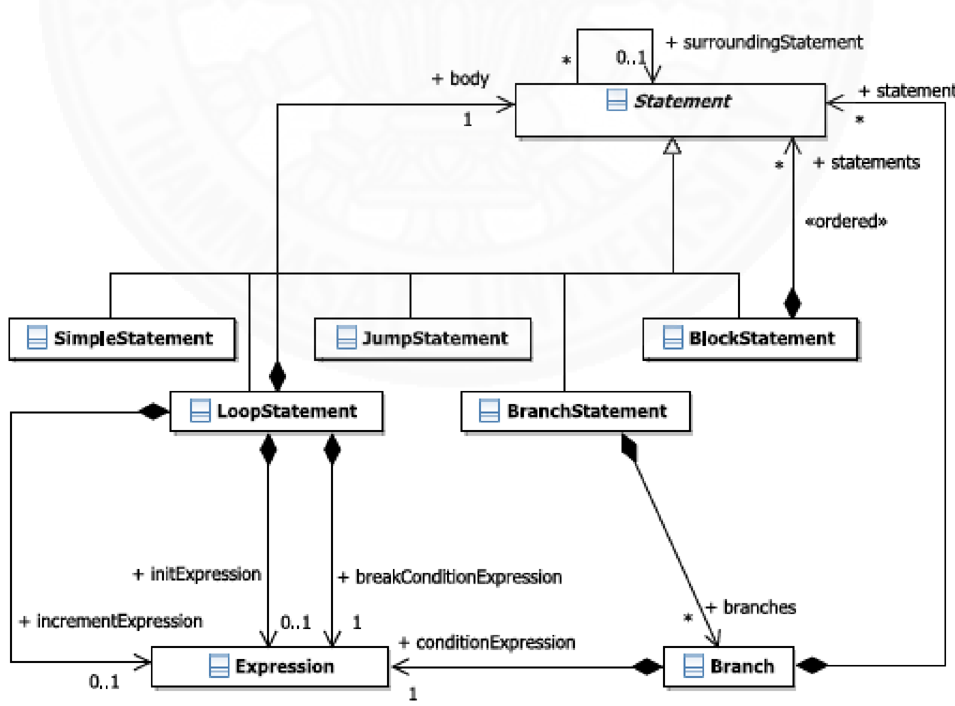
Ref. code: 25595509035084XRS

3.2 วิธีการสร้างเมตาโมเดลให้รองรับกับการคำนวณมาตรวัด

ในส่วนนี้จะกล่าวถึงวิธีการสร้างเมตาโมเดลให้รองรับกับการคำนวณมาตรวัดซึ่งประกอบไปด้วย เมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ (GAST : General Abstract Syntax Tree) เมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม (PDG : Program Dependency Graph Meta-model) เมตาโมเดลมาตรวัด (Metrics)

3.2.1 เมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ (General Abstract Syntax Tree Meta-model)

งานวิจัยนี้จะออกแบบเมตาโมเดลที่ใช้คำนวณค่ามาตรวัดโดยอยู่บนพื้นฐานของโครงสร้างต้นไม้ (Koziolek, Schlich et al. , 2011) ซึ่งเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ นั้นถูกสร้างขึ้นมาเพื่อสนับสนุนภาษาโปรแกรมเชิงวัตถุ เช่น JAVA, C หรือ C++ เป็นต้น โดยเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ แสดงดังภาพที่ 3.4 ซึ่งเมตาโมเดล โครงสร้างต้นไม้วากยสัมพันธ์ นั้น ประกอบไปด้วย แพ็คเกจ (Package) ต่างๆตามลักษณะการทำงาน ซึ่งงานวิจัยนี้จะเจาะจงไปที่ แพ็คเกจ Statement ซึ่งเกี่ยวข้องกับฟังก์ชั่นหรือเมธอด โดยแพ็คเกจ Statement นั้นประกอบไปด้วย Statements ต่างๆซึ่งแสดงตามลักษณะการทำงาน ดังภาพที่ 3.3 และตารางที่ 3.2 แสดงการอธิบายอิลิเม้นของแพคเกจ Statement



ภาพที่ 3.3 แสดงแพ็คเกจ Statement ของเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์

Ref. code: 25595509035084XRS

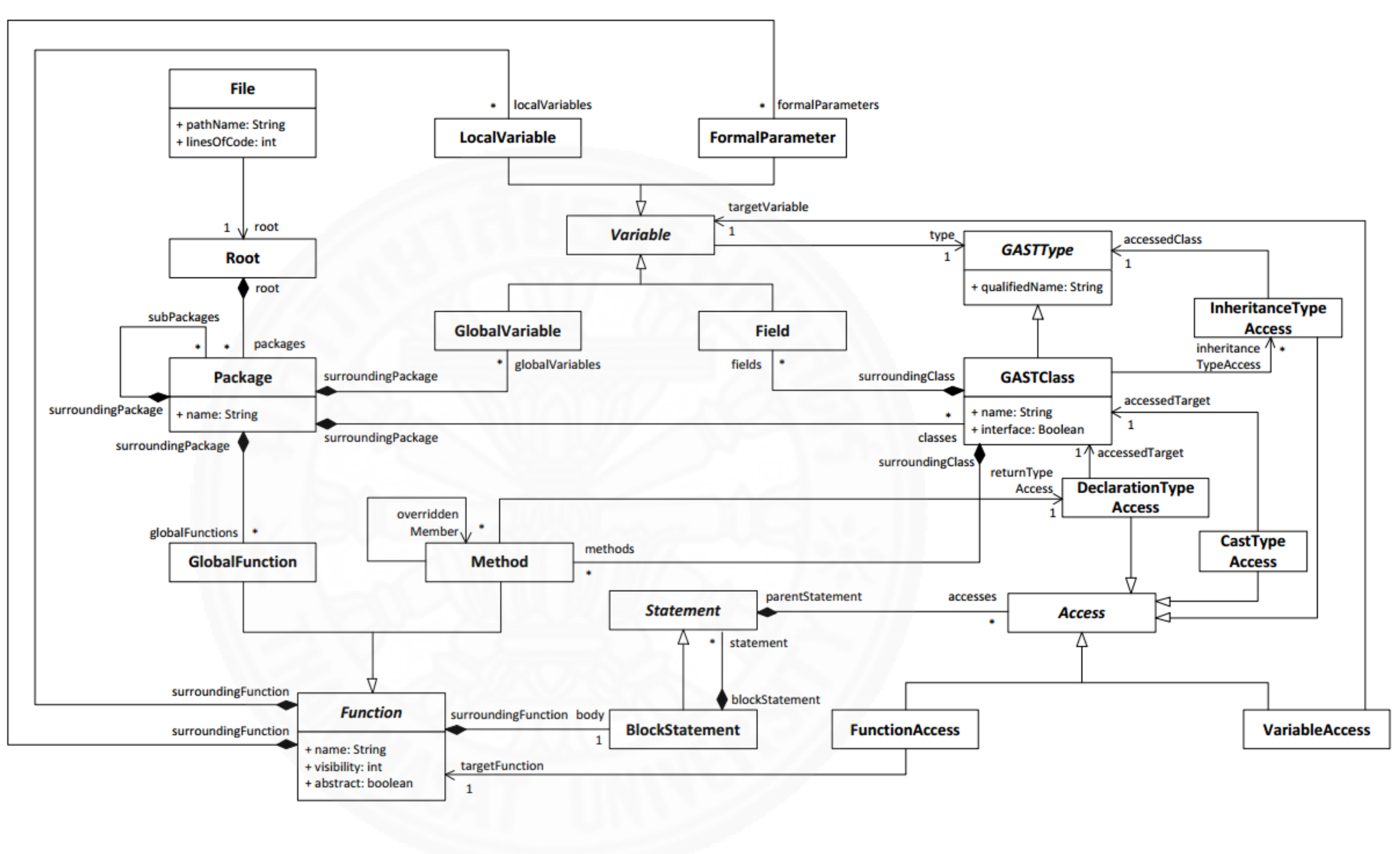
ตารางที่ 3.2

อิลิเม้นของแพคเกจ Statement

Element	Description
BlockStatement	เอนทิตี้ที่ใช้แสดง function body ของฟังก์ชันหรือเมธอด
-statement	แอททริบิวต์ที่เป็นลิสต์ที่เลียงลำดับกันของ Statement ที่อยู่ภายใน BlockStatement นี้
-surroundingFunction	ถ้า BlockStatement นี้เป็น function body แอททริบิวต์นี้จะเป็นการอ้างอิงกับ Statement ที่อยู่ข้างใน
-synchronize	ถ้าแอททริบิวต์นี้เป็น true BlockStatement นี้คือ synchronize block
Branch	เอนทิตี้ที่ใช้แสดงสาขาของ BranchStatment
-conditionExpression	แอททริบิวต์ที่ใช้แทน condition expression ของสาขานี้
-statement	แอททริบิวต์ที่ใช้แทน body ของสาขานี้
BranchStatement	เอนทิตี้ที่ใช้แทน Statement ที่มีสาขา เช่น if, switch statement
-branch	แอททริบิวต์
CatchBlock	เอนทริตี้ที่แทน Catch block ถ้าสืบทอดมาจาก BlockStatement จะมีการ catch parameter
-catchParameter	อ้างอิงกับ catch parameter สำหรับ Catch block
ExceptionHandler	เอนทริตีที่เป็นการดักจับ exception
-catchBlocks	แทนลิสต์ของ catch block ของการดักจับ exception นี้
-finallyBlock	แทน finally block ของการดักจับ exception นี้
-guardeBlock	แทน block statement ที่มี guard

Ref. code: 25595509035084XRS

GASTBehavior	เอนทิตี้ที่เกี่ยวกับกิจกรรมของ GAST meta-model
-blockStatement	แอททริบิวต์ที่เป็นคอนโทรล-โฟลว์ของพฤติกรรมการสร้างโมเดลผ่านทางโครงสร้าง statement
GASTExpression	เอนทิตี้ที่เป็นรูท (root) ของ expression
JumpStatement	เอนทิตี้ที่ใช้แทนการโดดที่เป็น non-condition เช่น Throw exception , return เป็นต้น
-kind	ใช้แทนชนิดของการโดด
LoopStatement	เอ็นทิตี้ที่เกี่ยวของกับการวนลูปทุกชนิด เช่น for , while เป็นต้น
-body	แทน statement ที่อยู่ภายในลูป
-breakConditionExpression	แทน expression ที่เป็น break-condition ของลูป
-incrementExpression	แทน increment expression ของลูป
-initExpression	แทน initialization expression ของลูป
-kind	แทนชนิดของลูป
SimpleStatement	เอนทิตี้ที่เกี่ยวกับ statement ประเภท variable, assignment ซึ่งแต่ละ statement จะจัดเลียงกันเป็นคอนโทรล-โฟลว์
Statement	เอนทิตี้ที่เป็นรท (root) ของ statement ทั้งหมด
- accesses	แทนลิสต์ของการเข้าถึง statement ที่อยู่ภายใน statement นั้นๆ
-expression	แทน expression ที่เกิดขึ้นใน statement นั้นๆ
-surroundingStatement	แทน statement ที่อยู่รอบๆของ statement นั้นๆ



ภาพที่ 3.4 แสดงภาพรวมของเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ (General abstract syntax tree Meta-model)

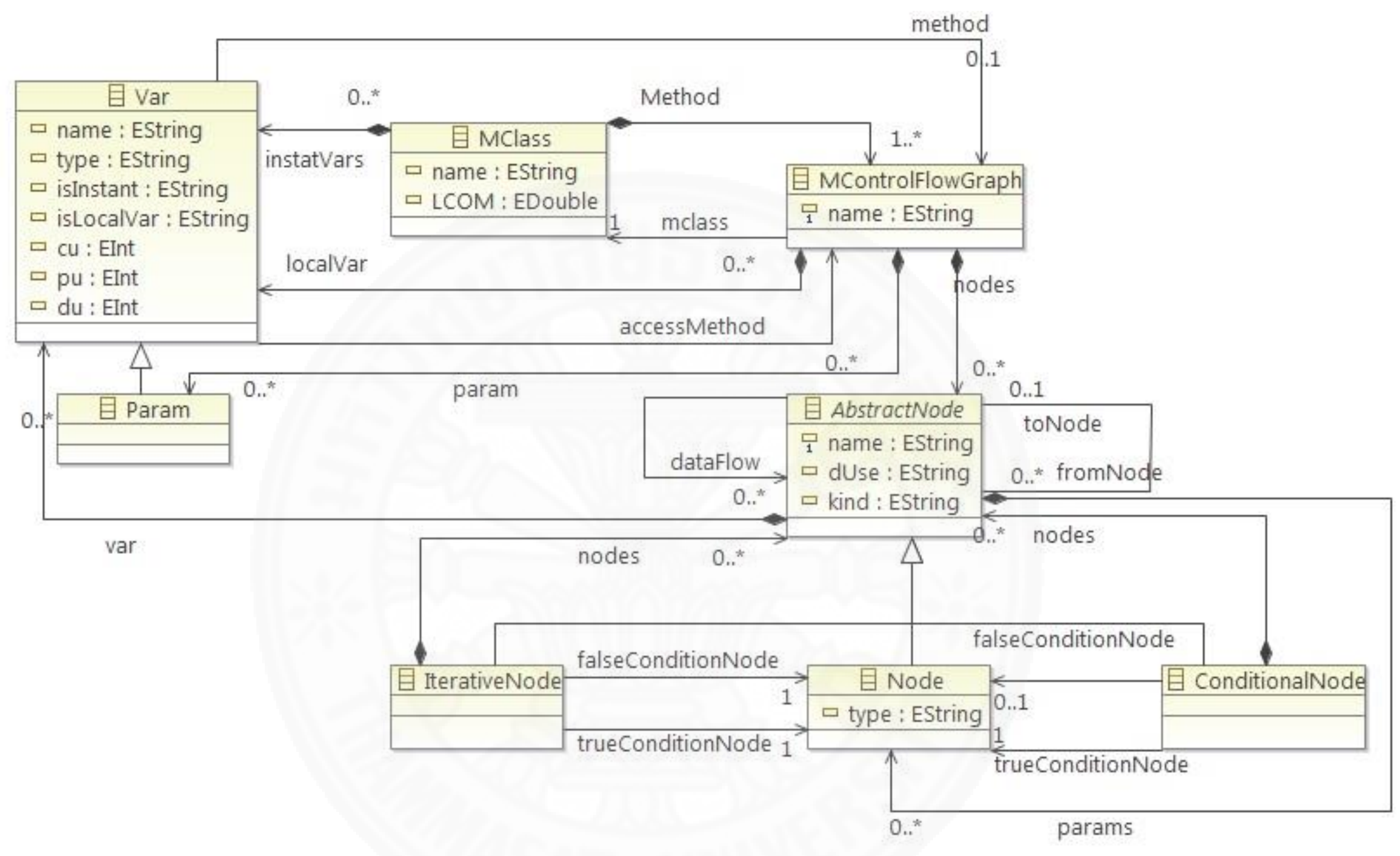
Ref. code: 25595509035084XRS

3.2.2 การออกแบบเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม (Program Dependency Graph Meta model)

เมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม คือเมตาโมเดลที่ใช้อธิบายลักษณะโครงสร้างการทำงานของโปรแกรมภาษาเชิงวัตถุซึ่งประกอบไปด้วยคลาสที่แสดงถึงคลาสของ Object ซึ่งในคลาสนั้นจะประกอบไปด้วยแอตทริบิวต์และฟังก์ชั่นหรือเมธอดซึ่งในเมธอดนั้นจะประกอบไปด้วย Statement ต่างๆ เช่น If-Else , while , return เป็นต้น รวมไปถึงลำดับการไหล(Control-flow) ของ Statement และลำดับการไหลของข้อมูล (Data-flow) ซึ่งจะไหลจาก Statement หนึ่งไปสู่อีก Statement โดยเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม นั้นได้ออกแบบตามแนวทางของงานวิจัย (Meananeatra, Rongviriyapanish et al. ,2011) ซึ่งได้ใช้ Control-Flow Graph และ Data-Flow Graph วิเคราะห์โครงสร้างของเมธอด โดยองค์ประกอบของเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม นั้นได้แสดงดังภาพที่ 3.5 ซึ่งมีรายละเอียดต่างๆ ดังนี้

เอนทิตี้ MControlflowGraph ใช้สำหรับแทนเมธอดที่อยู่ในคลาสซึ่งในเมธอดจะประกอบด้วย Statement ต่างๆที่เรียงลำดับการไหล (Control-flow) ตั้งแต่ Statement แรกไปจนถึง Statement สุดท้ายโดย Statement นั้นจะแทนด้วย AbstractNode ซึ่งมี fromNode ใช้สำหรับบอกลำดับการไหลของ Node ก่อนหน้าและ toNode ใช้สำหรับบอก Node ลำดับถัดไปสำหรับลำดับการไหลของข้อมูล จะใช้ dataFlow เพื่อบอก Node ถัดไปของข้อมูล เอนทิตี้ Node นั้นใช้สำหรับแทน Statement ทั่วไป เช่น double totalPrice = 3.5; เป็นต้น เอนทิตี้ConditionNode ใช้สำหรับแทน Statement ที่เกี่ยวข้องกับเงื่อนไข เช่น if , else เป็นต้น เอนทิตี้IterativeNode นั้นจะใช้สำหรับแทน Statement ที่เกี่ยวข้องกับ Loop เอนทิตี้ Param แทนพารามิเตอร์ของเมธอด โดยเอนทิตี้เหล่านี้ สามารถสรุปดังตารางที่ 3.3

Ref. code: 25595509035084XRS



ภาพที่ 3.5 เมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม (Program Dependency Graph Meta-model)

Ref. code: 25595509035084XRS

ตารางที่ 3.3

แสดงคำอธิบายอิลิเม้นของเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม

Element	Description
MClass	เอนทิตี้ที่แสดงรายละเอียดของคลาสโปรแกรมเชิงวัตถุ
MControlFlowGraph	เอนทิตี้ที่ใช้สำหรับแทนเมธอดที่อยู่ภายในคลาสของโปรแกรมเชิงวัตถุ
Var	เอนทิตี้ที่ใช้สำหรับตัวแปรชนิดต่างๆ
Param	เอนทิตี้ที่ใช้แทนพารามิเตอร์ของเมธอด โดยสืบทอดมาจาก Var
AbstractNode	เอนทิตี้ที่เป็นแอบสแตร็ค โดยใช้แทร Statement ต่างๆที่อยู่ในเมธอด
IterativeNode	เอนทิตี้ที่ใช้สำหรับ Statement ที่เกี่ยวข้องกับ Loop เช่น For, While เป็นต้น ซึ่งสืบทอมาจาก AbstractNode
Node	เอนทิตี้ที่ใช้แทน Statement ทั่วไปเช่น Simple Statement หรือ Jump Statement โดยสืบทอดมาจาก AbstractNode
ConditionalNode	เอนทิตี้ที่ใช้แทน Statment ที่เป็นทางเลือกหรือเงือนไข เช่น If เป็นต้น โดยสืบทอดมาจาก AbstractNode

3.2.3 การออกแบบเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม (Metrics Meta Model)

เมตาโมเดลเก็บค่ามาตรวัดของโปรแกรมนั้นจะใช้สำหรับการคำนวณหามาตรวัดต่างๆ ทั้งมาตรวัดที่เกี่ยวข้องกับคลาสและมาตรวัดที่เกี่ยวข้องกับเมธอด โดยมาตรวัดแต่ละชนิดนั้นจะมาจากการกระบวนการแปลงแบบจำลองระหว่างแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมกับแบบจำลองเก็บค่า

Ref. code: 25595509035084XRS

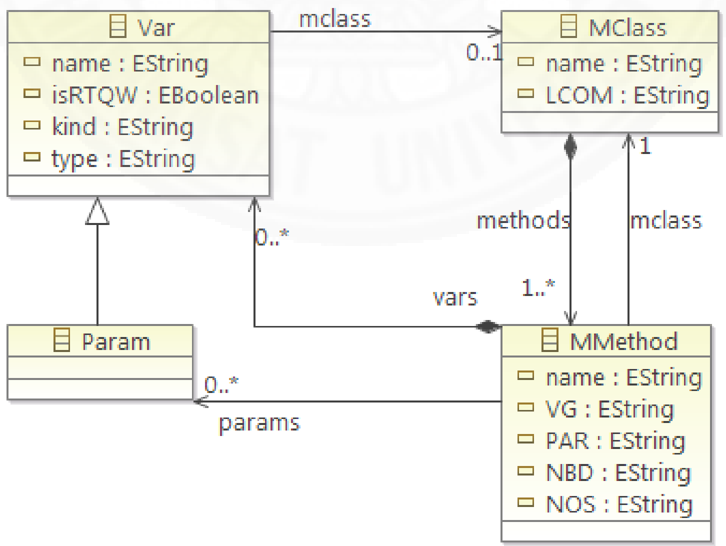
มาตรของโปรแกรม โดยจะกล่าวในภายหลัง เมตาโมเดลเก็บค่ามาตรของโปรแกรม จะประกอบไปด้วยเอนทิตี้ต่างๆ ดังตารางที่ 3.4

