การพัฒนาวิธีการตรวจหาความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวา

นายศราวุธ สอนนำ

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยเทคโนโลยีสุรนารี
ปีการศึกษา 2555

THE DEVELOPMENT OF A RUNTIME EXCEPTION CHECKING METHOD FOR JAVA LANGUAGE

Sarawuth Sonnum

A Thesis Submitted in Partial Fulfillment of the Requirements for the

Degree of Master of Engineering in Computer Engineering

Suranaree University of Technology

Academic Year 2012

การพัฒนาวิธีการตรวจหาความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวา

มหาวิทยาลัยเทคโนโลยีสุรนารี อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

คณะกรรมการสอบวิทยานิพนธ์

____

(รศ. ดร.กิตติศักดิ์ เกิดประสพ)
ประธานกรรมการ

____

(ผศ. ดร.พิชโยทัย มหัทธนาภิวัฒน์)
กรรมการ (อาจารย์ที่ปรึกษาวิทยานิพนธ์)

____

(ผศ. ดร.ปรเมศวร์ ห่อแก้ว)
กรรมการ

มหาวิทยาลัยเทคโนโลยีสุรนารี

____

(ศ. ดร.ชูกิจ ลิมปิจำนงค์)
รองอธิการบดีฝ่ายวิชาการ

____

(รศ. ร.อ. ดร.กนต์ธร ชำนิประศาสน์)
คณบดีสำนักวิชาวิศวกรรมศาสตร์

ศราวุธ สอนนำ : การพัฒนาวิธีการตรวจหาความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวา (THE DEVELOPMENT OF A RUNTIME EXCEPTION CHECKING METHOD FOR JAVA LANGUAGE) อาจารย์ที่ปรึกษา : ผู้ช่วยศาสตราจารย์ ดร. พิชโยทัย มหัทธนาภิวัฒน์, 177 หน้า.

ในปัจจุบันการพัฒนาโปรแกรมโดยใช้ภาษาจาวาเป็นที่นิยมและแพร่หลายมากยิ่งขึ้น เช่น การนำโปรแกรมภาษาจาวาเข้าไปใช้ในวงการอุตสาหกรรม วิทยาศาสตร์ และเศรษฐศาสตร์ เป็นต้น อีกทั้งในด้านการศึกษายังให้ความสำคัญกับภาษาจาวาเป็นอย่างมาก โดยจะเห็นจากมีการบรรจุการเขียนโปรแกรมด้วยภาษาจาวาเข้าไปในหลักสูตรของวิศวกรรมคอมพิวเตอร์ วิทยาศาสตร์คอมพิวเตอร์ และสาขาวิชาอื่นๆที่เกี่ยวข้อง เนื่องจากความนิยมที่มีจำนวนมากของภาษาจาวาจึงทำให้มีการพัฒนาซอฟต์แวร์ และนำออกมาเผยแพร่กันอย่างแพร่หลาย อย่างไรก็ตามบางกรณีจะพบเห็นบางซอฟต์แวร์ที่มีความผิดพลาดอยู่ ซึ่งอาจก่อให้เกิดความเสียหายแก่ทรัพยากร (เวลา ข้อมูล และรายได้ เป็นต้น) ของผู้ใช้งานหรือขององค์กรได้ ดังนั้นจึงได้เกิดแนวคิดในการทำวิจัยนี้ขึ้นเพื่อศึกษาวิธีการตรวจสอบความผิดพลาดก่อนการนำซอฟต์แวร์ไปใช้งาน โดยเฉพาะความผิดพลาดประเภทที่เกิดขึ้นขณะที่ซอฟต์แวร์กำลังประมวลผล (Runtime exception) ที่อาจเกิดขึ้นได้ โดยหัวใจหลักสำคัญคือการศึกษาหาความเป็นไปได้ในการใช้ไบต์โค๊ดเพื่อตรวจหาความผิดพลาดของซอฟต์แวร์ภาษาจาวา และหาข้อดีข้อเสียต่างๆที่เกิดขึ้นจากการใช้ไบต์โค้ด เพราะถ้าหากสามารถใช้ไบต์โค้ดในการทดสอบซอฟต์แวร์ได้ นั่นหมายความว่าไม่จำเป็นต้องอาศัยซอสโค้ดในการดำเนินการซึ่งจะทำให้การทดสอบซอฟต์แวร์สะดวกมากยิ่งขึ้น อีกทั้งหากสามารถตรวจสอบความผิดพลาดก่อนที่จะเกิดขึ้นจริงได้แล้วจะเป็นการลดความเสี่ยงในการสูญเสียทรัพยากรของผู้ใช้งานหรือองค์กรที่นำซอฟต์แวร์ไปใช้งาน และยังเป็นการลดภาระของวิศวกรผู้ทดสอบซอฟต์แวร์ นอกจากนั้นยังสามารถลดระยะเวลาในขั้นตอนการตรวจสอบซอฟต์แวร์ได้อีกด้วย ซึ่งนับว่าเป็นประโยชน์ต่อทั้งผู้ใช้ และผู้พัฒนาซอฟต์แวร์เป็นอย่างยิ่ง

สาขาวิชา วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนักศึกษา______________________
ปีการศึกษา 2555	ลายมือชื่ออาจารย์ที่ปรึกษา________________

มหาวิทยาลัยเทคโนโลยีสุรนารี

SARAWUTH SONNUM : THE DEVELOPMENT OF A RUNTIME EXCEPTION CHECKING METHOD FOR JAVA LANGUAGE. THESIS ADVISOR : ASST. PROF. PICHAYOTAI MAHATTHANAPIWAT, Ph.D., 177 PP.

RUNTIME EXCEPTION CHECKING/ JAVA LANGUAGE

At present, software development using java language is widely used, for instance, in the field of industry, science and economics. In addition, education field interests in java language containing it in the course of computer engineering, computer science and other involved departments. From the popularity of java language, many software developers use it to develop java software. However, we can find failures in running software that cause loss in resource, i.e., time, data, revenue, etc., of user or organization. This research purposes java software checking method, especially during runtime for runtime exception. The key importance is the feasibility study of using Bytecode for runtime exception checking and the pros and cons of using byte code. We can use Bytecode as an input for software testing instead of using source code. It will provide more convenient in software testing phase if it is able to locate the possible runtime exception, causing the reduction of the risk of resource loss. Furthermore, the software tester will reduce time and workload in testing java software. The runtime exception checking method will have benefits for both of users and software developers.