ตารางที่ 3.4

แสดงคำอธบายอิลิเม้นของเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม

Element	Description
MClass	เอนทิตี้ที่ใช้สำหรับมาตรวัดที่เกี่ยวข้องกับคลาส
MMethod	เอนทิตี้ที่ใช้สำหรับมาตรวัดที่เกี่ยวข้องกับเมธอด
Var	เอนทิตี้ที่ใช้แสดงรายละเอียดต่างๆของตัวแปร
Param	เอนทิตี้ที่ใช้แสดงรายละเอียดต่างๆของพารามิเตอร์

จากตารางที่ 3.4 เอนทิตี้ MClass เป็น เอนทิตี้ที่ใช้สำหรับเก็บมาตรวัดต่างๆที่เกี่ยวข้องกับคลาส เช่น LCOM เป็นต้น ซึ่งใน MClass จะมีความความสัมพันธ์กับ เอนทิตี้ MMethod ซึ่งใช้สำหรับมาตรวัดที่เกี่ยวข้องกับ เมธอด ซึ่งประกอบไปด้วยมาตรวัด VG, PAR, NBD และ NOS เป็นต้น เอนทิตี้ Var นั้นจะใช้สำหรับเก็บตัวแปรชนิดต่างๆและพารามิเตอร์ รวมไปถึงชนิดของการทำรีแฟคทอริงที่เกี่ยวข้องกับงานวิจัยนี้ โดยภาพรวมของเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม ได้แสดงดังภาพที่ 3.6



ภาพที่ 3.6 เมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม (Metrics Meta-model)

3.3 เครื่องมือที่ใช้พัฒนาโปรแกรม

เครื่องมือที่ใช้ในการวิจัยประกอบไปด้วยฮาร์ดแวร์และซอฟต์แวร์ โดยมีรายละเอียดดังนี้

3.3.1 องค์ประกอบทางด้านฮาร์ดแวร์

3.3.1.1 เครื่องคอมพิวเตอร์ส่วนบุคคล CPU Intel Core(TM) i5-2430M 2.40 GHz Memmory DDR3 8GB

3.3.2 องค์ประกอบทางด้านซอฟต์แวร์

3.3.2.1 ระบบปฏิบัติการ Window 7 Ultimate Service Pack 1

3.3.2.2 Java เวอร์ชั่น 1.6

3.3.2.3 Eclipse เวอร์ชั่น Kepler ใช้สำหรับพัฒนาปี

3.3.2.4 Eclipse Modeling Tools เวอร์ชั่น 1.5.2 ใช้สำหรับสร้างเมตาโมเดล ซึ่งเป็น Plugin ของ Eclipse

3.3.2.5 ATL SDK เวอร์ชั่น3.3.1 ใช้สำหรับการแปลงแบบจำลอง ซึ่งเป็น Plugin ของ Eclipse

3.3.2.6 Eclipse RCP เวอร์ชั่น 2.0.2 ใช้สำหรับพัฒนา Plugin ซึ่งเป็น Plugin ของ Eclipse

3.3.2.7 SISSy เวอร์ชั่น 1.1.0 ใช้สำหรับสกัดแบบจำลอง GAST ซึ่งเป็น Plugin ของ Eclipse

3.4 ขั้นตอนการพัฒนา

ในงานวิจัยนี้ได้แบ่งขั้นตอนของการพัฒนาออกเป็น 2 ส่วน คือ

3.4.1 ส่วนของการพัฒนาซอฟต์แวร์แบบ MDA (Model-driven Architecture)

เริ่มจากสร้างเมตาโมเดลที่ใช้สำหรับคำนวณมาตรวัดสำหรับโปรแกรมเชิงวัตถุให้สามารถรองรับกับมาตรวัดที่ได้คัดเลือกจากงานวิจัยที่ผ่านมา จากนั้นนำเมตาโมเดลที่ได้ทำการสร้างขึ้นมานั้นผ่านกระบวนการแปลงแบบจำลอง (Model-to –Model Transformation)เพื่อคำนวณหาค่ามาตรวัดที่ต้องการ

Ref. code: 25595509035084XRS

3.5 วิธีการวัดผล

งานวิจัยนี้ได้ประยุกต์ใช้ MDA (Model-driven Architecture) เพื่อเป็นแนวทางในการแปลงซอร์สโค้ดภาษาโปรแกรมเชิงวัตถุให้เป็นแบบจำลองโดยที่ไม่ขึ้นกับภาษาใดๆ เพื่อช่วยให้นักวิเคราะห์โครงสร้างโปรแกรมสะดวกในการวิเคราะห์โครงสร้างโปรแกรมเชิงวัตถุ โดยงานวิจัยนี้ได้เสนอวิธีการและพัฒนาเครื่องมือที่ช่วยในการวิเคราะห์โครงสร้างโปรแกรมเชิงวัตถุที่รองรับได้มากกว่า 1 ภาษาคือ JAVA และ C++ ซึ่งเครื่องมือนั้นจะเป็นส่วนเสริมของอีคลิปส์ (Eclipse plugin) โดยวิธีการวัดผลนั้นผู้วิจัยจะใช้ผลการทดสอบมาตรวัดซึ่งประกอบไปด้วย มาตรวัดทั้ง 2 กลุ่ม คือ มาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา (Metrics for Maintainability) ในส่วนคุณลักษณะย่อย (Sub characteristics) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) จากมาตรฐาน ISO 9126 และ มาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection) ซึ่งประกอบไปด้วยมาตรวัด Number Of Statement In Method (NOS), Number Of Parameter In Method (PAR), Nested Block Depth (NBD) , Lack Of Cohesion In Method (LCOM) และ Cyclomatic Complexity (VG) และ มาตรวัดสำหรับเช็คเงื่อนไขการเลือกรีแฟคทอริง (Metrics for conditions refactoring enabling) ที่เหมาะสมซึ่งงประกอบไปด้วยมาตรวัด Count of computation of v in path (CU) , Count of assignment of v in path (DU) , Count of predicate of v in path (PU)

ผู้วิจัยได้นำตัวอย่างซอร์สโค้ดโปรแกรมที่พัฒนา 2 ภาษาด้วยกัน คือภาษา java และ C++ จากนั้นได้ให้ผู้เชี่ยวชาญทางด้านภาษา java ซึ่งมีประสบการณ์การทำงาน 5 ปี และผู้เชี่ยวชาญทางด้าภาษา C++ ซึ่งมีประสบการณ์ด้านการทำงาน 3 ปี มาคำนวณหามาตรวัดแต่ละประเภทหลังจากนั้นนำตัวอย่างซอสร์สโค้ดโปรแกรมมาทดสอบกับเครื่องมือของงานวิจัยเพื่อทำการเปรียบเทียบผลการทดสอบความแม่นยำในการคำนวณของมาตรวัดระหว่างผู้เชี่ยวชาญและเครื่องมือ

Ref. code: 25595509035084XRS

บทที่ 4
ผลการวิจัยและอภิปรายผล

ในบทนี้จะกล่าวถึงกระบวนการแปลงแบบจำลองเพื่อคำนวณหามาตรวัดทั้ง 2 กลุ่ม คือ มาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา (Metrics for Maintainability) ในส่วนคุณลักษณะย่อย (Sub characteristics) ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ด (Analyzability) จากมาตรฐาน ISO 9126 และ มาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection) ซึ่งการแปลงแบบจำลองเพื่อหาค่ามาตรวัดดังกล่าวนั้นจะต้องทำทั้งสิ้น 2 กระบวนการคือ 1. ขั้นตอนการแปลงแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ ไปสู่แบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม 2. ขั้นตอนการแปลงแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมไปสู่แบบจำลองเก็บค่ามาตรของโปรแกรม โดยขั้นตอนการแปลงแบบจำลองนั้นจะต้องมีการกำหนดกฏการแปลงทั้ง 2 ขั้นตอน จากนั้นจะกล่าวถึงการทดสอบความเที่ยงตรงของค่ามาตรวัดที่ได้จากการแปลงแบบจำลอง จากนั้นจะทำการสรุปผลการทดลอง

4. นิยามกฎการแปลงแบบจำลอง

เพื่อที่จะให้ได้ผลของการคำนวณมาตรวัดทั้ง 2 กลุ่มนั้น งานวิจัยนี้ได้แบ่งขั้นตอนการแปลงแบบจำลองของงานวิจัยนี้ออกเป็น 2 ขั้นตอนคือ 1. การแปลงแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ ไปสู่แบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมโดยขั้นตอนนี้เป็นการแปลงแบบจำลองเพื่อให้ได้ข้อมูลที่จำเป็นสำหรับการคำนวณหาค่ามาตรวัดต่างๆตามที่ได้กำหนดเอาไว้ในงานวิจัย ซึ่งขั้นตอนนี้จำเป็นจะต้องมีแบบจำลองต้นทางโดยในที่นี้คือแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ซึ่งจะมาจากการแปลงซอร์สโค้ดโปรแกรมที่อยู่ในรูปของแบบจำลองของโปรแกรมซึ่งจะอธิบายในหัวข้อถัดไป 2. ขั้นตอนการแปลงแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมที่ได้จากขั้นตอนก่อนหน้านี้ไปสู่แบบจำลองเก็บค่ามาตรของโปรแกรม โดยขั้นตอนนี้จะเป็นการแปลงแบบจำลองเพื่อคำนวณหาค่ามาตรวัดแต่ละประเภท ตามที่ได้กำหนดเอาไว้ในงานวิจัยนี้

การแปลงแบบจำลองนั้นจะต้องอาศัยกฎสำหรับการแปลง (Transformation Rule) เพื่อเป็นการกำหนดเงื่อนไขของการแปลงจากแบบจำลองต้นทางไปสู่แบบจำลองปลายทาง ซึ่งการสร้างกฏการแปลงแบบจำลองนั้นจะมาจากการทำกระบวนการจับคู่แบบจำลอง (Mapping) โดยเป็นการจับคู่ระหว่างเอนทิตี้ (Entity) และแอตทริบิวต์ (Attribute) ของเมตาโมเดลต้นทางกับเมตาโมเดลปลายทาง ซึ่งการจับคู่แบบจำลองระหว่างเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ (GASTM : General Abstract Syntax Tree meta-

Ref. code: 25595509035084XRS

model) กับเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม (PDG : Program Dependency Graph Meta-model) ที่ผู้วิจัยได้ออกแบบไว้ในหัวข้อที่ 3.2.2 นั้นแสดงดังตารางที่ 4.1

ตารางที่ 4.1

การจับคู่ระหว่างเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์กับเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม

เมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์	เมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม
Class	MClass
- Method	- MControlFlowGraph
- Constructor	- MControlFlowGraph
- Statement	- AbstractNode
- LoopStatement	- IterativeNode
- BranchStatement	- ConditionNode
- SimpleStatement	- Node
- JumpStatement	- Node
- Field	- Var
- LocalVariable	- Var
- FormalParameter	- Param

จากตารางที่ 4.1 แสดงการจับคู่ระหว่างเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ กับเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม โดยการจับคู่นั้นจะเป็นการจับคู่ที่มีลักษณะที่เป็นโปรแกรมเชิงวัตถุซึ่งเริ่มต้นด้วยเอนทิตี้ Class จะจับคู่กับเอนทิตี้ MClass ซึ่งภายในเอ็นทิตี้ Class จะประกอบไปด้วยเอนทิตี้ Method เอนทิตี้ Constructor และเอนทิตี้ Field สำหรับเอนทิตี้ที่ระดับเมธอดนั้นจะเป็นเอนทิตี้ที่

Ref. code: 25595509035084XRS

เกี่ยวกับ Statement ต่างๆ เช่น if/then for while เป็นต้น โดยประกอบไปด้วย เอนทิตี้ Statement เอนทิตี้ LoopStatement เอนทิตี้ BranchStatement เอนทิตี้ SimpleStatement และ เอนทิตี้ JumpStatement จะจับคู่กับเอนทิตี้ AbstractNode เอนทิตี้ IterativeNode เอนทิตี้ ConditionNode และ เอนทิตี้ Node ตามลำดับ โดยมีเอนทิตี้ FormalParameter ที่เกี่ยวข้องกับพารามิเตอร์ของ เมธอดซึ่งจะจับคู่กับเอนทิตี้ Var

เมื่อเราพิจารณาการจับคู่เอนทิตี้ระหว่างเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์กับเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม นั้นจะพบว่ามีด้วยกัน 11 คู่ ดังนั้นกฎการแปลงแบบจำลองระหว่างเมตาโมเดลดังกล่าว มีจำนวนทั้งสิ้น 11 กฎด้วยกัน ในส่วนของกฎการการแปลงแบบจำลองระว่างแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมกับแบบจำลองมาตรวัดนั้น จะมาจากการจับคู่การแปลงแบบจำลองระหว่างเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรมกับเมตาโมเดลมาตรวัดตามที่ผู้วิจัยได้ออกแบบไว้ในหัวข้อที่ 3.2.3 ซึ่งได้แสดงดังตารางที่ 4.2

ตารางที่ 4.2

การจับคู่ระหว่างเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรมกับเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม

เมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม	**เมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม**
MClass	MClass
- MControlFlowGraph	- MMethod
- Var	- Var
- Param	- Var

จากตารางที่ 4.2 การจับคู่การแปลงแบบจำลองระหว่างเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม กับเมตาโมเดลเก็บค่ามาตรวัดนั้นจะเป็นจับคู่การแปลงเพื่อคำนวณหาค่ามาตรวัดของงานวิจัย โดย เอนทิตี้ MClass จะถูกจับคู่กับเอนทิตี้ MClass ในส่วนของเอนทิตี้ MControlFlowGraph เอนทิตี้ Var และเอนทิตี้ Param จะจับคู่กับเอนทิตี้ MMethod กับเอนทิตี้ Var ดังนั้น เมื่อพิจารณาการจับคู่การแปลงแบบจำลองระหว่างเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม กับเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม จากตารางที่ 4.2 นั้นจะพบว่ามีจำนวนทั้งสิ้น 4 กฎด้วยกัน ซึ่งสามารถสรุปกฎการแปลงแบบจำลองดังตารางที่ 4.3

Ref. code: 25595509035084XRS

ตารางที่ 4.3

สรุปจำนวนกฎการแปลงแบบจำลอง

กฎการแปลงแบบจำลอง	จำนวนกฎ
เมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์กับเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม	11
เมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม กับเมตาโมเดลมาตรวัด	4

4.1.1 การแปลงซอร์สโค้ดโปรแกรมเชิงวัตถุให้อยู่ในรูปของแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์

ผู้วิจัยได้เลือกใช้แบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์มาเป็นแบบจำลองต้นทางของงานวิจัยนี้ โดยแบบจำลองดังกล่าวนั้นจะมาจากการแปลงซอร์สโค้ดโปรแกรมเชิงวัตถุให้อยู่ในรูปของแบบจำลอง (Model Extractor) ซึ่งเป็นกระบวนการทำวิศวกรรมย้อนกลับ (Reverse Engineering) โดยการนำซอร์สโค้ดโปรแกรมเชิงวัตถุที่มีอยู่นั้นมาทำการจำแนกออกมาให้อยู่ในรูปของโครงสร้างต้นไม้วากยสัมพันธ์ (AST: Abstract Syntax Tree) ซึ่งทำให้สามารถมองเห็นโครงสร้างต่างๆของซอร์สโค้ด โดยขั้นตอนการแปลงซอร์สโค้ดไปสู่แบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์นั้นผู้วิจัยได้ใช้เครื่องมือ Sissy (SISSy , 2011) ซึ่งสามารถแปลงซอร์สโค้ดโปรแกรมเชิงวัตถุให้ออกมาอยู่ในรูปแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ที่มีนามสกุลเอ็กซ์เอ็มไอ (XMI: Metadata Information Interchange) โดยภาพที่ 4.1 แสดงตัวอย่างเมธอด statement ที่ได้พัฒนามาจากภาษาจาวา และภาพที่ 4.2 แสดงผลลัพธ์จากการแปลงแบบจำลองด้วย Sissy โดยแบบจำลองที่ได้จากขั้นตอนนี้จะเป็นแบบจำลองตั้งต้นที่ใช้สำหรับทำขั้นตอนการแปลงแบบจำลองระหว่างแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์กับแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม

Ref. code: 25595509035084XRS

```
public String statement () {
 1.      double  totalAmount  = 0;
 2.      int renterPoints = 0;
 3.      Enumeration rentals = _rentals.elements();
 4.      String result = "Rental Record for " + getName() + "\n";
 5.      while (rentals.hasMoreElements()) {
 6.              Rental each = (Rental) rentals.nextElement();
 7.              double thisAmount = 0;
 8.          if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE))
 9.                      renterPoints = renterPoints +2;
           else
10.                      renterPoints ++;
11.                  result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(thisAmount) + "\n";
12.                  totalAmount = totalAmount + thisAmount;
               }
13.              result += "Amount owed is" + String.valueOf(totalAmount)+ "\n";
14.              result += "You earned " + String.valueOf( renterPoints) + " frequent renter points";
15.              return result;
     }
```

ภาพที่ 4.1 เมธอด statement ที่สร้างโดยภาษาจาวา

```
Method statement():java.lang.String
    Position Customer.java Line: 18
    Declaration Type Access _jpzEn_SkEeSkRLfVvphzGQ java.lang.String
    Local Variable thisAmount (_jqdy_PSkEeSkRLfVvphzGQ)
    Local Variable each (_jqdy_vSkEeSkRLfVvphzGQ)
    Local Variable result (_jqdzAPSkEeSkRLfVvphzGQ)
    Local Variable rentals (_jqdzAvSkEeSkRLfVvphzGQ)
    Local Variable renterPoints (_jqdzBPSkEeSkRLfVvphzGQ)
    Local Variable totalAmount (_jqdzBvSkEeSkRLfVvphzGQ)
    Block Statement _jqFYfvSkEeSkRLfVvphzGQ
        Position Customer.java Line: 18
        Simple Statement _jqFYf_SkEeSkRLfVvphzGQ
            Position Customer.java Line: 19
            Variable Access _jq8UlfSkEeSkRLfVvphzGQ totalAmount
                Position Customer.java Line: 19
        Simple Statement _jqFYgPSkEeSkRLfVvphzGQ
            Position Customer.java Line: 20
            Variable Access _jq8UlvSkEeSkRLfVvphzGQ renterPoints
                Position Customer.java Line: 20
        Simple Statement _jqFYgfSkEeSkRLfVvphzGQ
            Position Customer.java Line: 21
            Variable Access _jq8UI_SkEeSkRLfVvphzGQ rentals
                Position Customer.java Line: 21
            Self Access _jq8UJPSkEeSkRLfVvphzGQ <self>
            Variable Access _jq8UJfSkEeSkRLfVvphzGQ _rentals
                Position Customer.java Line: 21
            Function Access _jq8UJvSkEeSkRLfVvphzGQ elements
                Position Customer.java Line: 21
        Simple Statement _jqFYgvSkEeSkRLfVvphzGQ
            Position Customer.java Line: 22
            Variable Access _jq8UJ_SkEeSkRLfVvphzGQ result
                Position Customer.java Line: 22
            Self Access _jq8UKPSkEeSkRLfVvphzGQ <self>
            Function Access _jq8UKfSkEeSkRLfVvphzGQ getName
        Loop Statement _jqFYg_SkEeSkRLfVvphzGQ
        Simple Statement _jqFYjPSkEeSkRLfVvphzGQ
        Simple Statement _jqFYjfSkEeSkRLfVvphzGQ
        Jump Statement _jqFYjvSkEeSkRLfVvphzGQ
Inheritance Type Access  java.lang.Object
```

ภาพที่ 4.2 แสดงแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ของเมธอด statement

Ref. code: 25595509035084XRS

4.1.2 การสร้างกฎการแปลงแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ไปสู่แบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม

การแปลงแบบจำลองในขั้นตอนนี้จะเป็นการแปลงแบบจำลองเพื่อจัดเตรียมข้อมูลสำคัญต่างๆ ที่จำเป็นสำหรับการคำนวนเพื่อหาค่ามาตรวัดตามที่ได้กำหนดไว้ในงานวิจัย โดยก่อนการแปลงแบบจำลองนั้นจะต้องทำการจับคู่แบบจำลองที่ต้องการแปลงจำลองจากนั้นจึงจะได้กฎการแปลงแบบจำลองซึ่งกฎการแปลงแบบจำลองระหว่างแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ ไปสู่แบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมนั้นมีทั้งสิ้น 11 กฎ ด้วยกัน ดังที่ได้สรุปไว้ในตารางที่ 4.3 สำหรับการสร้างกฎการแปลงแบบจำลองนั้นจะต้องอาศัยภาษาสำหรับสร้างกฎการแปลงแบบจำลอง โดย งานวิจัยนี้ได้ใช้ภาษาสำหรับการแปลงแบบจำลองด้วยเอทีแอล (ATL Transformation Language) ดังที่ได้กล่าวไว้ในบทที่ 2 จากการจับคู่การแปลงแบบจำลองเพื่อกำหนดกฎการแปลงแบบจำลองของขั้นตอนนี้นั้นจากตารางที่ 4.1 ซึ่งได้เอนทิตี้ที่ใช้สำหรับการแปลงแบบจำลองระหว่างแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์กับแบบจำลองกราฟแสดงความสัมพันธ์ โดยสามารถสร้างกฎการแปลงแบบจำลองได้ดังนี้