School of Computer Engineering Student's Signature__________________

Academic Year 2012 Advisor's Signature__________________

กิตติกรรมประกาศ

วิทยานิพนธ์นี้สามารถสำเร็จลุล่วงด้วยดี ผู้วิจัยขอกราบขอบพระคุณบุคคล และกลุ่มบุคคลต่างๆที่กรุณาให้คำปรึกษา แนะนำ ช่วยเหลืออย่างดียิ่ง ทั้งในด้านวิชาการ และด้านการดำเนินงานวิจัย ซึ่งมีผู้ช่วยศาสตราจารย์ ดร. พิชโยทัย มหัทธนาภิวัฒน์ อาจารย์ที่ปรึกษาวิทยานิพนธ์ รองศาสตราจารย์ ดร. กิตติศักดิ์ เกิดประสพ และรองศาสตราจารย์ ดร. นิตยา เกิดประสพ อาจารย์ประจำสาขาวิศวกรรมคอมพิวเตอร์ สำนักวิชาวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีสุรนารี และท่านอาจารย์ ดร.ชาญวิทย์ แก้วกสิ ที่กรุณาให้คำปรึกษาด้านวิชาการ

ท้ายนี้ ขอกราบขอบพระคุณบิดา มารดา ที่ให้การอบรมเลี้ยงดูและส่งเสริมการศึกษาเป็นอย่างดีมาโดยตลอด จนทำให้ผู้วิจัยประสบความสำเร็จในชีวิตตลอดมา

ศราวุธ สอนนำ

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญ

หน้า

บทคัดย่อ (ภาษาไทย) ..... ก
บทคัดย่อ (ภาษาอังกฤษ) ..... ข
กิตติกรรมประกาศ ..... ค
สารบัญ ..... ง
สารบัญตาราง ..... ฌ
สารบัญรูป ..... ญ

บทที่

1 บทนำ 1
1.1 ความสำคัญและที่มาของปัญหาการวิจัย 1
1.2 วัตถุประสงค์ของการวิจัย 2
1.3 สมมุติฐานการวิจัย 2
1.4 ข้อตกลงเบื้องต้น 2
1.5 ขอบเขตของการวิจัย 3
1.6 ประโยชน์ที่คาดว่าจะได้รับ 4
1.7 คำอธิบายศัพท์ 4

2 ปริทัศน์วรรณกรรมและงานวิจัยที่เกี่ยวข้อง 6
2.1 Exception 6
2.1.1 ความผิดพลาดที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง (Class cast exception) 9
2.1.2 ความผิดพลาดประเภทการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด (Array index out of bound exception) 9

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญ (ต่อ)

หน้า

2.1.3 ความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ (Arithmetic : Divided by zero exception) 10

2.1.4 ความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง (Number format exception) 10

2.1.5 ความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด (String index out of range) 11

2.2 ไบต์โค้ดของภาษาจาวา (Java Bytecode) 12

2.3 การเปรียบเทียบรูปแบบ (Pattern matching) 18

2.3.1 ตัวอักขระพิเศษในการเปรียบเทียบรูปแบบ (Schwartz, R., T. Christiansen, et al., 1997) 18

2.3.2 Anchoring pattern (Schwartz, R., T. Christiansen, et al., 1997) 20

2.3.3 อักขระที่กำหนดไว้ล่วงหน้าในการเปรียบเทียบรูปแบบ (Humbad, S. N., 2004) 20

2.3.4 ลำดับความสำคัญในการเปรียบเทียบรูปแบบ 21

2.4 ตารางคำสั่งปฏิบัติการของไบต์โค้ด (Java Bytecode instruction table) 21

2.5 ฐานข้อมูล H2 (H2 Database) 22

2.5.1 คุณสมบัติเด่นของฐานข้อมูล H2 23

2.5.2 คุณสมบัติของฐานข้อมูล H2 เมื่อเทียบกับกับฐานข้อมูลอื่นๆ 23

2.5.3 รูปแบบการเชื่อมต่อของฐานข้อมูล H2 25

2.5.4 การกำหนด URL เพื่อกำหนดโหมดการเชื่อมต่อ 27

สารบัญ (ต่อ)

หน้า

2.5.5 การเคลื่อนย้าย และเปลี่ยนชื่อฐานข้อมูล....29
2.5.6 โหมดความเข้ากันได้กับฐานข้อมูลอื่นๆ....30
2.5.7 การเชื่อมต่อฐานข้อมูล H2....30
2.6 โปรแกรม FindBugs....31
2.6.1 การติดตั้งโปรแกรมส่วนเสริม FindBugs ลงบน Eclipse....32
2.6.2 การใช้งานโปรแกรมส่วนเสริม FindBugs บน Eclipse....34
2.7 งานวิจัยที่เกี่ยวข้อง....35
3 วิธีดำเนินการวิจัย....41
3.1 วิธีวิจัย....41
3.1.1 การทำงานของไบต์โค้ด....41
3.1.2 ลักษณะคำสั่งของไบต์โค้ด....44
3.1.3 การแบ่งชุดคำสั่งของไบต์โค้ด....45
3.1.4 การค้นหารูปแบบการเกิดความผิดพลาดของชุดคำสั่งไบต์โค้ด....47
3.1.5 การสร้างเครื่องมือตัวอย่างสำหรับการค้นหาความผิดพลาดในภาษาจาวาด้วยไบต์โค้ด....51
3.1.6 ขั้นตอนการทำงานของเครื่องมือตรวจหาความผิดพลาดของภาษาจาวาโดยใช้ไบต์โค้ด....53
3.2 เครื่องมือที่ใช้ในการวิจัย....54
3.3 การเก็บรวบรวมข้อมูล....54
3.3.1 การเก็บรวบรวมข้อมูลจากสื่ออิเล็กทรอนิกส์บนอินเทอร์เน็ต....54

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญ (ต่อ)

หน้า

3.3.2 การเก็บรวบรวมข้อมูลจากการทดลองด้วยโปรแกรมประยุกต์ 55
3.4 การวิเคราะห์ข้อมูล 55
3.5 การออกแบบวิธีการตรวจหาความผิดพลาดในภาษาจาวาโดยใช้ไบต์โค้ด
เพื่อสร้างเครื่องมือตัวอย่าง 56
3.5.1 คลาส H2db 56
3.5.2 คลาส DBFunc 57
3.5.3 คลาส IFile 57
3.5.4 คลาส ByteCode 58
3.5.5 คลาส JPattern 58
3.5.6 คลาส ruleFunc 59
3.5.7 คลาส BCpool 59
3.5.8 คลาส tempStruct 60
3.5.9 คลาส ReportStruct 61
3.6 การทำงานของเครื่องมือตรวจหาความผิดพลาดในภาษาจาวาด้วยไบต์โค้ด 63
3.7 เครื่องมือตัวอย่างและการนำไปใช้งาน 71
4 ผลการวิเคราะห์ข้อมูลและอภิปรายผล 75
4.1 อุปกรณ์ในการทดสอบ 75
4.2 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดที่เกิดจากการแปลง
วัตถุของภาษาจาวาอย่างไม่ถูกต้อง (Class cast exception) 76
4.3 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดประเภทการเรียกใช้
อาร์เรย์เกินขอบเขตที่กำหนด (Array index out of bound exception) 85

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญ (ต่อ)

หน้า

4.4 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ (Arithmetic : Divided by zero exception) 99

4.5 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง (Number format exception).. 120

4.6 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด (String index out of range).. 122

4.7 สรุปผลการทดสอบเครื่องมือตัวอย่างด้วยกรณีทดสอบ (Test case) 125

5 สรุปผลการวิจัยและข้อเสนอแนะ 127

5.1 สรุปผลการวิจัย 127

5.1.1 ข้อจำกัดที่พบหลังการทดสอบเครื่องมือตัวอย่าง 127

5.1.2 ข้อดี-ข้อเสียของวิธีการค้นหาความผิดพลาดด้วยการเปรียบเทียบรูปแบบ 128

5.2 ข้อเสนอแนะ 129

รายการอ้างอิง 132

ภาคผนวก

ภาคผนวก ก. ตารางคำสั่งไบต์โค้ดในภาษาจาวา..135

ภาคผนวก ข. สรุปรายการกรณีทดสอบ (Test case) ที่ใช้ทดสอบเครื่องมือตัวอย่างJEPM 1.0 161

ภาคผนวก ค. บทความทางวิชาการที่ได้รับการตีพิมพ์เผยแพร่ในระหว่างศึกษา 169

ประวัติผู้เขียน 177

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญตาราง

ตารางที่ **หน้า**

2.1 การเปรียบเทียบคุณสมบัติของฐานข้อมูล H2 กับฐานข้อมูลอื่น 23
2.2 URL ที่ใช้ในการกำหนดการเชื่อมต่อฐานข้อมูล H2....................27
2.3 ไฟล์ที่เกี่ยวข้องกับฐานข้อมูล H2....................29
2.4 สรุปเปรียบเทียบงานวิจัยที่เกี่ยวข้องกับการการพัฒนาวิธีการตรวจหา
ความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวา....................39
4.1 ตารางแสดงผลการทดสอบการค้นหาความผิดพลาดประเภทที่เกิดจาก
การแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง....................82
4.2 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดการ
เรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวา....................95
4.3 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดจากการ
หารด้วยศูนย์....................114
4.4 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดจากการ
จัดรูปแบบตัวเลขที่ไม่ถูกต้อง 118
4.5 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดที่เกิดจากการใช้งาน
สตริงเกินขอบเขตที่กำหนด 122
4.6 สรุปผลลัพธ์การทดสอบเครื่องมือตัวอย่างด้วยความผิดพลาดทั้ง 5 ประเภท....................125

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญรูป

รูปที่ **หน้า**

2.1 ลำดับชั้นของคลาส Exception ในภาษาจาวา (Sun Microsystems Inc., 2005) 7

2.2 ตัวอย่างของ Exception handling ในภาษาจาวา (Robillard, M. P. and G. C. Murphy, 2003) 8

2.3 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษา จาวาอย่างไม่ถูกต้อง 9

2.4 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด........................10

2.5 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ 10

2.6 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง 11

2.7 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทการใช้งานสตริงเกินขอบเขตที่กำหนด........................11

2.8 การสร้างไบต์โค้ดของภาษาจาวาผ่าน Command line 12

2.9 ตัวอย่างไบต์โค้ดที่ได้จากการใช้คำสั่ง javap 14

2.10 ตัวอย่างการแยกคำสั่งปฏิบัติการ (Opcode) ของไบต์โค้ดภาษาจาวา 15

2.11 การนำโปรแกรมภาษาจาวาไปใช้งาน (Gupta, R., 2006)........................16

2.12 ลักษณะของแบบจำลองเฟรมของจาวาเวอร์ชวลแมชีน (Haggar, P., 2006) 17

2.13 ตัวอย่างการเปรียบเทียบรูปแบบในภาษา Perl 18

2.14 ตัวอย่างการใช้อักขระ + ในการเปรียบเทียบรูปแบบ 19

2.15 สัญลักษณ์ของฐานข้อมูล H2 (Muller, T., 2006)........................22

2.16 การทำงานของฐานข้อมูล H2 ในโหมดฝังกับโปรแกรม 25

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญรูป (ต่อ)

รูปที่ **หน้า**

2.17 การทำงานของฐานข้อมูล H2 ในโหมดแม่ข่าย....26
2.18 การทำงานของฐานข้อมูล H2 ในโหมดผสม....27
2.19 ตัวอย่างคำสั่งในการเชื่อมต่อฐานข้อมูล H2....30
2.20 หน้าต่างการติดตั้งส่วนเสริมความสามารถ FindBugs ลงบนโปรแกรมEclipse....32
2.21 หน้าต่างแจ้งเตือนหลังการติดตั้งส่วนเสริม FindBugs เสร็จ....33
2.22 หน้าต่างแสดงข้อตกลงเบื้องต้นในการใช้งานส่วนเสริม FindBugs....33
2.23 การเริ่มต้นใช้งานโปรแกรมส่วนเสริม FindBugs....34
2.24 ผลลัพธ์ของการทดสอบซอสโค้ดด้วยส่วนเสริม FindBugs....35
2.25 ประเภทของความผิดพลาดที่นำมาศึกษาในงานวิจัยของ Hovemeyer and Pugh (2004)....36
3.1 การคอมไพล์โปรแกรมในภาษาจาวา....41
3.2 ตัวอย่างซอสโค้ดของภาษาจาวา....42
3.3 ผลลัพธ์เมื่อใช้คำสั่ง javap –c....42
3.4 ส่วนประกอบต่างๆในไบต์โค้ด....43
3.5 ตัวอย่างคำสั่งแบบปรกติของไบต์โค้ด....44
3.6 ตัวอย่างคำสั่งที่อ้างอิงถึงตำแหน่งข้อมูลของไบต์โค้ด....44
3.7 ตัวอย่างคำสั่งที่อ้างอิงถึงคำสั่งภายใน....45
3.8 ผลลัพธ์การรันคำสั่ง javap –l....46
3.9 ตัวอย่างคำสั่งของไลน์โค้ด....46
3.10 การแบ่งชุดคำสั่งของไบต์โค้ดโดยอาศัยไลน์โค้ด....47
3.11 ตัวอย่างโปรแกรม Eclipse กับส่วนเสริม Bytecode outline....48
3.12 ตัวอย่างชุดคำสั่งของไบต์โค้ด....49
3.13 การแปลงชุดคำสั่งไบต์โค้ดเพื่อจัดเก็บในฐานข้อมูลตรวจสอบ....50

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญรูป (ต่อ)

รูปที่ **หน้า**

3.14 ลักษณะข้อมูลที่ถูกจัดเก็บอยู่ในฐานข้อมูลตรวจสอบ....51
3.15 ตัวอย่างโครงสร้างของโปรเจค (Java project) ที่ถูกสร้างโดย Eclipse....52
3.16 โครงสร้างของคลาส H2db....56
3.17 โครงสร้างของคลาส DBFunc....57
3.18 โครงสร้างของคลาส IFile....57
3.19 โครงสร้างของคลาส ByteCode....58
3.20 โครงสร้างของคลาส JPattern....58
3.21 ลักษณะการทำงานของคลาส JPattern....59
3.22 โครงสร้างของคลาส ruleFunc....59
3.23 โครงสร้างของคลาส BCpool และ asbBCpool....60
3.25 โครงสร้างของคลาส tempStruct....61
3.26 คลาสไดอะแกรมของเครื่องมือตรวจหาความผิดพลาดของภาษาจาวาด้วย
ไบต์โค้ด....62
3.28 ขั้นตอนการแปลงและจัดรูปแบบไฟล์นามสกุล .class....65
3.29 ตัวอย่างการทำงานและลักษณะข้อมูลที่ได้จากขั้นตอนการแปลงและจัด
รูปแบบไฟล์นามสกุล .class....66
3.30 ขั้นตอนการตรวจหาไบต์โค้ดที่อาจทำให้เกิดความผิดพลาด....67
3.31 ตัวอย่างการทำงานและลักษณะข้อมูลที่ได้จากขั้นตอนการตรวจหาไบต์โค้ด
ที่อาจทำให้เกิดความผิดพลาด....68
3.32 ขั้นตอนการตรวจสอบกฎการเกิดความผิดพลาด....69
3.33 ตัวอย่างการทำงานและลักษณะข้อมูลที่ได้จากขั้นตอนการตรวจสอบกฎ
การเกิดความผิดพลาด....70
3.34 ส่วนประกอบของเครื่องมือตัวอย่าง JEPM1.0....71