4.1.2.1 กฎ ClassToMClass

กฎ ClassToMClass เป็นกฎการแปลงเอนทิตี้ GASTClass กับเอนทิตี้ MCLass โดย กฎ ClassToMClass เป็นกฎที่เปรียบเสมือนคลาสของโปรแกรมเชิงวัตถุ ซึ่งภายในคลาสนั้นจะประกอบไปด้วยแอตทริบิวต์ (Attribute) และเมธอด (Method) ของคลาส โดยภายในกฎ ClassToMClass นั้นมีการแปลงแอตทริบิวต์ต่างๆซึ่งประกอบไปด้วย Methods และ Constructor ซึ่งในที่นี้จะถือว่าเป็นเมธอดประเภทหนึ่งที่อยู่ในคลาสซึ่งมาจากเฮลเปอร์ getAllMethodAndConstructor ซึ่งจะรวบรวมเมธอดและคอนสตรัคเตอร์ที่อยู่ภายในคลาส โดยไม่รวม ดีสตรัคเตอร์ (Destructor) และ ระดับการเข้าถึง(Access Modifiers) นอกจากนี้จะมีการแปลงฟิลด์ (Field) ที่อยู่ในคลาสอีกด้วย โดย กฎ ClassToMClass แสดงดังภาพที่ 4.3

Ref. code: 25595509035084XRS

```
rule ClassToMClass {
    from
        s: GAST!GASTClass
    to
        t: PDG!MClass (
            name <- s.simpleName,
            Method <- s.getAllMethodAndConstructor,
            instatVars <- s.fields
        )

}
```

ภาพที่ 4.3 แสดงกฎ ClassToMClass

4.1.2.2 กฎ MethodToMControlFlowGraph

กฎ MethodToMControlFlowGraph เป็นกฎการแปลงเมธอดที่อยู่ภายในคลาส โดยจะแปลงเอนทิตี้ Method ไปสู่เอนทิตี้ MControlFlowGraph โดยภายในเมธอดจะประกอบไปด้วยตัวแปรชั่วคราว และ Statement ต่างๆที่แสดงถึงลำดับการไหลของ Statement หนึ่งไปสู่อีก Statement หนึ่ง ซึ่งในที่นี้คือแอตทริบิวต์ localVar และแอตทริบิวต์ nodes ตามลำดับ รวมไปถึงจำนวนพารามิเตอร์ที่อยู่ภายในเมธอดที่เป็นการแปลงระหว่างแอตทริบิวต์ param กับแอตทริบิวต์ formalParameter โดยรายละเอียดของกฎ MethodToMControlFlowGraph แสดงดังภาพที่ 4.4

```
rule MethodToMControlFlowGraph {
    from
        s: GAST!Method (
            not s.body.oclIsUndefined()
        )
    to
        t: PDG!MControlFlowGraph (
            name <- s.simpleName,
            nodes <- if not s.body.oclIsUndefined() then
                        s.body.statements
                    else
                        s.body
                    endif,
            localVar <- s.localVariables,
            param <- s.formalParameters
            )
}
```

ภาพที่ 4.4 แสดงกฎ MethodToMControlFlowGraph

4.1.2.3 กฎ ConstructorToMControlFlowGraph

กฎ ConstructorToMControlFlowGraph เป็นกฎการแปลงที่เกี่ยวข้องกับ Constructor ของคลาสโดยไม่นับรวม Destructor ของภาษา C++ โดยทั่วไปแล้ว Constructor นั้นจะมีลักษณะการแปลงที่คล้ายคลึงกับกฎ MethodToControlFlowGraph ไม่ว่าจะเป็น การแปลงเอนทิตี้ Statement ไปสู่เอนทิตี้ node รวมไปถึงกฎการแปลง localVar และ กฎ param อีกด้วย โดยรายละเอียดของกฎ ConstructorToMControlFlowGraph แสดงดังภาพที่ 4.5

```
rule ConstructorToMControlFlowGraph {
    from
        s: GAST!Constructor (
            not s.body.oclIsUndefined()
        )
    to
        t: PDG!MControlFlowGraph (
            name <- s.simpleName,
            nodes <- if not s.body.oclIsUndefined() then
                    s.body.statements
                else
                    s.body
                endif,
            localVar <- s.localVariables,
            param <- s.formalParameters
        )
}
```

ภาพที่ 4.5 แสดงการแปลงกฎ ConstructorToMControlFlowGraph

4.1.2.4 กฎ StatementToAbstractNode

กฎ StatementToNode เป็นกฎที่ใช้แปลง Statement ต่างๆที่อยู่ภายในเมธอด ไม่ว่าจะเป็น Statement ประเภท if/then else หรือ return เป็นต้น โดยเอนทิตี้ Statement จะเป็นเอนทิตี้ประเภทแอพสแตรค (Abstract) ซึ่งจะแปลงไปสู่เอนทิตี้ AbstractNode ซึ่งมีชนิดเป็นแอพสแตรคเช่นเดียวกับ Statement โดยกฎ StatementToNode แสดงดังภาพที่ 4.6

```
abstract rule StatementToNode {
    from
        s: GAST!Statement (
            not s.oclIsKindOf(GAST!BlockStatement)
        )
    to
        t: PDG!AbstractNode (

         )
 }
```

ภาพที่ 4.6 แสดงกฎ StatementToNode

4.1.2.5 กฎ SimpleStatementToNode

กฎ SimpleStatementToNode เป็นกฎที่ใช้แปลงเอนทิตี้ที่เกี่ยวข้องกับ Statement ทั่วไปที่อยู่ภายในเมธอด โดยกฎ SimpleStatementToNode จะเป็นการแปลงระหว่างเอนทิตี้ SimpleStatement ไปสู่เอนทิตี้ Node กฎ SimpleStatementToNode จะเป็นการสืบทอดมาจากกฎ StatemenToAbstractNode ทำให้ได้รับคุณลักษณะการแปลงจากกฎดังกล่าวด้วย นอกจากนี้ยังมีการแปลงเอนทิตี้ที่สำคัญคือ เอนทิตี้ toNode ซึ่งจะเป็นการแสดงให้เห็นถึงลำดับการไหลของ Statement (Control flow) ว่า Statement ก่อนหน้าคือ Statement ใด และ Statement ต่อไปคือ Statement ใด โดย กฎ SimpleStatementToNode แสดงดังภาพที่ 4.7

Ref. code: 25595509035084XRS

```
rule SimpleStatementToNode extends StatementToNode {
    from
        s: GAST!SimpleStatement(s.chStmtIsInLoopCondition.size() = 0)
    to
        t: PDG!Node (
            toNode <- if s.oclIsUndefined() then
                    s.refImmediateComposite().refImmediateComposite()
                else
                    s.next
                endif,
            type <- if s.accesses.size() > 2 then
                    if not s.accesses.first().oclIsUndefined() and s.accesses.first().
                            oclIsKindOf(GAST!VariableAccess) then
                        if s.accesses.first().accessedTarget.typeDeclaration.
                                accessedTarget.simpleName = 'int' or s.accesses.first().
                                accessedTarget.typeDeclaration.accessedTarget.simpleName
                                = 'double' or s.accesses.first().accessedTarget.
                                typeDeclaration.accessedTarget.simpleName = 'float' then
                            'true'
                        else
                            'false'
                        endif
                    else
                        'false'
                    endif
                else
                    'false'
                endif
        )
```

ภาพที่ 4.7 แสดงกฎ SimpleStatementToNode

4.1.2.6 กฎ LoopStatmentToIterativeNode

กฎ LoopStatmentToIterativeNode เป็นการแปลงเอนทิตี้ LoopStatement ไปสู่เอนทิตี้ IterativeNode กฎ LoopStatmentToIterativeNode นั้นจะเป็นการแปลงของ Statement ที่ประเภท Loop ต่างๆของโปรแกรมเชิงวัตถุ เช่น for while เป็นต้น กฎ LoopStatmentToIterativeNode จะมีแอตทริบิวต์ที่ประกอบด้วย trueConditionNode ซึ่งจะแสดงลำดับการไหลของ Statement ถัดไปเมื่อเงื่อนไขเป็นจริง (True) และ falseConditionNode แสดงให้เห็นถึงลำดับการไหลของ Statement ถัดไปเมื่อเงื่อนไขเป็นเท็จ (False) นอกจากนี้ ภายในกฎ LoopStatmentToIterativeNode ยังมีการแปลงแอตทริบิวต์ nodes ซึ่งเป็น Statement ย่อยของ Statement ประเภท Loop นั้นๆ โดยภาพรวมของกฎ LoopToIterativeNode สามารถแสดงดังภาพที่ 4.8

```
rule LoopStatement extends StatementToNode  {
    from
        s: GAST!LoopStatement
    to
        t: PDG!IterativeNode (
            trueConditionNode <- s.body.statements.first(),
            falseConditionNode <- s.next,
            name <- s.kind.toString(),
            nodes <- s.body.statements
        )

}
```

ภาพที่ 4.8 กฎ LoopStatementToIterativeNode

4.1.2.7 กฎ BranchStatementToConditionNode

กฎ BranchStatementToConditionNode เป็นกฎการแปลงเอนทิตี้ BranchStatement ไปสู่เอนทิตี้ ConditionNode กฎ BranchStatementToConditionNode เป็นการแปลง Statement ประเภทเงื่อนไขต่างๆ เช่น if/then หรือ switch เป็นต้น โดยภายในกฎนั้นมีการแปลงแอตทริบิวต์ trueConditionNode เพื่อแสดง Statement ลำดับถัดไป เมื่อเงื่อนไขนั้นเป็นจริง และ falseConditioNode ที่แสดง Statement ลำดับถัดไปเมื่อเงื่นไขนั้นเป็นเท็จซึ่งลำดับการไหลดังกล่าวนี้จะไม่รวมในส่วนของ switch รวมไปถึง nodes ที่แสดงให้เห็นถึง Statement ทั้งหมดที่อยู่ภายใน Statement ประเภทเงื่อนไขนั้นๆ กฎ BranchStatementToConditionNode แสดงดังภาพที่ 4.9

```
rule BranchStatement extends StatementToNode{
    from
        s: GAST!BranchStatement
    to
        t: PDG!ConditionalNode (
 trueConditionNode <- s.branches.first().statement,
             falseConditionNode <-   s.branches.last().statement   ,
            nodes <- s.branches -> iterate(e1; acc1: Sequence(GAST!Statement) =Sequence{} |
            if e1.statement.oclIsKindOf(GAST!Statement) then
                        if e1.statement.oclIsKindOf(GAST!BlockStatement) then
                            acc1.append(e1.statement.statements)
                        else
                            if e1.statement.oclIsKindOf(GAST!BranchStatement) then
                                if not e1.statement.oclIsUndefined() then
                                    acc1.append(e1.statement)
                                else
                                    acc1.append(e1.statement)
                                endif
                            else
                                acc1.append(e1.statement)
                            endif
                        endif
            else
                acc1
            endif

            )
        )
}
```

ภาพที่ 4.9 กฎ BranchStatementToConditionNode

4.1.2.8 กฎ JumpStatementToNode

กฎ JumpStatementToNode เป็นการแปลงเอนทิตี้ JumpStatement ของเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์ ไปสู่เอนทิตี้ Node ของเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม กฎการแปลงนี้จะเกี่ยวข้องกับ Statement ที่มีลักษะการกระโดด เช่น Return หรือ Break เป็นต้น ซึ่งภายในกฎมีการแปลงแอตทริบิวต์ toNode เพื่อแสดง Statement ลำดับถัดไปที่ต่อจาก JumpStatement นี้ โดยรายละเอียดของกฎ JumpStatementToNode แสดงดังภาพที่ 4.10

```
rule jumpStatement extends StatementToNode {
    from
        s: GAST!JumpStatement
    to
        t: PDG!Node (
            toNode <- if s.oclIsUndefined() then
                    s.refImmediateComposite().refImmediateComposite()
                else
                    s.next
                endif,
            type <- if s.accesses.size() > 2 then
                    if not s.accesses.first().oclIsUndefined() and s.accesses.first().
                            oclIsKindOf(GAST!VariableAccess) then
                        if s.accesses.first().accessedTarget.typeDeclaration.
                                accessedTarget.simpleName = 'int' or s.accesses.first().
                                accessedTarget.typeDeclaration.accessedTarget.simpleName
                                = 'double' or s.accesses.first().accessedTarget.
                                typeDeclaration.accessedTarget.simpleName = 'float' then
                            'true'
                        else
                            'false'
                        endif
                    else
                        'false'
                    endif
                else
                    'false'
                endif
        )
}
```

ภาพที่ 4.10 กฎ JumpStatementToNode

4.1.2.9 กฎ FormalParamToParam

กฎ FormalParamToParam เป็นการแปลงเอนทิตี้ FormalParameter ไปสู่อิลิเม้น Param โดยกฎนี้จะทำหน้าที่แปลงพารามิเตอร์ที่เกี่ยวข้องกับเมธอดนั้นๆ โดยรายละเอียดของกฎ FormalParamToParam แสดงดังภาพที่ 4.11

```
rule FormalParam {
    from
        s: GAST!FormalParameter
    to
        t: PDG!Param (
            name <- s.simpleName
        )


}
```

ภาพที่ 4.11 กฎ FormalParamToParam

4.1.2.10 กฎ InstanceToVar

กฎ InstanceToVar เป็นการแปลงเอนทิตี้ Field ไปสู่เอนทิตี้ Var โดย กฎ InstanceToVar เป็นการแปลงแอตทริบิวต์ประเภทอินสแตนท์ที่อยู่ภายในคลาส โดยภายในกฎมีการแปลงแอตทริบิวต์ Type ซึ่งเป็นระดับการเข้าถึงของแอตทริบิวต์ accessType ระดับการเข้าถึงของแอตทริบิวต์ รวมไปถึง accessMethod ที่เป็นแอตทริบิวต์แสดงเมธอดทั้งหมดที่อินสแตนท์ของคลาสนั้นเข้าถึง โดยรายละเอียดของกฎ InstanceToVar แสดงดังภาพที่ 4.12

```
rule InstntToVar {
    from
        s: GAST!Field
    to
        t: PDG!Var (
            name <- s.simpleName,
            type <- s.typeDeclaration.accessedTarget.simpleName,
            isInstant <- 'true',
            accessType <- s.visibility.toString(),
            isStatic <- s.static
            ,accessMethod <- thisModule.getAllClass(s)
        )

}
```

ภาพที่ 4.12 แสดงกฎ InstantToVar

4.1.2.11 กฎ LocalVariableToVar

กฎ LocalVariableToVar เป็นการแปลงเอนทิตี้ LocalVariable ระดับการเข้าถึงไปสู่เอนทิตี้ Var โดย กฎ LocalVariableToVar เป็นการแปลงแอตทริบิวต์ที่เป็นตัวแปรชั่วคราวที่อยู่ภายในเมธอด โดยภายในกฎมีการแปลงแอททริบิวต์ที่สำคัญซึ่งประกอบไปด้วย du ซึ่งเป็นการค้นหา Statement ที่อยู่ภายในเมธอดว่ามี Statement ใดบ้างที่เป็น Statement ที่มีการกำหนดค่าของตัวแปร pu เป็นการค้นหา Statement ทั้งหมดที่อยู่ภายในเมธอดโดยจะเจาะจงไปที่ Statement ประเภทนิพจน์ เงื่อนไข cu จะเป็นการค้นหา Statement ทั้งหมดที่อยู่ภายในเมธอดโดยที่มีเงื่อนไขคือภายใน Statement นั้นต้องมีการเข้าถึงจากตัวแปรภายในที่มีมากว่า 1 ตัว โดยรายละเอียดของกฎ LocalVariableToVar แสดงดังภาพที่ 4.13

Ref. code: 25595509035084XRS

```
rule LocalVariableToVar {
    from
        s: GAST!LocalVariable
    to
        t: PDG!Var (
            name <- s.simpleName,
            type <- s.typeDeclaration.accessedTarget.simpleName,
            isLocalVar <- 'true',
            du <- thisModule.getNext(s)->reject(r | not r.oclIsKindOf(GAST!Statement))->
            iterate(e; accS: Sequence(GAST!Statement) = Sequence{} |
                    if e.accesses.first().write = true then
                        accS.append(e)
                    else
                        accS
                    endif  ).size(),
            pu <- thisModule.getNext(s).flatten()->select(e | e.oclIsKindOf(GAST!BranchStatement)).size(),
            cu <-   thisModule.getNext(s)->reject(p | p.oclIsTypeOf(GAST!BranchStatement))->
            reject(a | not a.oclIsKindOf(GAST!Statement))->
            iterate(e; accS: Sequence(GAST!Statement) = Sequence{} |
                if e.accesses->select(x | x.oclIsTypeOf(GAST!VariableAccess)).size() > 1 then
                        accS.append(e)
                else
                    accS
                endif).size()
        )
}
```

ภาพที่ 4.13 แสดงกฎ LocalVariableToVar

4.1.3 การแปลงแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรมไปสู่แบบจำลองเก็บค่ามาตรวัดของโปรแกรม

หลังจากทำกระบวนการแปลงแบบจำลองระหว่างแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์ไปสู่แบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม จากหัวข้อที่ผ่านมาซึ่งจะได้ข้อมูลพื้นฐานที่จำเป็นสำหรับการคำนวณค่ามาตรวัดของงานวิจัย โดยหัวข้อนี้จะเป็นการนำผลลัพธ์ที่ได้จากขั้นตอนดังกล่าว มาทำการแปลงแบบจำลองเพื่อคำนวณหามาตรวัดของงานวิจัย ด้วยกระบวนการแปลงแบบจำลองกราฟแสดงความสัมพันธ์ของโปรแกรม ไปสู่แบบจำลองเก็บค่ามาตรวัดของโปรแกรม โดยกฎการแปลงนั้นจะมาจากการจับคู่การแปลงระหว่างเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรมกับเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม ซึ่งมีทั้งสิ้น 4 กฎตามที่ได้สรุปไว้ดังตารางที่ 4.3 ซึ่งกฎการแปลงแบบจำลองระหว่างกราฟแสดงความสัมพันธ์ของโปรแกรมไปสู่แบบจำลองเก็บค่ามาตรวัดของโปรแกรม มีรายละเอียด ดังนี้

Ref. code: 25595509035084XRS

4.1.3.1 กฎ MClassToMClass

กฎ MClassToMClass จะเป็นการแปลงเอนทิตี้ MClass ไปสู่แบบจำลอง MClass ลักษณะการทำงานของกฎ MClassToMCLass นั้นจะเป็นการคำนวนมาตรวัดในระดับ Class คือมาตรวัด Lack of cohesion in method (LCOM) โดยมีวิธีคำนวณจากสูตร

สูตรการคำนวณ :

$$\frac{\left[\frac{1}{v}\sum_{i=1}^{v} m(v_i)\right]-m}{1-m} \tag{4.1}$$

เมื่อ

m = จำนวนเมธอด

v = จำนวนแอตทริบิวต์

$m(v_i)$ = เมธอดที่เรียกใช้แอตทริบิวต์ v_i

การคำนวณหาค่ามาตรวัด LCOM นั้นจะทำผ่าน เฮลเปอร์ getLCOM ซึ่งแสดงดังภาพที่ 4.14 โดยค่า v จะหาจาก self.instanceVar ซึ่งเป็นจำนวนของ Instance variable ที่อยู่ภายในคลาส ส่วนค่า m หาได้จาก self.Method.size() ซึ่งเป็นจำนวนเมธอดที่อยู่ภายในคลาส และ $m(v_i)$ คือเมธอดที่เรียกใช้แอตทริบิวต์ สามารถหาได้จากการค้นหาโดยเฮลเปอร์ getNumberMethodUseInstance ดังภาพที่ 4.15 เมื่อนำค่าดังกล่าวแทนค่าตัวแปรในสมการที่ 4.1 จะทำให้ได้ค่ามาตรวัด LCOM ของงานวิจัย โดยกฎ MClassToMClass นั้นแสดงดังภาพที่ 4.16

```
helper context PDG!MClass def : getLCOM : Real =

    let lcom : Real = 0
    in
    ((((1/self.instatVars.size())*self.getNumberMethodUseInstant.flatten().size())
    -self.Method.size()) / (1-self.Method.size())

);
```

ภาพที่ 4.14 เฮลเปอร์ getLCOM

```
helper context PDG!MClass def : getNumberMethodUseInstant : Sequence(PDG!MControlFlowGraph) =

self.instatVars.asSequence()->iterate(e; acc: Sequence(PDG!MControlFlowGraph) = Sequence{} |

        acc.append(e.accessMethod)

);
```

ภาพที่ 4.15 เฮลเปอร์ getNumberMethodUseInstant

```
rule MClass2MClass {
    from
        s : PDG!MClass
    to
        t : MTRIX!MClass (
            name <- s.name,
            methods <- s.Method
            ,
            LCOM <- s.getLCOM.toString()

        )


}
```

ภาพที่ 4.16 กฎ MClassToMClass

4.1.3.2 กฎ MMethodToMMethod

กฎ MMethodToMMethod จะเป็นการแปลงเอนทิตี้ MMethod ไปสู่เอนทิตี้ MMethod ลักษณะการทำงานของกฎ MMethodToMMethod นั้นจะเป็นการคำนวณมาตรวัดที่เกี่ยวข้องกับเมธอดซึ่งประกอบไปด้วยมาตรวัด MacCabe Cycromatic Complexity (VG) มาตรวัด Nested block depth (NBD) มาตรวัด Number of parameter (PAR) และ Number Of Statement In Method (NOS) โดยมาตรวัดดังกล่าวสามารถแปลงแอตทริบิวต์ตามสมการ ดังนี้