สารบัญรูป (ต่อ)

รูปที่ หน้า

3.35 การคัดลอกไฟล์ของเครื่องมือตัวอย่าง 72
3.36 การนำเข้าคลังโปรแกรมของเครื่องมือตัวอย่าง 73
3.37 การเขียนซอสโค้ดในเพื่อเรียกใช้เครื่องมือตัวอย่าง 74
3.38 ผลลัพธ์จากการทดสอบเครื่องมือในลักษณะคลังโปรแกรม 74
4.1 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อด main 76
4.2 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main 76
4.3 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง 77
4.4 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง 77
4.5 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อด main 78
4.6 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main 78
4.7 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง 79

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญรูป (ต่อ)

รูปที่ **หน้า**

4.8 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง 79
4.9 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main 80
4.10 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง 80
4.11 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง 81
4.12 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง 83
4.13 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องด้วยเครื่องมือตัวอย่าง 84
4.14 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อด main โดยใช้ตัวแปรระดับคลาส 85
4.15 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main 85

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญรูป (ต่อ)

รูปที่ หน้า

4.16 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้ อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียก เมท็อดย่อยซ้อนจำนวน 1 ครั้ง....86
4.17 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้ อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียก เมท็อดย่อยซ้อนจำนวน 2 ครั้ง....86
4.18 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้ อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อด main โดยใช้ตัวแปร ภายใน 87
4.19 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้ อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อด main โดยใช้ตัวแปร ภายในลูป for....87
4.20 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้ อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อด main โดยใช้ตัวแปร ภายในเมท็อด 88
4.21 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้ อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อดอื่นที่มีการเรียกใช้ โดยเมท็อด main....88
4.22 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้ อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อดอื่นที่มีการเรียกใช้ โดยเมท็อด main โดยใช้ตัวแปรภายในลูป for อย่างง่าย....89

สารบัญรูป (ต่อ)

รูปที่ หน้า

4.23 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อดอื่นที่มีการเรียกใช้โดยเมท็อด main โดยใช้ตัวแปรภายในเมท็อด 89
4.24 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง 90
4.25 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียกเมท็อดย่อยจำนวน 2 ครั้ง 90
4.26 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง ซึ่งเกิดความผิดพลาดภายในลูปที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน 91
4.27 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง ซึ่ง เกิดความผิดพลาดภายในลูปที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน 91
4.28 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง ซึ่งเกิดความผิดพลาดภายในลูปที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายใน 92

สารบัญรูป (ต่อ)

รูปที่ หน้า

4.29 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้งซึ่งเกิดความผิดพลาดภายในลูปที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายใน 92
4.30 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษา จาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main 93
4.31 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง 93
4.32 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง 94
4.33 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวา 97
4.34 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวาด้วยเครื่องมือตัวอย่าง 98
4.35 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยความผิดพลาดที่เกิดขึ้นสามารถสังเกตได้ง่าย 99
4.36 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีค่าเป็นศูนย์ 99

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญรูป (ต่อ)

รูปที่ หน้า

4.37 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีการเพิ่มค่าภายในลูป for 100
4.38 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีการลดค่าภายในลูป for 100
4.40 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าในลูป for 101
4.41 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าในลูป for 102
4.42 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้ง 102
4.43 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปรที่เป็นตัวหารมีการเพิ่มค่าภายในลูป for 103
4.44 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปรที่เป็นตัวหารมีการลดค่าภายในลูป for 103
4.45 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 2 ครั้ง 104
4.46 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยความผิดพลาดที่เกิดขึ้นสามารถสังเกตได้ง่าย 104

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญรูป (ต่อ)

รูปที่ หน้า

4.47 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีค่าเป็นศูนย์ 105
4.48 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีการเพิ่มค่าภายในลูป for 105
4.49 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีการลดค่าภายในลูป for 106
4.50 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์ 106
4.51 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่า ในลูป for 107
4.52 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่า ในลูป for 107
4.53 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้ง 108
4.54 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปร ที่เป็นตัวหารมีการเพิ่มค่าภายในลูป for 108
4.55 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปร ที่เป็นตัวหารมีการลดค่าภายในลูป for 109

สารบัญรูป (ต่อ)

รูปที่ **หน้า**

4.56 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 2 ครั้ง....109
4.57 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์....110
4.58 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่า ในลูป for....110
4.59 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่า ในลูป for....111
4.60 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้ง....111
4.61 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปร ที่เป็นตัวหารมีการเพิ่มค่าภายในลูป for....112
4.62 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปร ที่เป็นตัวหารมีการลดค่าภายในลูป for....112
4.63 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 2 ครั้ง....113
4.64 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดประเภท ที่เกิดจากการหารด้วยศูนย์....116

สารบัญรูป (ต่อ)

รูปที่ หน้า

4.65 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาด
ประเภทที่เกิดจากการหารด้วยศูนย์ด้วยเรื่องมือตัวอย่าง 117
4.66 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาด
ประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง 120
4.67 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาด
ประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้องด้วยเรื่องมือตัวอย่าง....... 121
4.68 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดที่เกิด
จากการใช้งานสตริงเกินขอบเขตที่กำหนด 123
4.69 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาดที่
เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนดด้วยเรื่องมือตัวอย่าง....... 124
4.70 แผนภูมิแท่งแสดงจำนวนความผิดพลาดจำแนกตามตำแหน่งที่เกิดความ
ผิดพลาด....... 126
5.1 ตัวอย่างของไบต์โค้ดที่มีการส่งผ่านพารามิเตอร์ในการเรียกใช้เมท็อด....... 128
5.2 ตัวอย่างของซอสโค้ดที่มีคำสั่งไม่พึงประสงค์ 130
5.3 ตัวอย่างของไบต์โค้ดที่มีคำสั่งไม่พึงประสงค์....... 131
5.4 ตัวอย่างการประยุกต์เครื่องมือ JEPM กับโปรแกรมไม่พึงประสงค์....... 131