McCabe Cyclomatic Complexity (VG)

สูตรการคำนวณ :

$$VG = P + 1 \tag{4.2}$$

เมื่อ

VG = ค่าความซับซ้อน

P = จำนวนของของ Statement ที่เป็นทางเลือกเงื่อนไข

จากสมการที่ 4.2 นั้นจะทำการค้นหาจำนวนของ node ที่เป็นทางเลือกเงื่อนไขจาก ConditionNode และ IterativeNode ที่อยู่ในเมธอดโดยในส่วนของ Switch statement ซึ่งจะนับตามจำนวนของ case ที่เกิดขึ้นแทน ซึ่งเมื่อนำค่าดังกล่าวไปแทนในสมการที่ 4.2 จะทำให้ได้ค่าของมาตรวัด VG ของงานวิจัย โดยแสดงดังภาพ 4.17

```
helper context PDG!MControlFlowGraph def :getVG :Integer =

    self.getNodeFromCFG.flatten()->reject(r | r.oclIsTypeOf(PDG!Node))->

    iterate(n; acc:  Integer = 0 |
        if n.oclIsTypeOf(PDG!ConditionalNode) then
            if n.nodes.size() > 2 then  -- switch case to countingcase
                acc + (n.nodes.size())
            else
                acc + 1  -- if/then
            endif
        else
                acc + 1 -- // loop
        endif
)+1;
```

ภาพที่ 4.17 เฮลเปอร์ getVG

Number Of Statement In Method (NOS)

สูตรการคำนวณ :

NOS = จำนวนของ Statement ทั้งหมดภายในเมธอด (4.3)

NOS สามารถหาโดยการนับจำนวน node ทั้งหมดที่อยู่ภายในเมธอด โดยรวมทั้ง node ที่อยู่ภายใต้ ConditionNode และ IterativeNode โดยวิธีการหาค่าของ NOS นั้นแสดงดังภาพที่ 4.28 ซึ่งเป็นการวนรอบหา AbstarctNode ครั้งละรอบเมื่อพบว่าเป็น Node จะถูกเพิ่มไว้ที่ Sequence แต่ถ้าพบว่า เป็น IterativeNode หรือ ConditionNode นั้น จะทำการวนรอบหา Node ที่อยู่ภายใน IterativeNode หรือ ConditionNode นั้นๆจนกระทั่งสิ้นสุด

```
helper context PDG!MControlFlowGraph def : getNOS()  : Sequence(PDG!AbstractNode) =
    self.nodes.asSequence()->iterate(e; acc: Sequence(PDG!AbstractNode) = Sequence{} |

    if e.oclIsTypeOf(PDG!Node) then
        acc.append(e)
    else
        if e.oclIsTypeOf(PDG!IterativeNode) then
            acc.append(e).append(e.getNodeFromIterative)
        else
            if e.oclIsTypeOf(PDG!ConditionalNode) then
                acc.append(e).append(e.getNodeFromCondition)
            else
                acc.append(e)
                endif
        endif
    endif

)->flatten().size();
```

ภาพที่ 4.18 เฮลเปอร์ getNOS

Nest Block Depth (NBD)

สูตรการคำนวณ :

NBD = จำนวณความลึกของลำดับชั้นปีกกาภายในเมธอด (4.4)

NBD สามารถหาได้จากการวนรอบหาชั้นที่ลึกที่สุดของ node ประเภท IterativeNode และ ConditionNode ดังภาพที่ 4.19 คือ เฮลเปอร์ getNBD ที่ใช้สำหรับหาค่าความลึกของลำดับชั้น ซึ่งจะวนรอบหา ConditionNode และ IterativeNode ของ Node ทั้งหมดที่อยู่ในเมธอด หากพบ ConditionNode หรือ IterativeNode นั้นจะทำการเรียกใช้เฮลเปอร์ getNBDFromNodeAndCondition เพื่อหาลำดับที่ลึกที่สุด จนกระทั่งถึง Node สุดท้าย

Ref. code: 25595509035084XRS

```
helper context PDG!IterativeNode def : getNBDFromLoopAndCondition() : Sequence(PDG!AbstractNode) =
    self.nodes->iterate(e; accLoop: Sequence(PDG!AbstractNode) = Sequence{} |

                if e.oclIsKindOf(PDG!ConditionalNode) or e.oclIsKindOf(PDG!IterativeNode) then
                    if e.oclIsKindOf(PDG!IterativeNode) then
                        accLoop.append(e).append(e.getNBDFromLoopAndCondition())
                    else
                        accLoop.append(e)
                    endif

                else
                    accLoop
                endif
                );


helper context PDG!MControlFlowGraph def : getNBD() : Integer =
    self.nodes->iterate(e; acc: Sequence(PDG!AbstractNode) = Sequence{} |

                if e.oclIsKindOf(PDG!ConditionalNode) or e.oclIsKindOf(PDG!IterativeNode) then
                    if e.oclIsKindOf(PDG!IterativeNode) then
                    acc.append(e).append(e.getNBDFromLoopAndCondition())
                    else
                        acc.append(e)
                    endif
                else
                    acc
                endif
                ).size()
    ;
```

ภาพที่ 4.19 เฮลเปอร์ getNBD

ในส่วนของมาตรวัด PAR ซึ่งเป็นมาตรวัดสำหรับหาจำนวนพารามิเตอร์ในเมธอดนั้น สามารถทำการนับจากแอตทริบิวต์ param ที่อยู่ภายในเมธอดได้เลย ดังนั้นรายละเอียดของกฎ MMethodToMMethod แสดงดังภาพที่ 4.20

```
rule MMethod2MMethod {
    from
        s : PDG!MControlFlowGraph
    to
        t : MTRIX!MMethod (

            name <- s.name,
            NOS <- s.getNOS,
            VG <- s.getVG().toString(),
            NBD <- s.getNBD().toString(),
            PAR <- s.param.size().toString(),
            vars <- s.localVar

}
```

ภาพที่ 4.20 กฎ MMethodToMMethod

4.1.3.3 กฎ VarToVar

กฎ VarToVar จะเป็นการแปลงเอนทิตี้ Var ไปสู่เอนทิตี้ Var โดยลักษณะการทำงานของกฎ VarToVar จะเป็นการแปลงตัวแปรและมาตรวัดสำหรับคำนวณมาตรวัดเพื่อระบุวิธีรีแฟคทอริง ของงานวิจัยนี้ โดยกฎการแปลงแบบจำลอง VarToVar นั้นแสดงดังภาพที่ 4.21

```
rule Var2Var {
    from
        s : PDG!Var
    to
        t : MTRIX!Var (
            name <- s.name,
            type <- if not s.type.oclIsUndefined() then
                        s.type
                    else
                        'notype'
                    endif
        )
}
```

ภาพที่ 4.21 กฎ VarToVar

4.1.3.3 กฎ ParamToVar

กฎ ParamToVar จะเป็นการแปลงเอนทิตี้ Param ไปสู่เอนทิตี้ Var โดยลักษณะการทำงานของกฎ ParamToVar จะเป็นการแปลงพารามิเตอร์ของเมธอด โดยกฎการแปลง ParamToVar แสดงดังภาพที่ 4.22

```
rule Param2Var {
    from
        s : PDG!Param
    to
        t : MTRIX!Param (
            type <- s.type
        )
}
```

ภาพที่ 4.22 กฎ ParamToParam

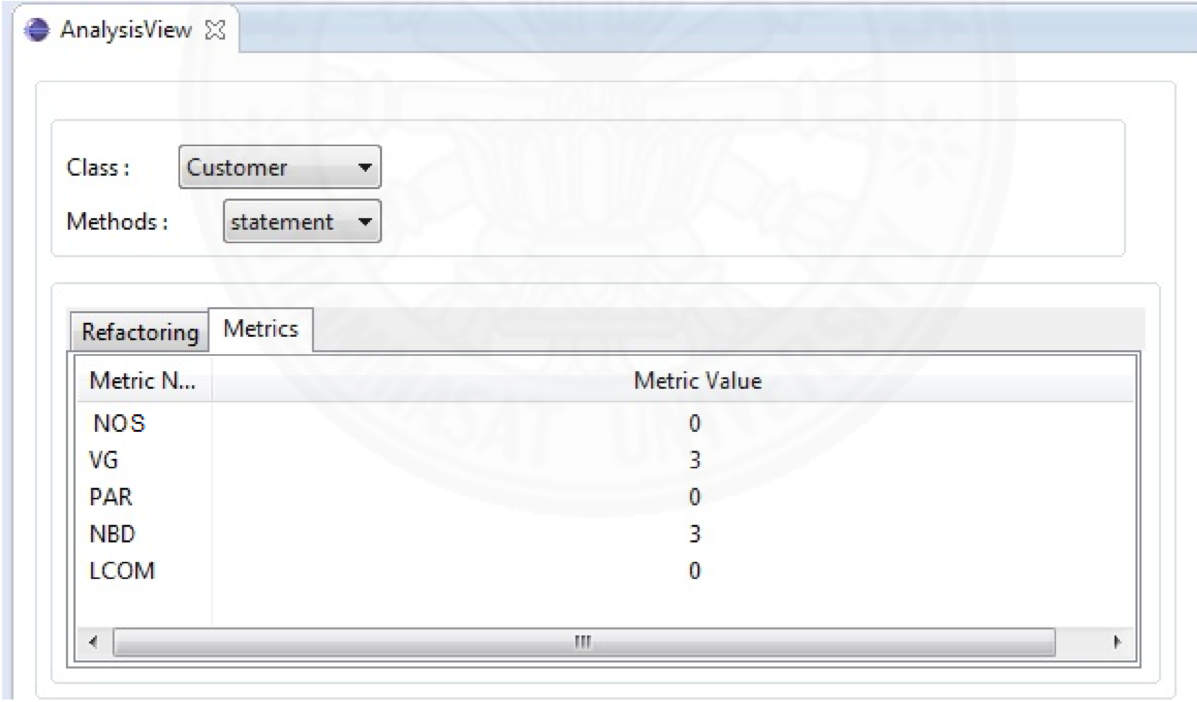
Ref. code: 25595509035084XRS

4.2 การแสดงผลจากส่วนเสริมโปรแกรมอีคลิปส์ (Eclipse-plugin)

งานวิจัยนี้ได้เสนอเครื่องมือช่วย Detect bad smell ชนิด Long method และ Select refactoring ที่เหมาะสมดังนั้นส่วนของการแสดงผลสามารถแบ่งได้ 2 ส่วนคือ ส่วนสำหรับแสดงผลลัพธ์การคำนวณมาตรวัด และส่วนของการแสดงผลลัพธ์การ Select refactoring ที่เหมาะสม

4.2.1 ส่วนแสดงผลลัพธ์การคำนวณมาตรวัด

สำหรับการแสดงผลทางด้านมาตรวัดนั้นจะเกี่ยวข้องกับมาตรวัดที่ได้สำรวจจากงานวิจัยที่ผ่านมา ซึ่งประกอบไปด้วยมาตรวัด Number Of Statement In Method (NOS) มาตรวัด MaCabe Cyclomatic Complexity (VG) มาตรวัด Nest Block Depth (NBD) มาตรวัด Number Of Parameters (PAR) และ มาตรวัด Lack Of Cohesion In Method (LCOM) โดยการแสดงผลของการคำนวณมาตรวัดดังกล่าว แสดงดังภาพที่ 4.23



ภาพที่ 4.23 ส่วนของการแสดงผลมาตรวัด

4.3 การทดสอบ

การทดสอบของงานวิจัยเพื่อเป็นการตรวจสอบว่าเครื่องมือที่ผู้วิจัยได้พัฒนาขึ้นมานั้นมีความแม่นยำ พร้อมทั้งสามารถรองรับได้ทั้ง 2 ภาษาโปรแกรมเชิงวัตถุได้จริงตามที่ได้นำเสนอเอาไว้คือ ภาษา Java และ ภาษา C++

ผู้วิจัยได้ทดสอบกับผู้เชี่ยวชาญ 2 ท่าน โดยแบ่งเป็นผู้ที่มีประสบการ์ณการพัฒนาภาษาโปรแกรมภาษา Java มากกว่า 5 ปี 1 ท่าน และภาษา C++ ที่มีประสบการณ์ทางด้านพัฒนาโปรแกรมมากกว่า 3 ปีหนึ่งท่าน โดยผู้วิจัยทำการอธิบายวิธีการคำนวณมาตรวัดแต่ละประเภทของงานวิจัยนี้ซึ่งแบ่งเป็นมาตรวัด Number Of Statement In Method (NOS) มาตรวัด McCabe Cyclomatic Complexity (VG) มาตรวัด Nested block depth (NBD) มาตรวัด Lack of cohesion in method (LCOM) และมาตรวัด Number of parameter in method (PAR) จากนั้นผู้วิจัยได้นำตัวอย่างซอร์สโค้ดโปรแกรมซึ่งได้พัฒนาออกมา 2 ภาษาโปรแกรมทั้งภาษา Java และ C++ โดยผู้วิจัยได้ให้ผู้เชี่ยวชาญทั้ง 2 ท่านทำการคำนวณมาตรวัดแต่ละประเภทจากนั้นนำผลลัพธ์ที่ได้จากผู้เชี่ยวชาญมาเปรียบเทียบกับผลการคำนวณมาตรวัดแต่ละประเภทของเครื่องมือที่ผู้วิจัยได้พัฒนาขึ้น โดยผลการทดสอบที่ได้ แบ่งเป็นตามประเภทมาตรวัดดังนี้

4.4.1 การทดสอบมาตรวัด Number Of Statement In Method (NOS)

การทดสอบมาตรวัด Number Of Statement In Method (NOS) นั้นผู้วิจัยได้นำตัวอย่างซอร์สโค้ด เมธอด statement() มาใช้ในการทดสอบซึ่งพัฒนาด้วยกัน 2 ภาษาคือ ภาษา Java และ C++ ดังภาพที่ 4.24 และ 4.25 ตามลำดับ

จากภาพที่ 4.24 เป็นตัวอย่างเมธอด statement() ที่ถูกพัฒนาโดยภาษา C++ ซึ่งผู้เชี่ยวชาญสามารถนับจำนวนของ Statement ทั้งหมดนั้นพบว่ามีจำนวนทั้งสิ้น 25 Statement ด้วยกันซึ่งเมื่อนำซอร์สโค้ดดังกล่าวทดสอบกับเครื่องมือของงานวิจัยนี้ พบว่าเครื่องมือสามารถนับจำนวน Statement ได้ที่ 25 Statement เท่ากันกับผู้เชี่ยวชาญ ซึ่งต่างจากตัวอย่างภาษา Java ดังภาพที่ 4.25 ซึ่งผู้เชี่ยวชาญสามารถนับจำนวนของ Statement ทั้งหมดพบว่ามีจำนวนทั้งสิ้น 24 Statement เมื่อนำซอร์สโค้ดดังกล่าวมาทดสอบกับเครื่องมือพบว่าจำนวนของ Statement จะได้เท่ากับการทดสอบของผู้เชี่ยวชาญคือ 24 Statement โดยผลลัพธ์ของการทดสอบด้วยมาตรวัด Number Of Statement In Method (NOS) สามารถแสดงดังตารางที่ 4.10

Ref. code: 25595509035084XRS

```
string Customer::statement() {
            double totalAmount = 0;
            int frequentRenterPoints      = 0;
    ostringstream result;
    result << "Rental Record for " << getName () << "\n";
    for(Iter iter = rentals.begin(); iter != rentals.end(); ++iter) {
                  double thisAmount = 0;
                  Rental* each = *iter;
                  // determines the amount for each line
                  switch (each->getMovie()->getPriceCode()) {
        case Movie::Regular:
                  thisAmount += 2;
                  if (each->getDaysRented () > 2)
                  thisAmount += (each->getDaysRented () - 2) * 1.5;
                              break;
        case Movie::NewRelease:
                        thisAmount += each->getDaysRented () * 3;
                              break;
        case Movie::Childrens:
                              thisAmount += 1.5;
                  if (each->getDaysRented () > 3)
                  thisAmount += (each->getDaysRented () - 3) * 1.5;
                              break;
                  }
                  frequentRenterPoints++;
      if (each->getMovie()->getPriceCode() == Movie::NewRelease && each-
>getDaysRented () > 1)
                        frequentRenterPoints++;
            result << "\t" << each->getMovie()->getTitle() << "\t" <<
thisAmount << "\n";
                  totalAmount += thisAmount;
            }
            result << "You owed " << totalAmount << "\n" << "You earned
" << frequentRenterPoints << " frequent renter points\n";
            return result.str();
            }
```

ภาพที่ 4.24 เมธอด statement () ในภาษา C++

```java
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        String result = "Rental Record for " + getName() + "\n";

        for (Rental each: _rentals) {
            double thisAmount = 0;
            //determine amounts for each line
            switch (each.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (each.getDaysRented() > 2)
                        thisAmount += (each.getDaysRented() - 2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.getDaysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if (each.getDaysRented() > 3)
                        thisAmount += (each.getDaysRented() - 3) * 1.5;
                    break;
            }
            // add frequent renter points
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
&& each.getDaysRented() > 1)
                frequentRenterPoints++;
            // show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;
        }
        // add footer lines
        result += "Amount owed is " + String.valueOf(totalAmount) +
"\n";
        result += "You earned " + String.valueOf(frequentRenterPoints)
+ " frequent renter points";

        return result;
                }
```

ภาพที่ 4.25 เมธอด statement() ในภาษา Java

ตารางที่ 4.10

ผลการทดลองมาตรวัด Number Of Statement In Method (NOS)

เมธอด	ภาษา	จำนวน Statement (ผู้เชี่ยวชาญ)	จำนวน Statement (เครื่องมือของงานวิจัย)
statement	C++	25	25
statement	Java	24	24

4.4.2 การทดสอบมาตรวัด Number of parameter in method (PAR)

การทดสอบมาตรวัด Number of parameter in method (PAR) นั้นผู้วิจัยได้ทดสอบกับตัวอย่างซอร์สโค้ดของเมธอด Movie() ซึ่งเป็นคอนสตรัคเตอร์ของคลาส Movie โดยเมธอด Movie() ที่พัฒนาในภาษา Java และ C++ แสดงดังภาพที่ 4.26 และ 4.27 ตามลำดับ

```
Movie::Movie(const string &title, PriceCode
priceCode) :
  title(title), priceCode(priceCode)
{
}
```

ภาพที่ 4.26 เมธอด Movie ในภาษา C++

```
  public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
}
```

ภาพที่ 4.27 เมธอด Movie ในภาษา Java

Ref. code: 25595509035084XRS

จากภาพที่ 4.27 คือเมธอด Movie() ในภาษา C++ เมื่อทำการทดสอบโดยการนับด้วยผู้เชี่ยวชาญพบว่าจำนวนของพารามิเตอร์มีจำนวนทั้ง 2 จำนวนด้วยกันซึ่งเท่ากันกับผลการทดสอบกับเครื่องมือของงานวิจัยพบว่าได้จำนวนทั้งสิ้น 2 จำนวน จากนั้นได้ทดสอบกับเมธอด Movie() ในภาษา Java พบว่าเมื่อทำการนับจำนวนพารามิเตอร์ด้วยผู้เชี่ยวชาญจะได้ 2 จำนวนด้วยกันซึ่งสอดคล้องกับการทดสอบด้วยเครื่องมือของงานวิจัยนี้ พบว่าเครื่องมือสามารถนับพารามิเตอร์ทั้งสิ้น 2 จำนวน โดยผลลัพธ์ของการทดสอบมาตรวัด Number of parameter in method (PAR) สามารถแสดงดังตารางที่ 4.11

ตารางที่ 4.11

ผลการทดลองมาตรวัด Number of parameter in method (PAR)

เมธอด	**ภาษา**	**จำนวน Parameter (ผู้เชี่ยวชาญ)**	**จำนวน Parameter (เครื่องมือของงานวิจัย)**
Movie	C++	2	2
Movie	Java	2	2

4.4.3 การทดสอบมาตรวัด Nested Block Depth (NBD)

การทดสอบมาตรวัด Nested Block Depth (NBD) ผู้วิจัยได้ทดสอบกับตัวอย่างซอสร์สโค้ดเมธอด maxSongs() โดยพัฒนาทั้งภาษา Java และ C++ ซึ่งตัวอย่างซอร์สโค้ดภาษา C++ แสดงดังภาพที่ 4.28 เมื่อทำการทดสอบด้วยการคำนวณโดยผู้เชี่ยวชาญจะพบว่าจำนวนลำดับความลึกชั้นปีกกานั้นได้เท่ากับ 3 ซึ่งสอดคล้องกับผลการทดสอบกับเครื่องมือของงานวิจัยนี้พบว่าเครื่องมือคำนวณได้ค่าออกมาเท่ากับ 3 เช่นกันกับผลการทดสอบภาษา Java ดังภาพที่ 4.29 นั้นผู้เชี่ยวชาญพบว่าจำนวนความลึกชั้นปีกกานั้นได้เท่ากับ 3 และเครื่องมือของงานวิจัยนี้คำนวนความลึกได้ที่ 3 ซึ่งสรุปได้ดังตารางที่ 4.12