มหาวิทยาลัยเทคโนโลยีสุรนารี

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มาของปัญหาการวิจัย

ในยุคปัจจุบันเทคโนโลยีด้านซอฟต์แวร์ได้เข้ามามีบทบาทสำคัญทั้งในชีวิตประจำวันและด้านอื่นๆของมนุษย์ จึงผลักดันให้เกิดการพัฒนาซอฟต์แวร์ใหม่ๆขึ้นมาจำนวนมากเพื่อตอบสนองความต้องการของมนุษย์ อีกทั้งได้ก่อให้เกิดบุคลากรทางด้านการพัฒนาซอฟต์แวร์ที่มากขึ้นอีกด้วย แต่เนื่องจากเทคโนโลยีได้พัฒนาไปอย่างรวดเร็วทำให้ในบางครั้ง ซอฟต์แวร์ที่ผลิตขึ้นนั้นมีความผิดพลาด (Exception) เกิดขึ้น ซึ่งอาจเกิดจากการขาดความรู้หรือประสบการณ์ที่น้อยเกินไปของผู้พัฒนาซอฟต์แวร์ ดังนั้นจึงได้เกิดเป็นแรงบันดาลใจให้มีการจัดทำการศึกษาวิจัยนี้ โดยในการวิจัยครั้งนี้จะเป็นการวิจัยเกี่ยวกับวิธีการที่ช่วยให้ผู้พัฒนาโปรแกรมสามารถตรวจสอบความผิดพลาดที่อาจจะเกิดขึ้นในการเขียนซอสโค้ด (Source code) ของซอฟต์แวร์ในภาษาจาวาซึ่งเป็นภาษาที่ได้รับความนิยมในการพัฒนาซอฟต์แวร์ โดยในภาษาจาวานั้นได้มีการแบ่งความผิดพลาดที่อาจเกิดขึ้นออกเป็น 3 กรณีตาทฤษฎีของ Pugh, W. (2007) หลักๆดังนี้

1. ความผิดพลาดที่เกิดจากไวยากรณ์ของภาษาที่ไม่ถูกต้อง (Syntax Errors)
2. ความผิดพลาดในระหว่างการรันโปรแกรม (Runtime Errors)
3. ความผิดพลาดที่เกิดจากตรรกะของผู้พัฒนา (Logic Errors)

ในการวิจัยครั้งนี้มุ่งเน้นให้ความสำคัญไปที่การพัฒนาเครื่องมือที่ใช้ตรวจสอบความผิดพลาด (Exception) แบบความผิดพลาดในระหว่างการรันโปรแกรม (Runtime Errors) เป็นหลัก เนื่องจากในภาษาจาวาได้มีการพัฒนาให้สามารถตรวจสอบความผิดพลาดบางอย่างได้แล้ว เช่น ความผิดพลาดประเภทที่เกิดจากไวยากรณ์ของภาษาที่ไม่ถูกต้อง (Syntax Errors) เป็นต้น อย่างไรก็ตามความผิดพลาดในระหว่างการรันโปรแกรมที่เกิดขึ้นยังไม่มีการพัฒนาให้สามารถตรวจสอบได้ จึงทำให้เมื่อมีความผิดพลาดเกิดขึ้นโปรแกรมอาจหยุดการทำงาน หรือทำงานไม่ถูกต้อง ดังนั้นหากสามารถสร้างวิธีการที่สามารถตรวจสอบความผิดพลาดในระหว่างการรันโปรแกรม จะเป็น

ประโยชน์ต่อผู้พัฒนาเป็นอย่างมาก อีกทั้งเป็นการลดความผิดพลาดของโปรแกรมต่างที่ถูกพัฒนาขึ้นด้วยภาษาจาวาที่นำออกมาใช้อีกด้วย ซึ่งจะเป็นผลดีต่อผู้ใช้ในแง่ของการลดความเสี่ยงในการสูญเสียทรัพยากร (เวลา ข้อมูล และรายได้ เป็นต้น)

1.2 วัตถุประสงค์ของการวิจัย

1.2.1 เพื่อศึกษาความเป็นไปได้ในการใช้ไบต์โค้ดแทนซอสโค้ดในขั้นตอนการทดสอบซอฟต์แวร์

1.2.2 เพื่อศึกษาทดลองหาวิธีการตรวจหาความผิดพลาดในระหว่างการรันโปรแกรม (Runtime exception) ประเภทต่างๆที่เกิดขึ้นในภาษาจาวาโดยใช้ไบต์โค้ด

1.2.3 เพื่อทดลองสร้างเครื่องมือตัวอย่างที่ใช้ในการตรวจหาความผิดพลาดประเภทความผิดพลาดในระหว่างการรันโปรแกรม (Runtime exception) สำหรับใช้ทดลองค้นหาความผิดพลาดเพื่อนำผลลัพธ์การทดลองมาวิเคราะห์และสรุปผลได้

1.2.4 เพื่อเพิ่มความถูกต้องในการตรวจสอบซอฟต์แวร์ในภาษาจาวาให้ดียิ่งขึ้น

1.3 สมมุติฐานการวิจัย

1.3.1 หากสามารถค้นหาความผิดพลาดในโปรแกรมภาษาจาวาโดยใช้ไบต์โค้ดได้ จะสามารถใช้ไบต์โค้ดแทนซอสโค้ดในการทดสอบโปรแกรมภาษาจาวาได้

1.3.2 หากสามารถค้นหาความผิดพลาดในภาษาจาวาด้วยไบต์โค้ดได้แล้ว จะสามารถนำวิธีการที่คิดค้นขึ้นไปพัฒนาเป็นเครื่องมือที่ใช้ในการตรวจสอบความผิดพลาดในภาษาจาวาได้

1.3.3 วิธีการที่สร้างขึ้นสามารถทำงานได้โดยไม่ต้องอาศัยซอสโค้ดของภาษาจาวา เพียงแค่ใช้ไฟล์นามสกุล .class ที่ได้จากการคอมไพล์แล้วเท่านั้น

1.4 ข้อตกลงเบื้องต้น

1.4.1 วิธีการที่พัฒนาขึ้นนี้สามารถตรวจสอบความผิดพลาดในระหว่างการรันโปรแกรม (Runtime Errors) บางประเภท ซึ่งในงานวิจัยนี้จะยกตัวอย่างความผิดพลาดประเภทความผิดพลาดที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง (Class cast exception) ความผิดพลาดประเภทการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด (Array index out of bound exception) ความผิดพลาด

ประเภทที่เกิดจากการหารด้วยศูนย์ (Arithmetic : Divided by zero exception) ความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง (Number format exception) และความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด (String index out of range) เท่านั้น

1.4.2 วิธีการที่พัฒนาขึ้นนี้สามารถตรวจสอบไบต์โค้ด (Bytecode) ของภาษาจาวาในไฟล์นามสกุล .class ได้เท่านั้น

1.4.3 วิธีการที่พัฒนาขึ้นนี้สามารถตรวจสอบความผิดพลาดในไบต์โค้ดและแจ้งให้ผู้ใช้ทราบเท่านั้น แต่ไม่สามารถแก้ไขความผิดพลาดของไบต์โค้ดได้

1.4.4 ในงานวิจัยครั้งนี้จะสร้างเครื่องมือตัวอย่าง ที่ใช้วิธีการตรวจหาความผิดพลาดในภาษาจาวาที่ได้ออกแบบไว้เพื่อใช้ในการทดลองค้นหาความผิดพลาดเพื่อนำผลลัพธ์มาวิเคราะห์และสรุปผลเท่านั้น

1.5 ขอบเขตของการวิจัย

1.5.1 วิธีการตรวจหาความผิดพลาดที่สร้างขึ้นนี้รองรับเฉพาะไวยากรณ์ของภาษาจาวาเท่านั้น

1.5.2 ในการศึกษาวิจัยการหาวิธีการตรวจหาความผิดพลาดในภาษาจาวาครั้งนี้อ้างอิงคำสั่งไบต์โค้ดในจาวาเวอร์ชวลแมชีน (JVM) เวอร์ชัน 1.6 เท่านั้น

1.5.3 วิธีการตรวจหาความผิดพลาดพัฒนาขึ้นนี้รองรับไบต์โค้ดที่ได้จากไฟล์นามสกุล .class และสามารถนำเข้าประมวลผลได้ครั้งละไฟล์เท่านั้น

1.5.4 วิธีการตรวจหาความผิดพลาดในภาษาจาวาที่สร้างขึ้นนี้สามารถตรวจหาความผิดพลาดในคลาสเพียงคลาสเดียวที่ไม่มีการอ้างอิงถึงคลาสอื่นเท่านั้น

1.5.5 เครื่องมือตัวอย่างที่ทดลองพัฒนาขึ้นนี้ไม่สามารถแก้ไขความผิดพลาดที่ตรวจพบได้เพียงแต่แจ้งเตือนให้ทราบเท่านั้น

มหาวิทยาลัยเทคโนโลยีสุรนารี

1.6 ประโยชน์ที่คาดว่าจะได้รับ

1.6.1 สามารถใช้ไบต์โค้ดแทนซอสโค้ดในการทดสอบซอฟต์แวร์ของจาวาได้ในกรณีที่ผู้ทดสอบซอฟต์แวร์ไม่มีซอสโค้ด ซึ่งจะให้ให้ผู้ใช้ทราบถึงความผิดพลาดของโปรแกรมภาษาจาวาก่อนที่จะนำมาใช้ได้

1.6.2 วิธีการที่คิดค้นขึ้นสามารถช่วยลดความผิดพลาดของโปรแกรมที่พัฒนาด้วยภาษาจาวาก่อนที่จะถูกนำไปใช้งานได้

1.6.3 วิธีการที่คิดค้นขึ้นสามารถเป็นอีกทางเลือกในการทดสอบซอฟต์แวร์ของภาษาจาวา

1.6.4 เพิ่มความสะดวกสบายให้แก่ผู้พัฒนาซอฟต์แวร์ด้วยภาษาจาวา เนื่องจากสามารถที่จะตรวจสอบความถูกต้องของซอฟต์แวร์ไปพร้อมกับการพัฒนาซอฟต์แวร์ได้

1.6.5 ลดภาระการตรวจสอบความผิดพลาดของโปรแกรมของผู้ทดสอบซอฟต์แวร์ในขั้นตอนของการทดสอบซอฟต์แวร์ที่พัฒนาด้วยภาษาจาวา

1.7 คำอธิบายศัพท์

1.7.1 ความผิดพลาด (Exception)

หมายถึง ความไม่ถูกต้องภายในโปรแกรมภาษาจาวาที่ส่งผลให้โปรแกรมทำงานผิดปรกติ หรือ หยุดการทำงาน

1.7.2 ไบต์โค้ด (Bytecode)

หมายถึง คำสั่งปฏิบัติการของภาษาจาวาที่บรรจุอยู่ในไฟล์นามสกุล .class ที่ถูกรันด้วยคำสั่ง javap -c จนได้ข้อความที่สามารถอ่านทำความเข้าใจได้

1.7.3 ไลน์โค้ด (Line number)

หมายถึง ข้อความที่บ่งบอกถึงตำแหน่งการแบ่งไบต์โค้ดออกเป็นชุดๆตามหมายเลขบรรทัด โดยไลน์โค้ดจะได้จากการรันคำสั่ง javap –l

1.7.4 สแตก (Operand stack)

หมายถึง โครงสร้างข้อมูลจำลองภายในจาวาเวอร์ชวลแมชีน ที่สามารถนำข้อมูลเข้าหรือออกได้ทางเดียวคือส่วนบนของสแตก(ข้อมูลตัวสุดท้ายที่ถูกเพิ่มเข้าไป)

มหาวิทยาลัยเทคโนโลยีสุรนารี

1.7.5 แพทเทิร์น (Pattern)

หมายถึง แต่ละชุดของคำสั่งไบต์โค้ดที่ถูกแบ่งออกเป็นส่วนๆด้วยไลน์โค้ด ซึ่งหนึ่งแพทเทิร์นจะประกอบไปด้วยคำสั่งไบต์โค้ดจำนวนหนึ่ง

1.7.6 โปรเจค (Java project)

หมายถึง โปรแกรมภาษาจาวาที่ถูกพัฒนาอยู่ภายในโปรแกรมประยุกต์เช่น Eclipse หรือ NetBeans โดยภายในโปรเจคจะประกอบไปด้วยไฟล์นามสกุล .java ที่บรรจุซอสโค้ดของโปรแกรมเอาไว้

1.7.7 สอบถามข้อมูล (Query)

หมายถึง การใช้คำสั่ง SQL ในการเข้าถึงข้อมูลในฐานข้อมูลที่ต้องการ

1.7.8 ฐานข้อมูลตรวจสอบ (Check pattern database)

หมายถึง ฐานข้อมูลที่เก็บแพทเทิร์นที่อาจก่อให้เกิดความผิดพลาด และ ข้อมูลอื่นๆที่ใช้ประกอบในการตรวจสอบความผิดพลาด

1.7.9 ฐานข้อมูลนำเข้า (Input pattern database)

หมายถึง ฐานข้อมูลที่เก็บแพทเทิร์น และ ข้อมูลประกอบอื่นๆที่ได้จากการประมวลผลไฟล์นามสกุล .class ของเครื่องมือตัวอย่างที่สร้างขึ้น