Ref. code: 25595509035084XRS

```
int Customer::maxSongs(vector <int> duration, vector <int> tone, int T){
    int n = tone.size();
    int best = 0;

    for (int mask=1; mask<(1<<n); mask++) {
        int maxTone = -1, minTone = 200000, durationSum = 0, c = 0;

        for (int i=0; i < n; i++) {
            if (mask & (1<<i)) {
                maxTone = max(maxTone, tone[i]);
                minTone = min(minTone, tone[i]);
                durationSum += duration[i];
                c++;
            }
        }

        if ( durationSum + maxTone - minTone <= T) {
            best = std::max(best, c);
        }
    }
    return best;
}
```

ภาพที่ 4.28 เมธอด maxSongs() ในภาษา C++

```
public int maxSongs(int[] duration , int[] tone, int[] T){
    int n = tone.length;
    int best = 0;

    for (int mask=1; mask<(1<<n); mask++) {
        int maxTone = -1, minTone = 200000, durationSum = 0, c = 0;

        for (int i=0; i < n; i++) {
            if (mask < i) {
                maxTone = Math.max(maxTone, tone[i]);
                minTone = Math.min(minTone, tone[i]);
                durationSum += duration[i];
                c++;
            }
        }

        if ( durationSum + maxTone - minTone <= T.length) {
            best = Math.max(best, c);
        }
    }
    return best;
}
```

ภาพที่ 4.29 เมธอด maxSongs() ในภาษา Java

ตารางที่ 4.12

ผลการทดลองมาตรวัด Nested Block Depth (NBD)

เมธอด	**ภาษา**	**ค่า NBD (ผู้เชี่ยวชาญ)**	**ค่า NBD (เครื่องมือของงานวิจัย)**
maxSongs()	C++	3	3
maxSongs()	Java	3	3

4.4.4 การทดสอบมาตรวัด Cyclomatic complexity (VG)

การทดสอบมาตรวัด Cyclomatic complexity (VG) นั้นสามารถคำนวณได้โดยนับจาก จำนวน Statement ทางเลือกเงื่อนไขบวกด้วยหนึ่ง ซึ่งในส่วนของ switch นั้น จะนับตามจำนวนของ case ที่เกิด โดยผู้วิจัยได้ทดสอบกับเมธอด maxSongs() จากภาพที่ 4.28 ในส่วนของภาษา C++ พบว่าผู้เชี่ยวชาญนั้นสามารถคำนวณได้ค่า Cyclomatic complexity เท่ากับ 5 ซึ่งประกอบไปด้วยลูป for จำนวน 2 statement ด้วยกัน และ 2 if statement โดยเมื่อนำไปแทนค่าในสมการที่ 4.2 ทำให้ได้ค่า Cyclomatic complexity เท่ากับ 5 สอดคล้องกับผลการทดสอบกับเครื่องมือของงานวิจัยพบว่าได้ค่า Cyclomatic complexity เท่ากับ 5 เช่นเดียวกันกับผลการทดลองภาษา JAVA จากภาพภาพที่ 4.29 ซึ่งผู้เชี่ยวชาญได้ทำการคำนวณ Cyclomatic complexity ซึ่งได้ค่าเท่ากับ 5 เช่นกัน โดยผลการทดลองมาตรวัด Cyclomatic complexity นั้นสรุปได้ดังตารางที่ 4.13

ตารางที่ 4.13

ผลการทดลองมาตรวัด Cyclomatic complexity (VG)

เมธอด	**ภาษา**	**ค่า VG (ผู้เชี่ยวชาญ)**	**ค่า VG (เครื่องมือของงานวิจัย)**
Statement	C++	5	5
Statement	Java	5	5

4.4.5 การทดสอบมาตรวัด Lack Of Cohesion In Method (LCOM)

การทดสอบมาตรวัด Lack Of Cohesion In Method (LCOM) ผู้วิจัยทำการทดสอบกับคลาสทั้งหมดจำนวน 3 คลาสด้วยกัน คือ คลาส Customer คลาส Movie และ คลาส Rental ตามลำดับ

ในส่วนของคลาส Customer ที่พัฒนาด้วยภาษา Java และ C++ แสดง ดังภาพที่ 4.30 และภาพที่ 4.31 โดยใช้สูตรการคำนวณจากสมการที่ 4.1 เมื่อทำการคำนวณโดยผู้เชี่ยวชาญพบว่าคลาส Customer ในภาษา Java มีจำนวนของเมธอดทั้งหมด 4 เมธอด คือ เมธอด Customer() เมธอด addRental() เมธอด getName() และเมธอด statement() ในส่วนของ แอททริบิวต์มีทั้งสิ้น 2 จำนวนด้วยกัน ซึ่งประกอบไปด้วย name และ _rental และพบว่าเมธอดที่มีการเรียกใช้งานแอตทริบิวต์ดังกล่าว มีดังนี้ แอตทริบิวต์ name มีเมธอดที่เรียกใช้งานคือ เมธอด Customer() และเมธอด getName() ในส่วนของแอตทริบิวต์ _rental พบว่ามีจำนวนเมธอดที่เรียกใช้งานอยู่ 2 เมธอดคือ เมธอด adRental() และเมธอด statement() เมื่อนำค่าดังกล่าวแทนที่ตัวแปรในสูตรจะพบว่าได้ $m = 4$, $v = 2$ และ $m(v_i) = 4$ ซึ่งเมื่อทำการคำนวณจะทำให้ได้ค่า LCOM เท่ากับ 0.667 จากนั้นได้นำคลาส Customer ดังกล่าวมา ทดสอบกับเครื่องมือของงานวิจัยพบว่าเครื่องมือนั้นสามารถคำนวนค่า LCOM ได้ที่ 0.666 ซึ่งให้ผลต่างจากส่วนของคลาส Customer ในภาษา C++ นั้น ผู้เชี่ยวชาญพบว่ามีจำนวนเมธอด 4 เมธอดด้วยกันและมีจำนวนของแอททริบิวต์ 2 จำนวนคือ name และ rentals และจำนวนของเมธอดที่เรียกใช้แอทรริบิวต์มี 3 เมธอดโดยแบ่งตามแอตทริบิวต์ดังนี้ แอตทริบิวต์ name มีเมธอดที่เรียกใช้งานคือ เมธอด getNmae() ในส่วนของแอตทริบิวต์ rentals นั้นมีเมธอดที่เรียกใช้งานคือ เมธอด addRental() และเมธอด statement() เมื่อนำค่าดังกล่าวแทนที่ค่าตัวแปรในสูตรจะทำให้ได้ค่า $m = 4$, $v = 2$ และ $m(v_i) = 3$ ซึ่งเมื่อทำการคำนวณค่าของ LCOM จะได้เท่ากับ 0.83 จากนั้นทดสอบกับเครื่องมือของงานวิจัยพบว่า เครื่องมือนั้นสามารถคำนวณค่า LCOM ได้ที่ 0.833 เท่ากันกับผู้เชี่ยวชาญ

Ref. code: 25595509035084XRS

```java
public class Customer {
    private String _name;
    private Vector _rentals =new Vector();
    public Customer(String name) {
        _name = name;
    }
    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }
    public String getName() {
        return _name;
    }
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();
            switch (each.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (each.getDaysRented() > 2)
                       thisAmount+=(each.getDaysRented()-2)*1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.getDaysRented() * 3;
                    break;
                case Movie.CHILDREN:
                    thisAmount += 1.5;
                    if (each.getDaysRented() > 3)
                       thisAmount+=(each.getDaysRented()-3)*1.5;
                    break;
            }
            frequentRenterPoints ++;
            if ((each.getMovie().getPriceCode()== Movie.NEW_RELEASE)&&
                each.getDaysRented() > 1) frequentRenterPoints++;
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;
        }
        result += "Amount owed is "+String.valueOf(totalAmount) + "\n";
        result += "You earned "+String.valueOf(frequentRenterPoints)
                   + " frequent renter points\n";
        return result;
    }
```

ภาพที่ 4.30 คลาส Customer ในภาษา Java

```
class Customer{
public:
  Customer(const string& name);
  void addRental(Rental* rental);
  const string& getName() const { return name; }
  string statement();
private:
  typedef vector<Rental*> RentalCollection;
  typedef RentalCollection::iterator Iter;

  string name;
  RentalCollection rentals;
};

Customer::Customer(const string& name) : name(name){}
void Customer::addRental(Rental* rental) {
  rentals.push_back(rental);
}
string Customer::statement() {
        double totalAmount = 0;
        int frequentRenterPoints    = 0;
    ostringstream result;
    result << "Rental Record for " << getName () << "\n";
    for(Iter iter = rentals.begin(); iter != rentals.end(); ++iter) {
            double thisAmount = 0;
            Rental* each = *iter;

            // determines the amount for each line
            switch (each->getMovie()->getPriceCode()) {
        case Movie::Regular:
                    thisAmount += 2;
                    if (each->getDaysRented () > 2)
                        thisAmount += (each->getDaysRented () - 2) * 1.5;
                    break;

                        // end class
```

ภาพที่ 4.31 คลาส Customer ในภาษา C++

ในส่วนของการทดสอบด้วยคลาส Movie นั้น ทางด้านผู้เชี่ยวชาญได้ทำการทดสอบกับซอร์สโค้ดโปรแกรมภาษา Java ซึ่งแสดงดังภาพที่ 4.32 พบว่าคลาส Movie นั้น มีเมธอดทั้งสิ้น 4 เมธอดด้วยกัน คือ เมธอด Movie() เมธอด getPriceCode() เมธอด setPriceCode() และเมธอด getTitle() และมีจำนวนแอตทริบิวต์ของคลาสที่ไม่ใช่ static ซึ่งประกอบไป _title กับ _priceCode และพบว่ามีเมธอดที่เรียกใช้แอตทริบิวต์ดังนี้ _title มีเมธอดที่เรียกใช้คือ เมธอด Movie() กับ เมธอด getTitle() และ ส่วนแอตทริบิวต์ _ priceCode มีเมธอดที่เรียกใช้คือ เมธอด Movie() เมธอด getPriceCode() และเมธอด setPriceCode() ซึ่งเมื่อนำค่าดังกล่าวไปแทนในตัวแปรเพื่อทำการคำนวณจะได้ $m = 4$, $r = 2$ และ $m(v_i) =$

Ref. code: 25595509035084XRS

5 ทำให้ได้ผลการคำนวณมาตรวัด LCOM ของ คลาส Movie ในภาษา Java เท่ากับ 0.5 ซึ่งค่าที่ได้นั้นสอดคล้องกับการทดสอบของเครื่องมือโดยได้ที่ 0.5 เช่นกัน

สำหรับการทดสอบคลาส Movie ในภาษา C++ ซึ่งแสดงดังภาพที่ 4.33 พบว่า มีเมธอดทั้งสิ้น 4 เมธอดด้วยกัน คือ เมธอด Movie() เมธอด getPriceCode() เมธอด setPriceCode() และ เมธอด getTitle() โดยมีแอตทริบิวต์ด้วยกัน 2 จำนวนคือ title กับ priceCode และมีจำนวนเมธอดที่เรียกใช้แอตทริบิวต์ดังนี้ title มีเมธอดที่เรียกใช้งานคือ เมธอด getTitle() ส่วนแอตทริบิวต์ priceCode มีเมธอดที่เรียกใช้งาน คือ เมธอด getPrice() และเมธอด setPrice() เมื่อนำค่าที่ได้ดังกล่าวแทนที่ในตัวแปรจะได้ค่า m = 4 ค่า v = 2 และ ค่า $m(v_i)$ = 3 ดังนั้นค่า LCOM เมื่อทดสอบโดยผู้เชี่ยวชาญมีค่าเท่ากับ 0.83 ซึ่งเมื่อนำซอร์สโค้ดดังกล่าวมาทดสอบกับงานวิจัยพบว่าเครื่องมือนั้นสามารถคำนวณค่ามาตรวัด LCOM ได้เท่ากับ 0.833 ซึ่งเท่ากันกับในส่วนของการคำนวณจากผู้เชี่ยวชาญ

```
public class Movie {

    public static final int CHILDRENS = 2;
    public static final int NEW_RELEASE = 1;
    public static final int REGULAR = 0;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        _priceCode = arg;
    }
    public String getTitle() {
        return _title;
    }

  }
```

ภาพที่ 4.32 คลาส Movie ในภาษา Java

Ref. code: 25595509035084XRS

```
#include <string>
using namespace std;

class Movie{
public:
  enum PriceCode { Childrens, Regular, NewRelease };

  Movie(const string & title, PriceCode priceCode);

  int getPriceCode() const { return priceCode; }
  void setPriceCode(PriceCode code) { priceCode = code; }
  const string& getTitle() const { return title; }

private:
  string title;
  PriceCode priceCode;
};
```

ภาพที่ 4.33 คลาส Movie ในภาษา C++

ในส่วนของคลาส Rental เมื่อทดสอบด้วยผู้เชี่ยวชาญภาษา Java ด้วยซอร์สโค้ดตัวอย่างดังภาพที่ 4.34 พบว่า จำนวนเมธอดที่อยู่ในคลาสนั้นมีทั้งหมด 3 เมธอดด้วยกันคือ เมธอด Rental() เมธอด getDaysRented() และเมธอด getMovie() และพบว่า มีแอตทริบิวต์ของคลาสอยู่ด้วยกัน 2 แอตทริบิวต์ คือ _movie และ _daysRented โดย แอตทริบิวต์ _movie มีเมธอดเรียกใช้งานคือ เมธอด Rental() กับ เมธอด getPrice() และแอตทริบิวต์ _dayRented มีเมธอดที่เรียกใช้งานคือ เมธอด Rental และเมธอด getDaysRented() เมื่อนำค่าที่หาได้ดังกล่าวแทนที่ในตัวแปรจะได้ค่า $m = 3$ ค่า $v = 2$ และค่า $m(v_i) = 4$ ดังนั้นค่ามาตรวัด LCOM ของคลาส Rental เมื่อทดสอบจากผู้เชี่ยวชาญจะได้เท่ากับ 0.5 ซึ่งเมื่อนำซอร์สโค้ดดังกล่าวมาทดสอบกับเครื่องมือ พบว่าค่ามาตรวัด LCOM ที่เครื่องมือคำนวณได้เท่ากับ 0.5 ซึ่งเท่ากับการทดสอบจากผู้เชี่ยวชาญ

การทดสอบคลาส Rental กับภาษา C++ ดังภาพที่ 4.35 จากการทดสอบด้วยผู้เชี่ยวชาญพบว่า คลาส Rental มีจำนวนเมธอดทั้งสิ้น 3 เมธอดด้วยกัน คือ เมธอด Rental() เมธอด getDaysRented() และเมธอด getMovie() โดยมีจำนวนแอตทริบิวต์ 2 จำนวนด้วยกันคือ movie กับ daysRented ซึ่งแอตทริบิวต์ movie นั้นมีเมธอดที่เรียกใช้งานคือ เมธอด getMovie() ส่วน แอตทริบิวต์ daysRented นั้นมีเมธอดที่เรียกใช้งานคือ getDaysRented ซึ่งเมื่อนำค่าดังกล่าวแทนในตัวแปรพบว่าได้ค่า $m = 3$ ค่า $v = 2$ และ $m(v_i) = 2$ ดังนั้นคลาส Rental ในภาษา C++ เมื่อทดสอบด้วยผู้เชี่ยวชาญพบว่า ได้ค่ามาตรวัด LCOM เท่ากับ 1.0 สอดคล้องกับผลการทดสอบจากเครื่องมือซึ่งได้ค่า LCOM เท่ากับ 1.0 เช่นกัน

ดังนั้นผลการทดสอบของมาตรวัด LCOM สามารถสรุปได้ดังตารางที่ 4.14

Ref. code: 25595509035084XRS

```java
public class Rental {

    private Movie _movie;
    private int _daysRented;

    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }

    public int getDaysRented() {
        return _daysRented;
    }

    public Movie getMovie() {
        return _movie;
    }
}
```

ภาพที่ 4.34 คลาส Rental ในภาษา Java

```cpp
class Rental
{
public:
  Rental(Movie *movie, int daysRented);

  int getDaysRented() const { return daysRented; }
  Movie* getMovie() const { return movie; }

private:
  Movie* movie;
  int daysRented;
};
```

ภาพที่ 4.35 คลาส Rental ในภาษา C++

Ref. code: 25595509035084XRS

ตารางที่ 4.14

ผลการทดลองมาตรวัด Lack Of Cohesion In Method (LCOM)

Class	ภาษา	ค่า LCOM (ผู้เชี่ยวชาญ)	ค่า LCOM (เครื่องมือของงานวิจัย)
Customer	C++	0.83	0.833
Customer	Java	0.667	0.666
Movie	C++	0.83	0.833
Movie	Java	0.5	0.5
Rental	C++	1.0	1.0
Rental	Java	0.5	0.5

4.4.5 การทดสอบมาตรวัด CU มาตรวัด PU และมาตรวัด DU

กลุ่มมาตรวัด Computation Use (CU) มาตรวัด Predicate Use (PU) และมาตรวัด Definition Use (DU) นั้นเป็นกลุ่มมาตรวัดที่ใช้สำหรับระบุวิธีการรีแฟคทอริงซึ่งสามารถหาได้โดยการแปลง Control flow graph และ Data flow graph จากนั้นทำการหาค่ามาตรวัดแต่ละประเภท โดย CU เป็นการนับตัวแปรชั่วคราวและใช้สำหรับการคำนวณ, PU เป็นการนับตัวแปรชั่วคราวที่ใช้สำหรับเงื่อนไขหรือทางเลือก และ DU นับจากตัวแปรชั่วคราวที่ใช้สำหรับกำหนดค่าตัวแปร โดยมีซอร์สโค้ดโปรแกรมที่ใช้ทดสอบคือเมธอด getPrice() ซึ่งสร้างไว้ทั้ง 2 ภาษาโปรแกรมด้วยกันคือ Java และ C++ ซึ่งแสดงดังภาพที่ 4.36 และ 4.37 ตามลำดับ การทดสอบโดยผู้เชี่ยวชาญนั้นพบว่าค่าของกลุ่มมาตรวัดที่ได้จากการซอร์สโค้ดทั้ง 2 ภาษานั้นให้ค่าเท่ากันคือ ตัวแปร basePrice ในเส้นทาง 2 3 4 และ 6 ซึ่งเมื่อจำแนกตามมาตรวัดจะพบว่าได้ค่า DU = 1 ที่บรรทัดที่ 2 ค่า PU = 1 ที่บรรทัดที่ 3 และ CU = 2 ที่บรรทัดที่ 4 และ 6 หลังจากนั้นผู้วิจัยได้นำซอร์สโค้ดโปรแกรมทั้ง 2 ภาษามาทดสอบกับเครื่องมือของงานวิจัยพบว่าทั้ง Java และ C++ ต่างก็ให้ค่าเหมือนกันคือ DU =1 PU =1 และ CU = 2 โดยผลลัพธ์การทดลองดังกล่าวแสดงดังตารางที่ 4.15

Ref. code: 25595509035084XRS

```
1    public double getPrice(){
2        double basePrice = _quantity*_itemPrice;
3        if(basePrice > 1000)
4                return basePrice*0.95;
5        else
6                return basePrice*0.98;
7    }
8
```

ภาพที่ 4.36 เมธอด getPrice() ในภาษา Java

```
1    double double::getPrice(){
2        const double basePrice = _quantity * _itemPrice;
3        if(basePrice > 1000)
4                return basePrice*0.95;
5            else
6                return basePrice*0.98;
7    }
```

ภาพที่ 4.37 เมธอด getPrice() ในภาษา C++

ตารางที่ 4.15

ผลการทดลองมาตรวัด CU , PU และ DU

เมธอด	ภาษา	ผู้เชี่ยวชาญ			เครื่องมือ		
		DU	CU	PU	DU	CU	PU
getPrice()	C++	1	2	1	1	2	1
getPrice()	Java	1	2	1	1	2	1

บทที่ 5
สรุปผลการวิจัยและข้อเสนอแนะ

5.1 สรุปงานวิจัย

งานวิจัยนี้นำเสนอวิธีการและเครื่องมือที่ใช้ในการวิเคราะห์คำนวณหาค่ากลุ่มมาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษา ในส่วนคุณลักษณะย่อย ด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ดจากมาตรฐาน ISO 9126 และกลุ่มมาตรวัดสำหรับเลือกรีแฟคทอริงที่เหมาะสม (Metrics for refactoring selection) ซึ่งสามารถใช้ได้กับภาษาโปรแกรมเชิงวัตถุ ในที่นี้คือ ภาษา Java และภาษา C++ โดยกลุ่มมาตรวัดคุณภาพซอฟต์แวร์ด้านการบำรุงรักษาในส่วนคุณลักษณะย่อยด้านความสามารถวิเคราะห์หาข้อผิดพลาดในซอร์สโค้ดจากมาตรฐาน ISO 9126 ประกอบด้วยมาตรวัด Number Of Statement In Method (NOS) มาตรวัด Number Of Parameter In Method (PAR) มาตรวัด Nested Block Depth (NBD) มาตรวัด McCabe Cyclomatic Complexity (VG) มาตรวัด Lack Of Cohesion In Method (LCOM) และ กลุ่มมาตรวัดของการเลือกรีแฟคทอริงที่เหมาะสมประกอบไปด้วยมาตรวัด CU มาตรวัด DU และมาตรวัด PU โดยวิธีการที่เสนอนั้นสามารถรองรับได้หลายภาษาโปรแกรมเชิงวัตถุ โดยใช้เทคนิคกระบวนการพัฒนาซอฟต์แวร์แบบเอ็มดีเอ (MDA : Model-Driven Architecture) ซึ่งกระบวนการนี้เริ่มต้นโดยนำเครื่องมือ Sissy มาแปลงซอร์สโค้ดโปรแกรมภาษาเชิงวัตถุให้เป็นแบบจำลองโดยมีเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์แบบนามธรรม (GAST : Generalized Abstract Syntax Tree Meta-model) ซึ่งใช้สำหรับอธิบายแบบจำลองดังกล่าว จากนั้นทำการสร้างเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรม (PDG : Program Dependency Graph Meta-model) เพื่อทำการแปลงแบบจำลองเพื่อให้ได้ข้อมูลที่จำเป็นสำหรับคำนวณค่ามาตรวัดที่ต้องการ หลังจากนั้นทำการสร้างเมตาโมเดลเก็บค่ามาตรวัดของโปรแกรม (Metrics Meta-model) เพื่อใช้สำหรับคำนวณหามาตรวัดที่ใช้ในงานวิจัยนี้ซึ่งได้กล่าวไว้ข้างต้น จากนั้นได้ทำการกำหนดกฎการแปลงแบบจำลองและการคำนวณค่ามาตรวัดซึ่งรวมแล้ว 15 กฎด้วยกัน จากนั้นทำกระบวนการแปลงแบบจำลองโดยใช้ภาษาเอทีแอล (ATL transformation language) เพื่อให้ได้ผลลัพธ์ตามที่ต้องการ หลังจากนั้นนำขั้นตอนทั้งหมดที่กล่าวมานั้นมาพัฒนาเป็นเครื่องมือที่เป็นส่วนเสริมของโปรแกรมอิคลิปส์ (Eclipse plugins) สุดท้ายนั้นทำการทดสอบเครื่องมือเพื่อวัดความถูกต้องของมาตรวัด และความสามารถในการทำงานได้มากกว่า 1 ภาษาโปรแกรมเชิงวัตถุโดยทำการทดสอบกับผู้เชี่ยวชาญ ซึ่งประกอบไปด้วย ผู้เชี่ยวชาญทางด้านภาษา Java และผู้เชี่ยวชาญทางด้านภาษา C++ โดยแบ่งการทดสอบ

Ref. code: 25595509035084XRS

ออกเป็น 6 การทดลองด้วยกันตามรายการมาตรวัดด้วยตัวอย่างซอร์สโค้ดโปรแกรมภาษา Java และภาษา C++ ที่มีหน้าที่การทำงานเหมือนกันซึ่งได้ผลดังนี้

5.1.1 มาตรวัด Number Of Statement In a Method (NOS)

การทดสอบมาตรวัด Number Of Statement In a Method (NOS) นั้นได้ทดสอบกับเมธอด statement() โดยแบ่งเป็นภาษา Java กับภาษา C++ จากนั้นให้ผู้เชี่ยวชาญทำการทดสอบพบว่าเครื่องมือนั้นสามารถคำนวณค่ามาตรวัดได้ผลเดียวกันกับทางด้านผู้เชี่ยวชาญ

5.1.2 มาตรวัด Number of parameters in method (PAR)

สำหรับมาตรวัด Number of parameter in method (PAR) ได้ทำการทดสอบกับเมธอด Movie() ซึ่งพบว่า ผลการทดสอบจากผู้เชี่ยวชาญทั้งสองภาษานั้นเท่ากันกับผลการคำนวณค่ามาตรวัดจากเครื่องมือ

5.1.3 มาตรวัด Nested Block Depth (NBD)

มาตรวัด Nested Block Depth (NBD) ได้ทำการทดสอบกับเมธอด maxSong() โดยในส่วนของภาษา Java นั้น พบว่าผลการทดสอบจากผู้เชี่ยวชาญให้ผลลัพธ์เดียวกันกับผลการคำนวณของเครื่องมือ สอดคล้องกับผลการทดสอบทางด้านภาษา C++ ต่างก็คำนวณได้เท่ากัน

5.1.4 มาตรวัด McCabe Cyclomatic complexity (VG)

การทดสอบมาตรวัด Cyclomatic complexity (VG) นั้นได้ใช้ทดสอบกับเมธอด maxSongs() โดยในส่วนของภาษา Java พบว่าผลการทดสอบจากผู้เชี่ยวชาญนั้นให้ค่าตรงกันกับผลการทดสอบจากเครื่องมือ สอดคล้องกับภาษา C++ ซึ่งพบว่าผลการทดสอบจากผู้เชี่ยวชาญและเครื่องมือต่างก็ให้ค่าเท่ากัน

5.1.5 มาตรวัด Lack Of Cohesion In Method (LCOM)

การทดสอบมาตรวัด Lack Of Cohesion In Method (LCOM) ได้ทำกับทดสอบด้วยคลาสจำนวน 3 คลาสด้วยกันคือคลาส Customer คลาส Movie และคลาส Rental โดยแบ่งออกเป็นภาษา Java และภาษา C++ โดยผลการทดสอบพบว่าเครื่องมือนั้นคำนวณค่ามาตรวัดได้ค่าเดียวกันกับผลการทดสอบจากผู้เชี่ยวชาญทั้ง 2 ภาษา

Ref. code: 25595509035084XRS

5.1.6 กลุ่มมาตรวัด Computation Use (CU) มาตรวัด Predicate Use (PU) และมาตรวัด Definition Use (DU)

มาตรวัด Computation Use (CU) มาตรวัด Predicate Use (PU) และมาตรวัด Definition Use (DU) ได้ทำการทดสอบด้วยเมธอด getPrice() การทดสอบโดยผู้เชี่ยวชาญนั้นพบว่าค่าของกลุ่มมาตรวัดที่ได้จากการซอร์สโค้ดทั้ง 2 ภาษานั้นให้ค่าเดียวกันกับผลการทดสอบจากเครื่องมือ

5.2 ข้อเสนอแนะ

5.2.1 ในด้านการคำนวณกลุ่มมาตรวัด CU, DU และ PU นั้นเพื่อให้เครื่องมือทำงานได้อย่างมีประสิทธิภาพมากขึ้น ระบบควรที่จะมีการเก็บเส้นทางของตัวแปรที่คำนวณเพื่อให้ทราบว่ามีเส้นทางใดบ้าง ซึ่งสามารถทำได้โดยการขยายเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรมเพิ่มเติม

5.2.2 ด้วยข้อจำกัดของเมตาโมเดลโครงสร้างต้นไม้วากยสัมพันธ์แบบนามธรรมที่ให้ข้อมูลรายละเอียดของ Expression นั้นไม่เพียงพอ อาจทำให้ผู้ที่สนใจวิธีการคำนวณมาตรวัดของงานวิจัยนี้ไม่สามารถนำไปใช้กับมาตรวัดที่ต้องคำนวณโดยใช้รายละเอียดของ Expression ได้ ผู้วิจัยขอแนะนำให้ทำการสำรวจ เมตาโมเดลที่มีลักษณะโครงสร้างต้นไม้วากยสัมพันธ์แบบนามธรรม ประเภทอื่น เช่น Knowledge Discovery Metamodel (KDM) ที่ให้รายละเอียดในส่วนของ Expression มากกว่า เป็นต้น

5.2.3 งานวิจัยนี้มุ่งเน้นการคำนวณมาตรวัดซอฟต์แวร์ในระดับเมธอด ผู้ที่สนใจสามารถพัฒนาให้สามารถคำนวณในระดับคลาสได้โดยการขยายเมตาโมเดลกราฟแสดงความสัมพันธ์ของโปรแกรมเพิ่มเติมเพื่อให้สามารถรองรับมาตรวัดในระดับคลาสตามที่ต้องการได้

5.2.4 ในด้านของ Sissy นั้นนอกจากแปลงภาษา Java และ C++ ให้ออกมาอยู่ในรูปของแบบจำลองโครงสร้างต้นไม้วากยสัมพันธ์แบบนามธรรมได้แล้ว ยังสามารถใช้กับภาษา Pearl ได้ ด้วยเวลาที่จำกัดนั้นทางผู้วิจัยไม่ได้ทดสอบกับภาษาดังกล่าว ผู้ที่สนใจนั้นสามารถทดสอบเพิ่มเติมกับภาษา Pearl ได้

Ref. code: 25595509035084XRS

รายการอ้างอิง

เอกสารประกอบการประชุมวิชาการ

Charalampidou, S., et al. (2015). Size and cohesion metrics as indicators of the long method bad smell: An empirical study. Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering. Beijing, China

Chawla, M. K. and I. Chhabra (2015). SQMMA: Software Quality Model for Maintainability Analysis. Proceedings of the 8th Annual ACM India Conference. Ghaziabad, India, ACM: 9-17.

Chidamber, S. R. and C. F. Kemerer (1992). A metrics suite for object oriented design. Cambridge, Mass., Center for Information Systems Research, Sloan School of Management, Massachusetts Institute of Technology.

Dallal, J. A. and L. C. Briand (2012). "A Precise Method-Method Interaction-Based Cohesion Metric for Object-Oriented Classes." ACM Trans. Softw. Eng. Methodol. 21(2): 1-34.

Danphitsanuphan, P. and T. Suwantada (2012). Code Smell Detecting Tool and Code Smell-Structure Bug Relationship. 2012 Spring Congress on Engineering and Technology.

Fokaefs, M., et al. (2007). JDeodorant: Identification and Removal of Feature Envy Bad Smells. Software Maintenance, 2007. ICSM 2007. IEEE International Conference on.

Fontana, F. A., et al. (2012). "Automatic detection of bad smells in code: An experimental assessment." Journal of Object Technology 11(2): 5:1-38.

Fontana, F. A., et al. (2013). Code Smell Detection: Towards a Machine Learning-Based Approach. Software Maintenance (ICSM), 2013 29th IEEE International Conference on.

Fowler, M. (1999). Refactoring : improving the design of existing code. Reading, Mass., Addison-Wesley.

Heitlager, I., et al. (2007). A Practical Model for Measuring Maintainability. Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the.

Ref. code: 25595509035084XRS

Higo, Y., et al. (2005). ARIES: refactoring support tool for code clone. Proceedings of the third workshop on Software quality. St. Louis, Missouri, ACM: 1-4.

Higo, Y., et al. (2008). "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system." J. Softw. Maint. Evol. 20(6): 435-461.

Jouault, F., et al. (2008). "ATL: A model transformation tool." Science of computer programming 72(1): 31-39.

Kaur, S. and R. Maini "Analysis of Various Software Metrics Used To Detect Bad Smells." The International Jounal Of Engineering And Science (IJES) : 14-20.

Koziolek, H., et al. (2011). An industrial case study on quality impact prediction for evolving service-oriented software. Proceedings of the 33rd International Conference on Software Engineering, ACM.

Lorenz, M. and J. Kidd (1995). "Book review: Object-Oriented Software Metrics by Mark Lorenz and Jeff Kidd." SIGSOFT Softw. Eng. Notes 20(1): 91-93.

Mäntylä, M. V. and C. Lassenius (2006). "Subjective evaluation of software evolvability using code smells: An empirical study." Empirical Software Engineering 11(3): 395-431.

McCabe, T. J. and C. W. Butler (1989). "Design complexity measurement and testing." Commun. ACM 32(12): 1415-1425.

Meananeatra, P., et al. (2013). Modeling Code Analyzability at Method Level in J2EE Applications. 2013 20th Asia-Pacific Software Engineering Conference (APSEC).

Meananeatra, P., et al. (2011). Using software metrics to select refactoring for long method bad smell. Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on.

Meananeatra, P. (2016). "An Effective Approach To Identify And Suggest Appropriate Refactoring For Removing Long Mmethod Bad Smell."

Miller, J. a. M., J. (2003). "Mda guide version 1.0.1. Tech. rep., Object Management Group (OMG)."

Rani, A. and H. Kaur (2014). "Detection of bad smells in source code according to their object oriented metrics." International Journal for Technological Research in Engineering 1(10): 1211-1214.

Ref. code: 25595509035084XRS

Singh, S. and K. Kahlon (2012). "Effectiveness of refactoring metrics model to identify smelly and error prone classes in open source software." ACM SIGSOFT Software Engineering Notes 37(2): 1-11.

SISSy (2011). "Structural Investigation in Software Systems ,http://www.sqools.org/sissy/."

Sreenu, K. and D. J. Rao "Performance-Detection of Bad Smells In Code for Refactoring Methods."

Soley, R. (2000). "Model driven architecture." OMG white paper 308(308): 5.

Srivisut, K. and P. Muenchaisri (2007). Bad-smell metrics for aspect-oriented software. 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007), IEEE.

Zhao, L. and J. Hayes (2006). Predicting classes in need of refactoring: an application of static metrics. 2nd International PROMISE Workshop, Philadelphia, Pennsylvania USA (co-located with the IEEE Conference on Software Maintenance)

นงเยาว์และคณะ (2545). "การวัดซอฟต์แวร์เชิงวัตถุ." NECTEC Technical 4.

Ref. code: 25595509035084XRS

ภาคผนวก

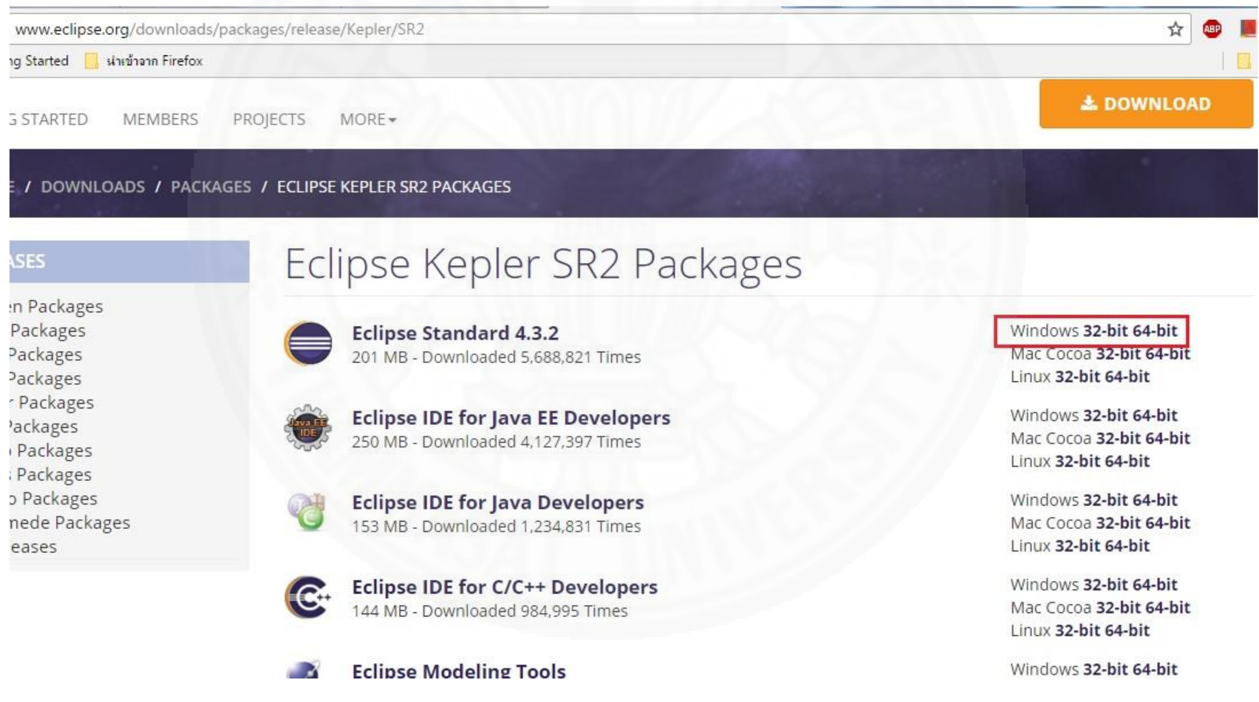
Ref. code: 25595509035084XRS

ภาคผนวก ก
การติดตั้งเครื่องมือ

เครื่องมือที่ใช้ติดตั้งสามารถแบ่งได้เป็น 3 ขั้นตอนคือ 1. การติดตั้งโปรแกรม Eclipse 2. การติดตั้งเครื่องมือ SISSy 3. การติดตั้งเครื่องมือ Scan Code 4. การใช้งานเครื่องมือ โดยมีรายละเอียดดังนี้

1. การติดตั้งโปรแกรม Eclipse (บน Window)

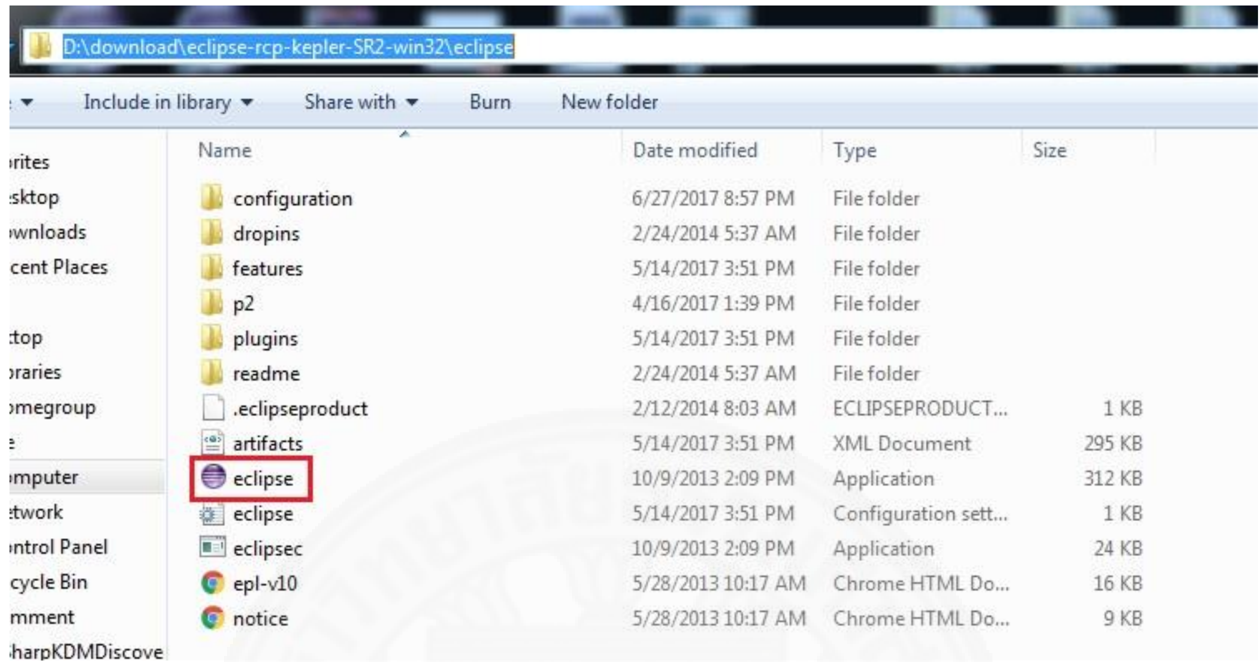
1.1 ดาวน์โหลดโปรแกรมที่ http://www.eclipse.org/downloads/packages/release/Kepler/SR2 โดยเลือกดาวน์โหลดตามประเภท OS บนเครื่อง (Windows 32 Bit และ Windows 64 Bit) ดังภาพที่ ก-1



ภาพที่ ก-1. หน้าเว็บไซต์ดาวน์โหลดโปรแกรม Eclipse

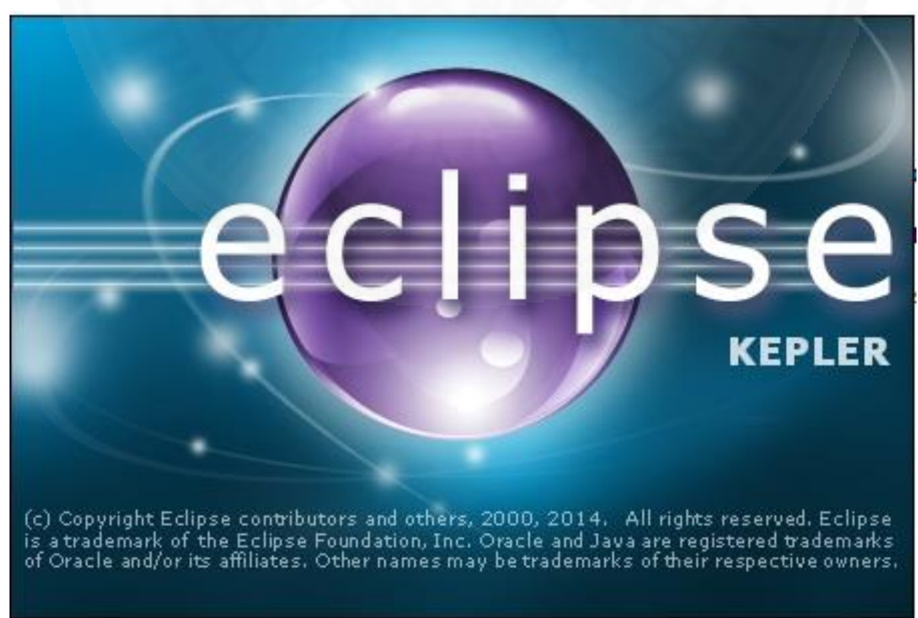
2.2 นำไฟล์ที่ดาวน์โหลดไปไว้ในที่ที่ต้องการโดยจะเห็นไอคอนของโปรแกรม Eclipse ในกรอบสีแดงดังภาพที่ ก-2