มหาวิทยาลัยเทคโนโลยีสุรนารี

บทที่ 2
ปริทัศน์วรรณกรรมและงานวิจัยที่เกี่ยวข้อง

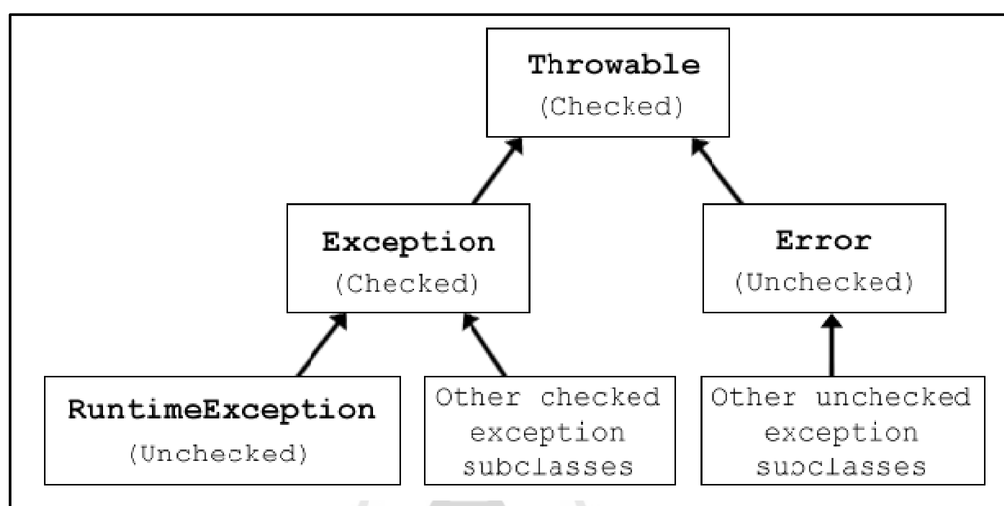
ในบทนี้จะเป็นการนำเสนอทฤษฎีพื้นฐานที่ใช้ในการสร้างวิธีการตรวจหาความผิดพลาด (Exception) ในไบต์โค้ดของภาษาจาวา (Java Bytecode) โดยในหัวข้อแรกจะเป็นการอธิบายเกี่ยวกับความผิดพลาด (Exception) บนภาษาจาวา ในหัวข้อที่ 2 จะอธิบายถึงไบต์โค้ดของภาษาจาวาว่ามีโครงสร้างและหลักการทำงานอย่างไร เพื่อให้สามารถเข้าใจหลักการและนำไปใช้สร้างวิธีการตรวจหาความผิดพลาดได้อย่างเหมาะสม ในหัวข้อที่ 3 จะอธิบายถึงทฤษฎีของการเปรียบเทียบรูปแบบ (Pattern matching) ซึ่งจะเป็นวิธีการที่จะนำมาประยุกต์ใช้กับไบต์โค้ดเพื่อสร้างเป็นวิธีการค้นหาไบต์โค้ดที่ทำให้เกิดความผิดพลาดขึ้นมา ในหัวข้อที่ 4 จะอธิบายถึงรายละเอียดของคำสั่งไบต์โค้ดเพื่อนำไปสร้างวิธีการค้นหาความผิดพลาดในไบต์โค้ดของภาษาจาวาด้วยวิธีการตีความชุดคำสั่งบางคำสั่งของไบต์โค้ดเพื่อนำมาเพิ่มประสิทธิภาพของวิธีการตรวจหาความผิดพลาดในไบต์โค้ดของภาษาจาวาโดยใช้วิธีการเปรียบเทียบรูปแบบ เพื่อให้ได้วิธีการที่มีประสิทธิภาพในการค้นหาความผิดพลาดในไบต์โค้ดของภาษาจาวา และในหัวข้อที่ 5 จะนำเสนอเกี่ยวกับฐานข้อมูล H2 ที่จะถูกนำมาใช้ในการจัดเก็บข้อมูลต่างๆเพื่อนำมาใช้ร่วมกับการสร้างเครื่องมือต่อไป

วิทยาลัยเทคโนโลยีสุรนารี

2.1 Exception

Exception เป็นส่วนที่แสดงความผิดพลาดที่เกิดขึ้นในระหว่างที่โปรแกรมทำงาน โดยส่วนจัดการความผิดพลาด (Exception handling) จะถูกใช้เมื่อเมท็อด (Method) หรือส่วนประกอบ (Component) นั้นๆ ไม่สามารถจะทำงานต่อได้ เมื่อมีความผิดพลาดเกิดขึ้น เช่น การหารด้วย 0 การเรียกใช้อาร์เรย์ในตำแหน่งที่ไม่ถูกต้อง และการแปลงประเภทของข้อมูลที่ไม่ถูกต้อง เป็นต้น หากเกิดความผิดพลาดขึ้นส่วนจัดการความผิดพลาดจะถูกใช้สำหรับเมท็อดหนึ่งๆ เพื่อตรวจความผิดพลาดที่ทำให้โปรแกรมไม่สามารถทำงานต่อได้ เมื่อตัวจัดการความผิดพลาด (Exception handler) ตรวจพบความผิดพลาดที่เกิดขึ้นเมท็อดนั้นจะทำการโยนความผิดพลาด (Throw an

exception) หรือทำการรัน โปรแกรมในส่วนที่จะแจ้งความผิดพลาดให้ผู้ใช้ทราบ หรืออาจเป็นการทำงานแบบอื่นๆ ทั้งนี้เพื่อจุดประสงค์ในการทำให้โปรแกรมทำงานได้โดยไม่ต้องหยุดการทำงานกะทันหัน โดยในภาษาจาวาคลาส Exception จะสืบทอดมาจากคลาส Throwable เพื่อทำให้สามารถโยนความผิดพลาดที่เกิดขึ้นระหว่างวัตถุ (Object) ได้ ดังที่ได้แสดงในรูปที่ 2.1

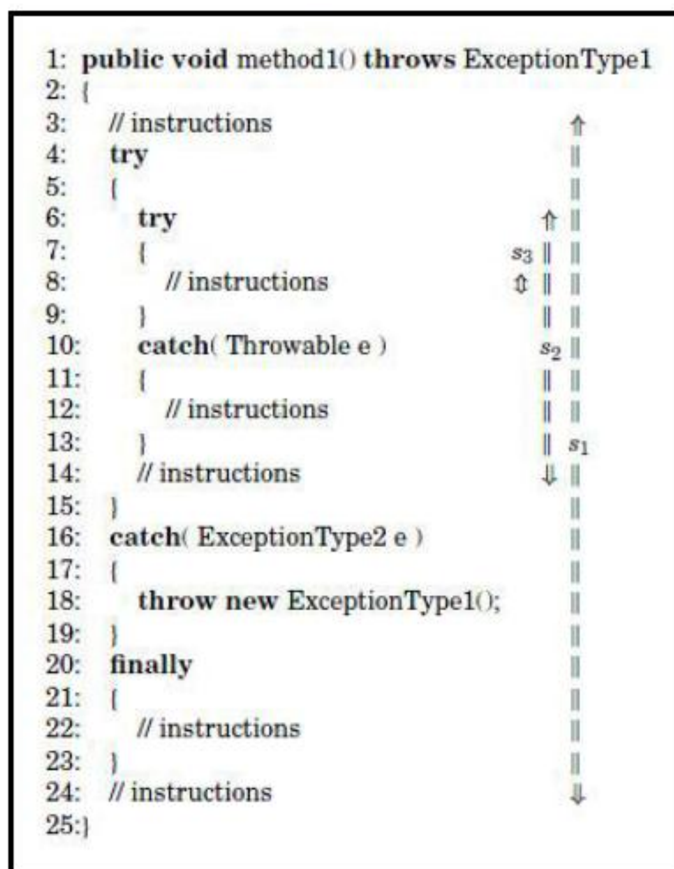


รูปที่ 2.1 ลำดับชั้นของคลาส Exception ในภาษาจาวา (Sun Microsystems Inc., 2005)

ในรูปที่ 2.1 จะเห็นว่าคลาส RuntimeException ได้สืบทอดมาจากคลาส Exception ที่สืบทอดมาจากคลาส Throwable อีกที ดังนั้นเมื่อเกิดความผิดพลาดประเภท Runtime exception ขึ้นจึงสามารถโยนความผิดพลาดระหว่างคลาสได้ แต่ความผิดพลาดประเภท Runtime exception จะมีข้อแตกต่างคือ ไม่สามารถถูกตรวจพบในขณะคอมไพล์ซอฟต์แวร์ จึงไม่สามารถที่จะแก้ไขความผิดพลาดได้ในทันที แต่จะถูกพบในระหว่างที่ซอฟต์แวร์กำลังประมวลผลอยู่ซึ่งอาจต้องกลับมาแก้ไขในภายหลัง (Sun Microsystems Inc., 2005)