Ref. code: 25595509035084XRS



ภาพที่ ก-2. แสดงการติดตั้งโปรแกรมอีคลิปส์

3.3 หลังจากทำการติดตั้งโปรแกรม Eclipse ในที่ต้องการแล้วทำการเปิดโปรแกรม Eclipse ขึ้นมา หากทำการติดตั้งสำเร็จ โปรแกรมจะแสดงดังภาพที่ ก-3

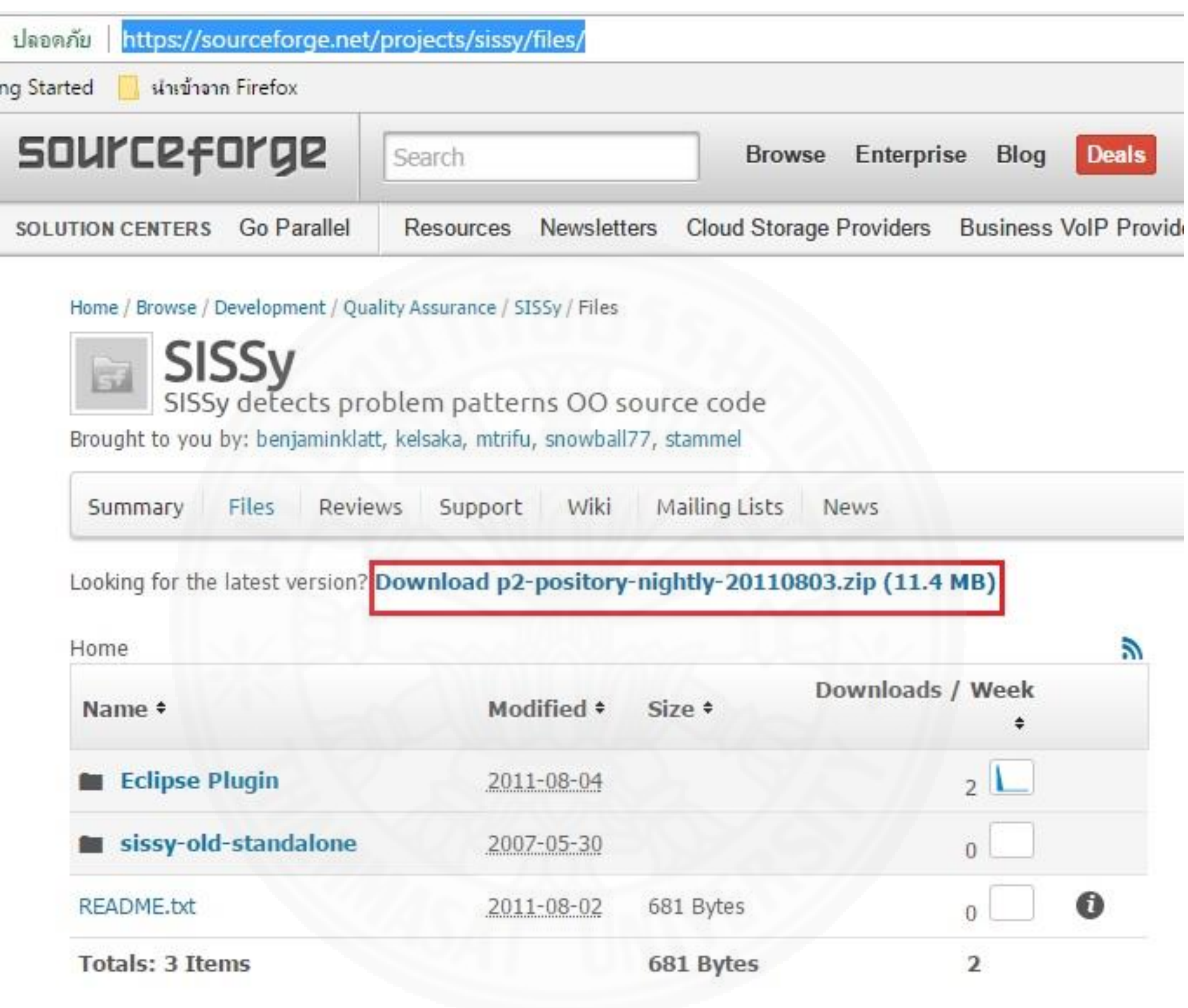


ภาพที่ ก-3. การเปิดใช้งานโปรแกรม Eclipse

Ref. code: 25595509035084XRS

2. การติดตั้งเครื่องมือ SISSy

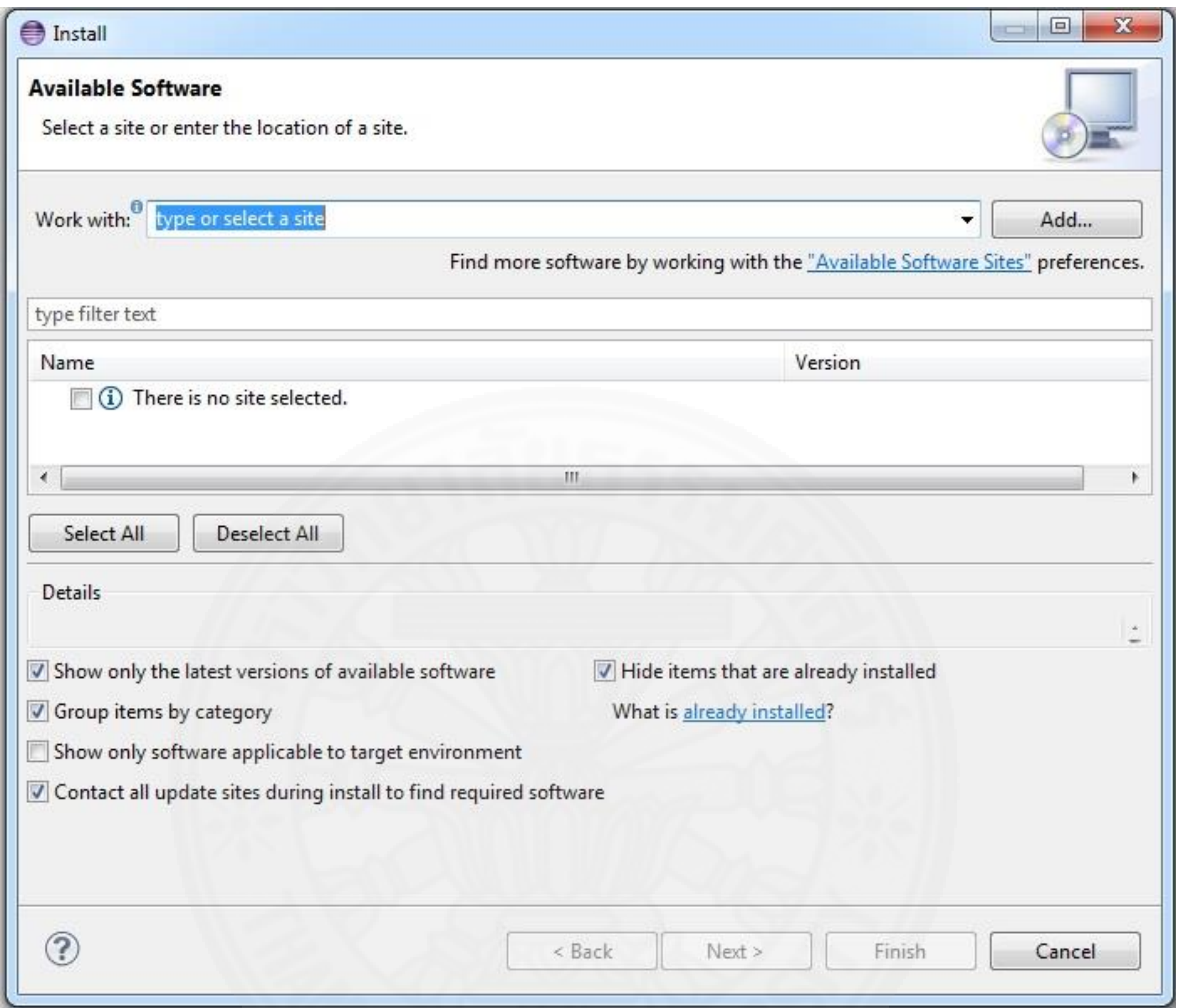
2.1 ดาวน์โหลดโปรแกรมที่ https://sourceforge.net/projects/sissy/files/ โดยเลือกดาวน์โหลดในกรอบสีแดงตามภาพที่ ก-4



ภาพที่ ก-4. ดาวน์โหลดเครื่องมือ SISSy

2.2 หลังจากดาวน์โหลดแล้วจะได้ไฟล์ชื่อ p2-pository-nightly-20110803.zip แล้วเปิดโปรแกรม Eclipse ขึ้นมา

2.3 ไปที่เมนู Help -> Install New Software จะแสดงหน้าต่างตามภาพที่ ก-5



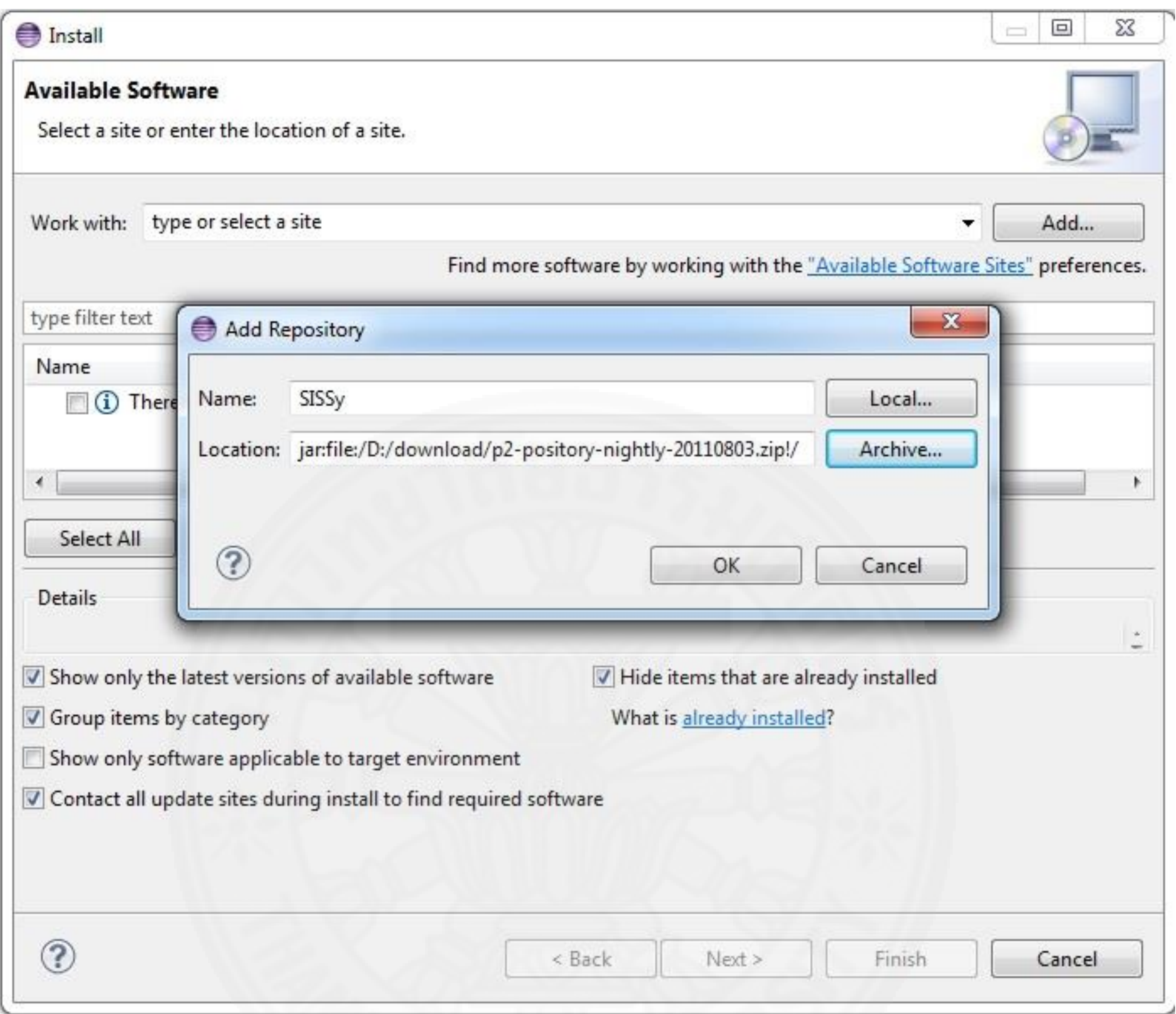
ภาพที่ ก-5. แสดงหน้าจอ Install New Software

2.4 กดปุ่ม Add จะแสดงหน้าต่างให้เพิ่มโปรแกรม

2.5 ช่อง Name พิมพ์ชื่อ “SISSy”

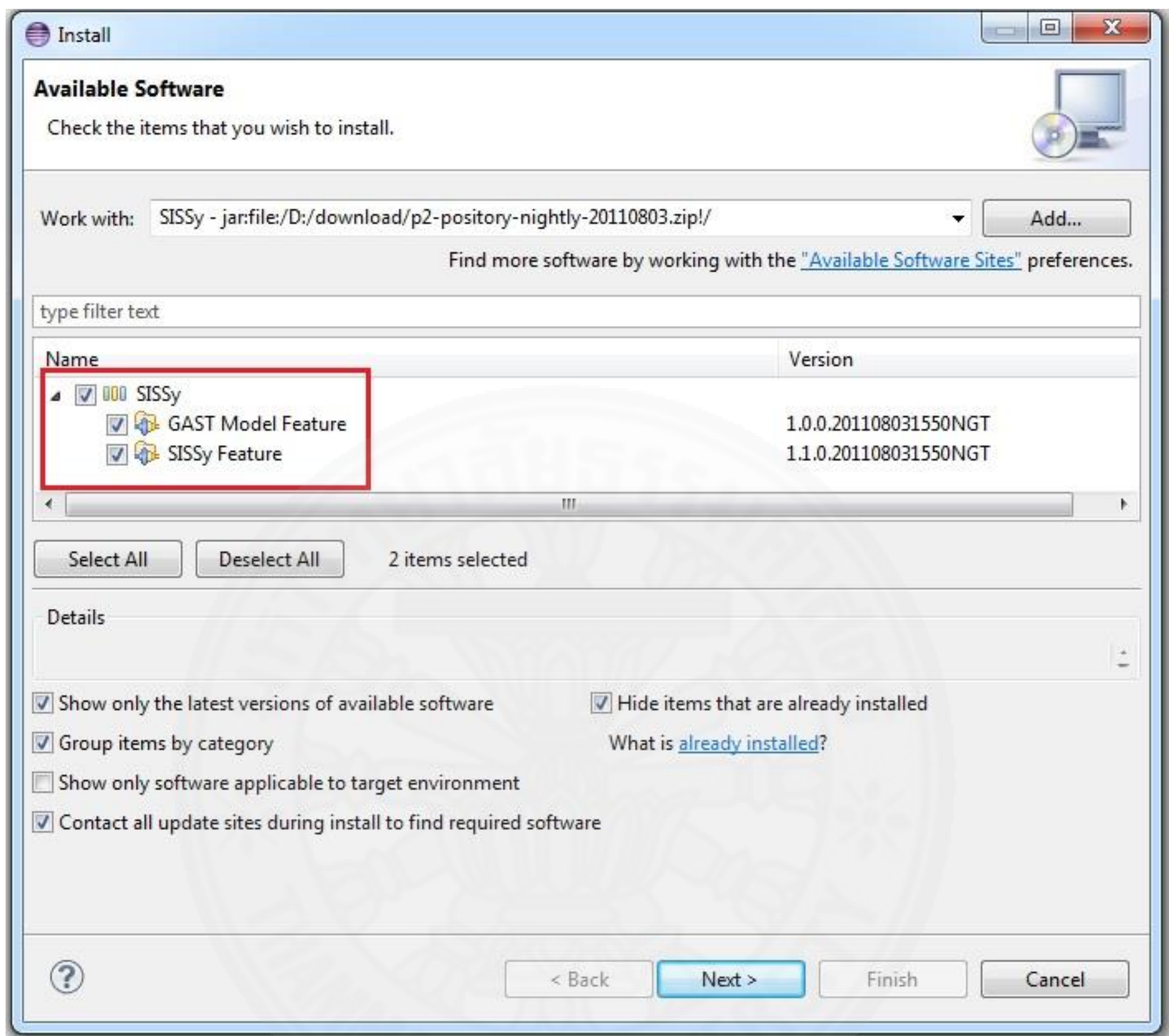
2.6 ช่อง Locationให้เลือกไฟล์ p2-pository-nightly-20110803.zip ที่ได้ดาวน์โหลดไว้ดังภาพที่ ก-6

Ref. code: 25595509035084XRS



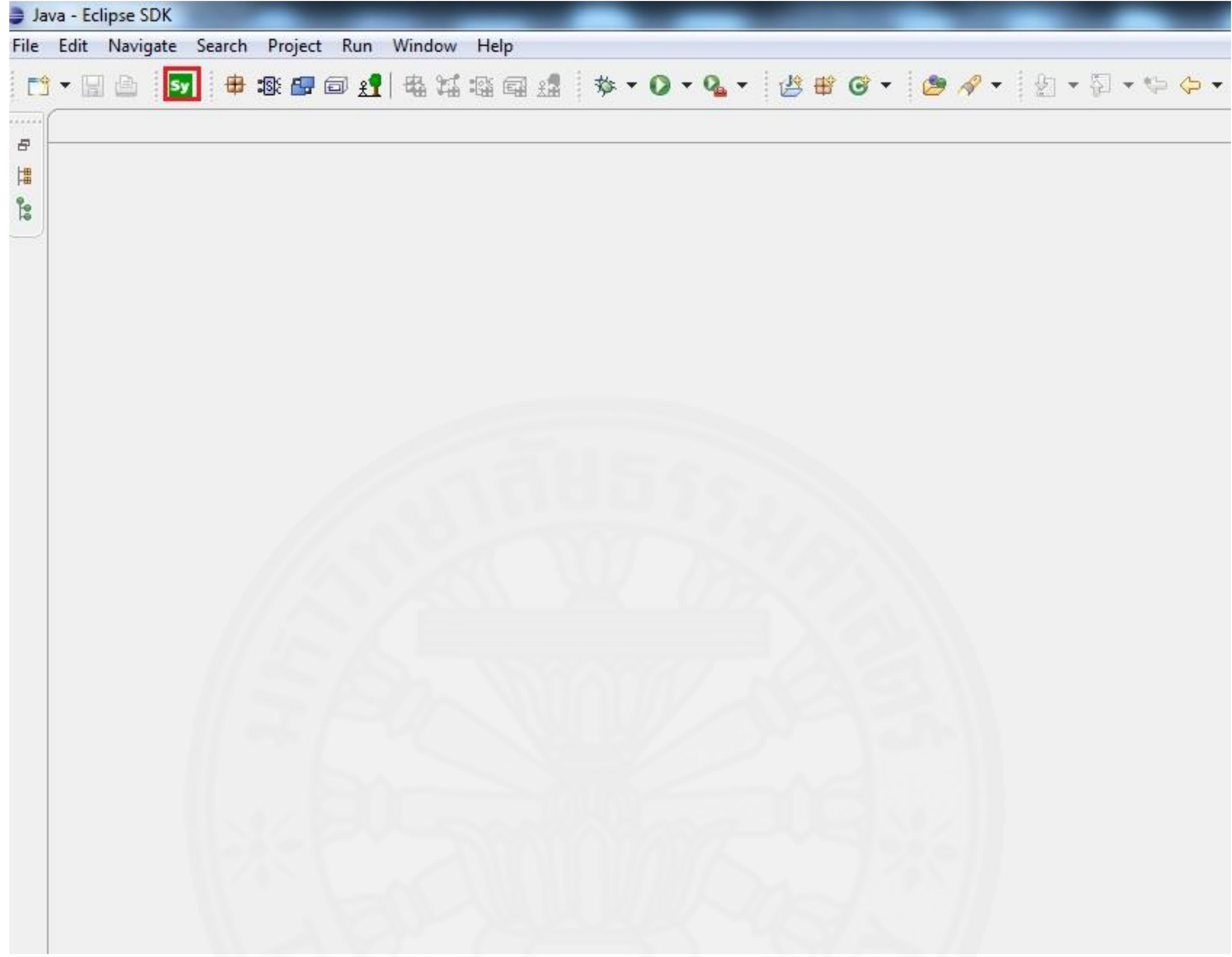
ภาพที่ ก-6. แสดงหน้าจอการเพิ่มโปรแกรม

2.7 หลังจากกดปุ่ม OK แล้วจะกลับมาสู่หน้าจอของ Install New Software ให้เลือกดังภาพ ก-6 จากนั้นกดปุ่ม Next จนกระทั่งติดตั้งโปรแกรมเสร็จ



ภาพที่ ก-7. หน้าจอ Install New Software หลังจากเพิ่มโปรแกรม

2.8 หลังจากติดตั้งโปรแกรมสำเร็จแล้วโปรแกรมจะทำการ restart เมื่อโปรแกรมเปิดขึ้นมาจะแสดงไอคอนในกรอบสีแดงดังภาพที่ ก-8



ภาพที่ ก-8. แสดงไอคอน SISSy

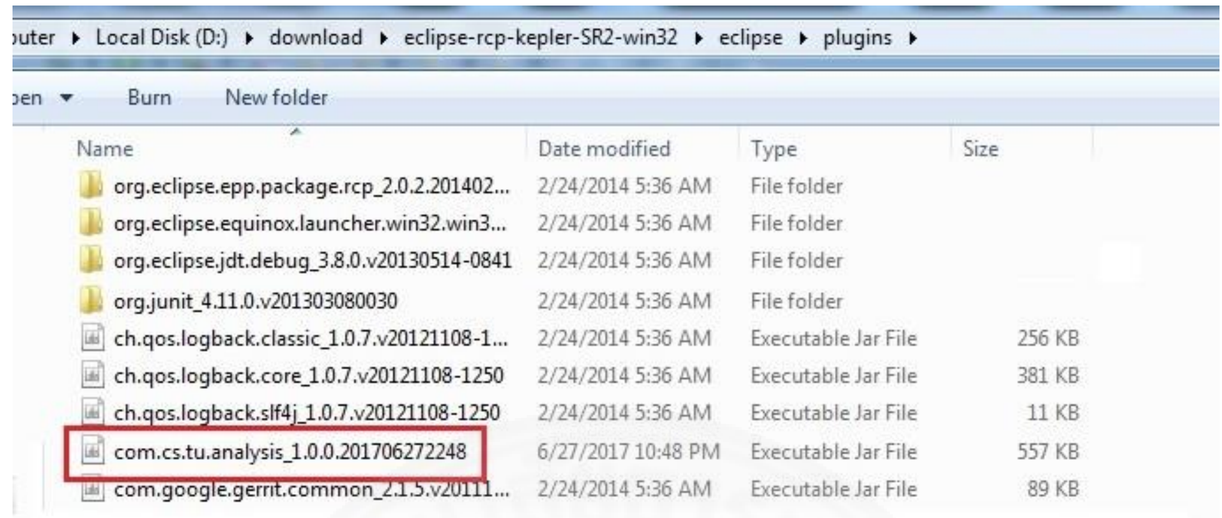
3. การติดตั้งเครื่องมือ Scan Code

3.1 เข้าไปที่โฟลเดอร์ plugins ในโปรแกรม Eclipse

3.2 นำไฟล์ com.cs.tu.analysis_1.0.0.201706272248 วางในโฟลเดอร์ plugins ดังภาพที่ ก-9

3.3 Restart โปรแกรม Eclipse

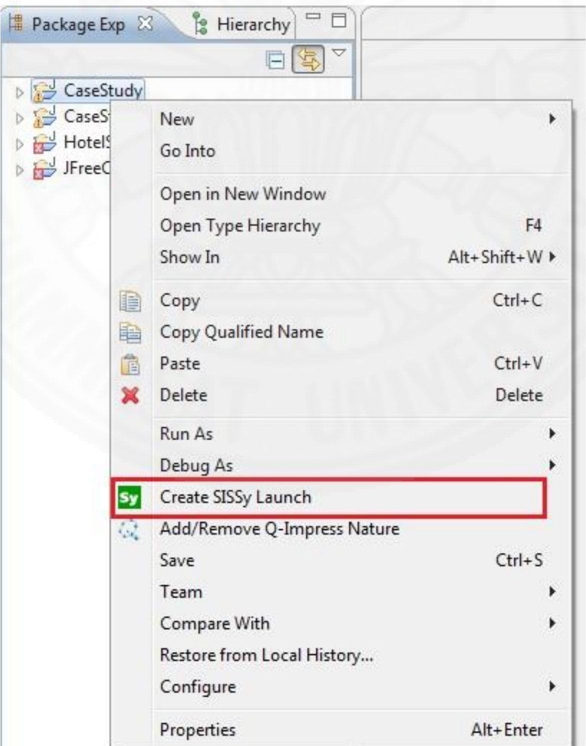
Ref. code: 25595509035084XRS



ภาพที่ ก-9. แสดงการติดตั้ง ScanCode

4. การใช้งานเครื่องมือ

4.1 คลิกขวาที่โปรเจคที่ต้องการทดสอบ เลือก Create SISSy Launch ดังภาพที่ ก-10

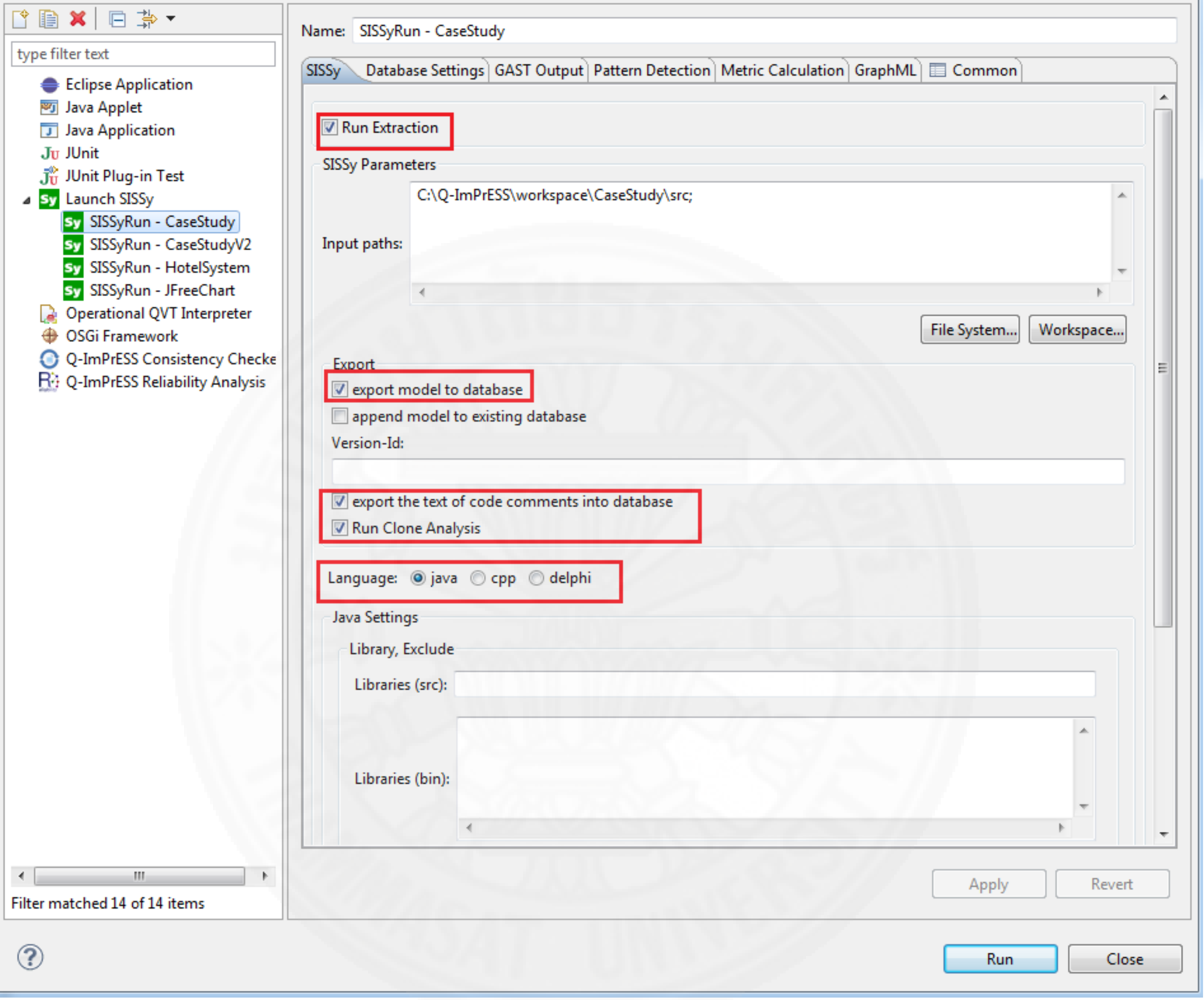


ภาพที่ ก-10. Create SISSy Launch

Ref. code: 25595509035084XRS

4.2 หลังจาก Create SISSy Launch จะปรากฏหน้าจอ Launch SISSy

4.3 เลือกแถบ SISSy คลิกเลือก ตามกรอบสีแดง ดังภาพที่ ก-11

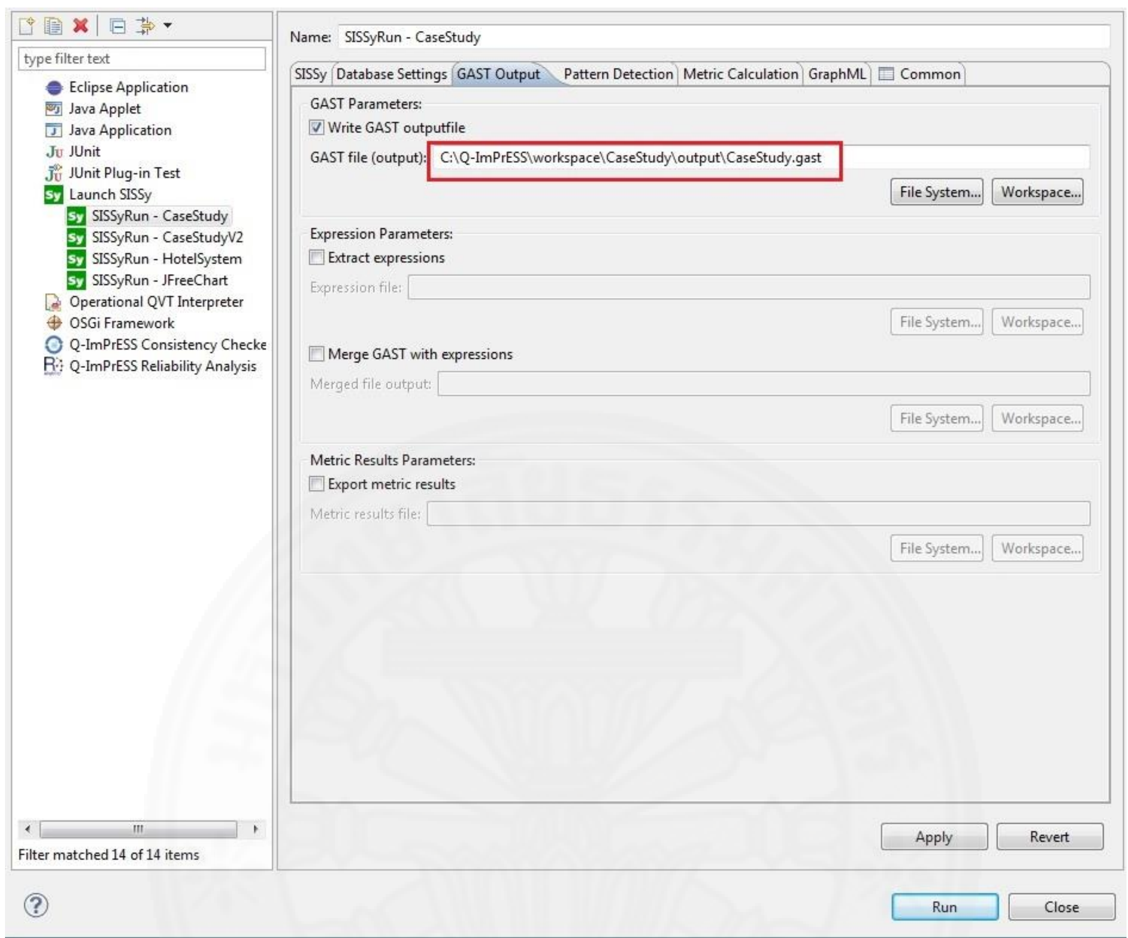


ภาพที่ ก-11. การใช้งานแถบ SISSy

4.4 เลือกแถบ GAST Output คลิกเลือก ตามกรอบสีแดง

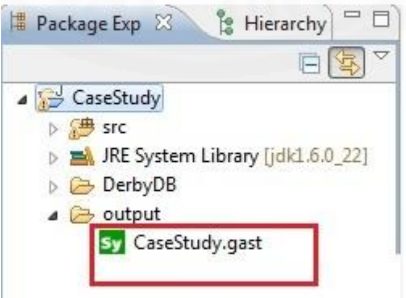
4.5 เลือก Write GAST outputfile

4.6 พิมพ์ชื่อไฟล์ที่ต้องการ เช่น CasStudy.gast ดังภาพที่ ก-12



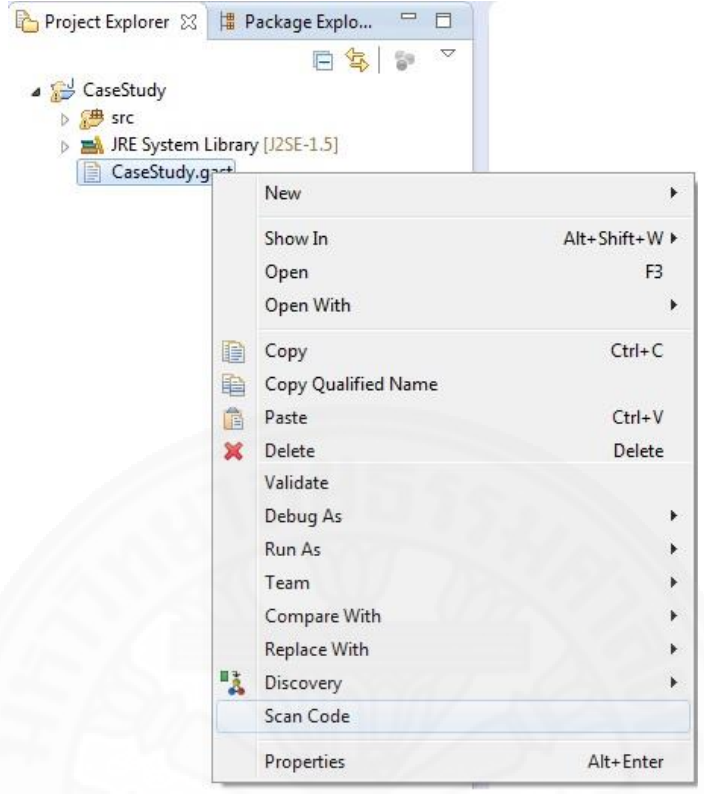
ภาพที่ ก-12. การใช้งานแถบ GAST Output

4.7 จากนั้นทำการกด Run จะได้ไฟล์ CastStudy.gast ดังภาพที่ ก-13



ภาพที่ ก-13. CastStudy.gast

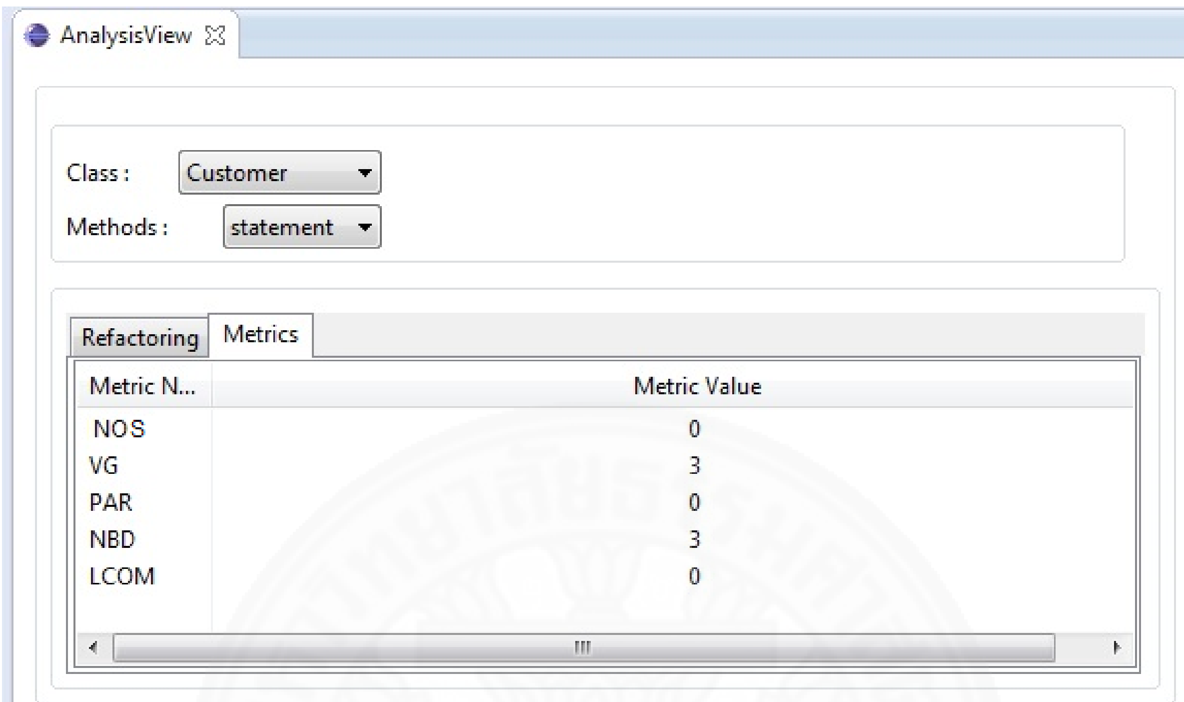
4.8 คลิกขวาที่ไฟล์ CastStudy.gast เลือก ScanCode ดังภาพที่ ก-14



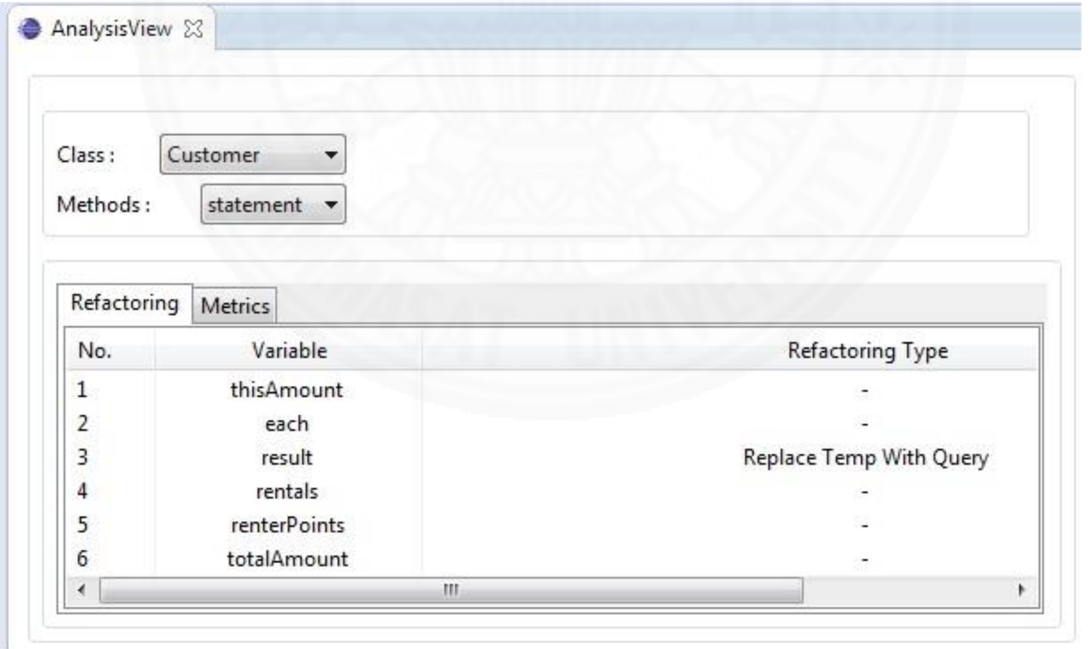
ภาพที่ ก-14. ScanCode

4.9 หลังจาก เลือก ScanCode จะแสดงผลลัพธ์ ดังภาพที่ ก-15 และภาพ ก-16

Ref. code: 25595509035084XRS



ภาพที่ ก-15. แถบ Metrics



ภาพที่ ก-16. แถบ Refactoring

ประวัติผู้เขียน

ชื่อ	นายธีรภัทร์ เสถียรพงษ์
วันเดือนปีเกิด	14 พฤศจิกายน 2529
วุฒิการศึกษา	ปีการศึกษา 2553 : ครุศาสตร์อุตสาหกรรมบัณฑิต มหาวิทยาลัยเทคโนโลยีราชมงคลธัญบุรี

ผลงานทางวิชาการ

ธีรภัทร์ เสถียรพงษ์ และ ทรงศักดิ์ รองวิริยะพานิช. (กรกฎาคม 2560). วิธีการค้นหาร่องรอยที่ไม่ดีประเภทลองเมธอดโดยใช้แบบจำลอง : Model-based Approach for Long Method Bad Smell Detection. The 13th National Conference on Computing and Information Technology (NCCIT 2017), Bangkok.

Ref. code: 25595509035084XRS