ในภาษาจาวามีวิธีการจัดการกับความผิดพลาดเรียกว่า Exception handling เป็นกลไกในการจัดการกับความผิดพลาดที่เกิดขึ้นในโปรแกรม โดยทั่วไปภาษาที่เป็นที่นิยมส่วนใหญ่จะพยายามสร้างส่วนการจัดการความผิดพลาดให้อยู่ในโครงสร้างภาษาของตัวเอง เพื่อให้สามารถคืนสภาพของซอฟต์แวร์ เมื่อเกิดความผิดพลาดขึ้นซึ่งจะเป็นการสร้างมาตรฐานและความสมบูรณ์

ให้แก่ภาษา ในภาษาจาวาก็มีส่วนประกอบนี้อยู่ในโครงสร้างของภาษาเช่นกัน จะเห็นได้จากมีการใช้ try และ catch เข้ามาเพื่อตรวจสอบความผิดพลาดของโครงสร้างภาษา ดังรูปที่ 2.2



```
1: public void method1() throws ExceptionType1
2: {
3:    // instructions
4:    try
5:    {
6:       try
7:       {
8:          // instructions
9:       }
10:      catch( Throwable e )
11:      {
12:         // instructions
13:      }
14:      // instructions
15:   }
16:   catch( ExceptionType2 e )
17:   {
18:      throw new ExceptionType1();
19:   }
20:   finally
21:   {
22:      // instructions
23:   }
24:   // instructions
25:}
```


วิทยาลัยเทคโนโลยีสุร

รูปที่ 2.2 ตัวอย่างของ Exception handling ในภาษาจาวา (Robillard, M. P. and G. C. Murphy, 2003)

Robillard, M. P. และ G. C. Murphy (2003) ได้อธิบายถึงหลักการทำงานของ Exception handling โดยอธิบายว่าในภาษาจาวากำหนดบล็อกที่ชื่อ try ซึ่งภายในบล็อกนี้จะมีคำสั่งต่างๆที่อาจก่อให้เกิดความผิดพลาด และอาจก่อให้เกิด error ขึ้นได้ บล็อก try จะคู่กับบล็อก catch เสมอ catch จะเป็นบล็อกที่คอยดักจับความผิดพลาดโดยตัวจัดการความผิดพลาด (Exception handler) อีกบล็อกหนึ่งที่อาจมีหรือไม่มีก็ได้นั่นคือบล็อก finally ซึ่งเป็นบล็อกที่จะถูกทำเสมอแม้ว่าจะไม่มีความผิดพลาดก็ตาม อย่างไรก็ตามหากบล็อก try ไม่มีบล็อก catch ผู้พัฒนาซอฟต์แวร์ต้องเขียนบล็อก finally กำกับด้วยเสมอ

งานวิจัยนี้ผู้วิจัยได้ทำการคัดเลือกประเภทของความผิดพลาดในภาษาจาวาที่สามารถพบเห็นได้ทั่วไปมาทำการศึกษาวิเคราะห์จำนวน 5 ประเภทดังนี้

2.1.1 ความผิดพลาดที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง (Class cast exception)

คือ ความผิดพลาดที่เกิดจากการนำเอาตัวแปรที่ถูกประกาศเอาไว้เป็นวัตถุของคลาสใดคลาสหนึ่งมาใช้งานหรือแปลงให้อยู่ในรูปของคลาสอื่นๆที่ไม่สามารถรองรับวัตถุของคลาสที่ประกาศเอาไว้ได้ ซึ่งหากวัตถุของคลาสใดๆจะสามารถรองรับวัตถุของคลาสอื่นๆได้คลาสเหล่านั้นต้องมีลักษณะความสัมพันธ์เป็นคลาสแม่-คลาสลูกกัน (Sub class) ตัวอย่างเช่น มีการประกาศตัวแปรที่เป็นวัตถุของคลาส Integer เอาไว้ แต่เมื่อจะนำไปใช้กลับมีการแปลงให้อยู่ในรูปของวัตถุของคลาส String ซึ่งคลาส String ไม่สามารถรองรับวัตถุของคลาส Integer ได้ จึงทำให้เกิดความผิดพลาดประเภทการแปลงวัตถุอย่างไม่ถูกต้องขึ้นดังรูปที่ 2.3 ในบรรทัดที่ 5

```
1  public class ClassCastEx {
2      public static Object num = new Integer(3);
3
4      public static void main(String args[]) {
5          String str = (String) num;
6          System.out.println("String is " + str);
7      }
8  }
```

รูปที่ 2.3 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง

2.1.2 ความผิดพลาดประเภทการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด (Array index out of bound exception)

คือ ความผิดพลาดประเภทที่เกิดขึ้นกับโครงสร้างข้อมูลที่มีลักษณะเป็นอาร์เรย์ (รวมถึงอาร์เรย์ลิสต์ ลิงค์ลิสต์ และโครงสร้างข้อมูลอาร์เรย์อื่นๆ) โดยเกิดจากการเรียกใช้อาร์เรย์ในตำแหน่งที่ไม่ถูกต้องเช่น ประกาศตัวแปรอาร์เรย์ชนิด Integer โดยจองตำแหน่งในการเก็บข้อมูลเอาไว้ 5 ตำแหน่ง แต่เมื่อมีการเรียกใช้ตัวแปรอาร์เรย์นี้มีการเรียกใช้อาร์เรย์ในตำแหน่งที่ 5 จึงทำให้เกิดความผิดพลาดขึ้นดังตัวอย่างในรูปที่ 2.4 ซึ่งเกิดความผิดพลาดขึ้นในบรรทัดที่ 5

```
1  public class ArrayIndexEx {
2  
3      public static void main(String[] args) {
4          int[] var = new int[5];
5          System.out.println(var[5]);
6      }
7  }
```

รูปที่ 2.4 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด

2.1.3 ความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ (Arithmetic : Divided by zero exception)

คือ ความผิดพลาดที่เกิดจากการหารตัวเลขโดยตัวหารมีค่าเป็น 0 ซึ่งไม่สามารถทำได้

```
1  public class DivByZeroEx {
2      public static int var = 5;
3      public static int num = 0;
4  
5      public static void main(String[] args) {
6          System.out.println(var / num);
7      }
8  }
```

รูปที่ 2.5 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์

2.1.4 ความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง (Number format exception)

คือ ความผิดพลาดที่เกิดขึ้นจากการแปลงวัตถุที่มีค่าไม่ใช่ตัวเลขให้อยู่ในรูปแบบตัวเลข ตัวอย่างเช่นในรูปที่ 2.6 ที่มีการประกาศตัวแปรแบบสตริงที่มีค่า 125w จากนั้นมีการแปลงให้อยู่ในรูปแบบของคลาส Integer (แลงเป็นตัวเลข) โดยใช้คำสั่ง valueOf ซึ่งไม่สามารถทำได้จึงเกิดความผิดพลาดขึ้นในบรรทัดที่ 5

```
1  public class StrFormatEx {
2      public static String var = "125w";
3
4      public static void main(String[] args) {
5          int x = Integer.valueOf(var);
6      }
7  }
```

รูปที่ 2.6 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง

2.1.5 ความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด (String index out of range)

คือ ความผิดพลาดที่เกิดกับข้อมูลชนิดข้อความ (String) ซึ่งในภาษาจาวาข้อมูลชนิดนี้สามารถเข้าถึงในลักษณะของอาร์เรย์ของตัวอักษร (Character) โดยการระบุตำแหน่งของตัวอักษรที่ต้องการลงไปเพื่อเลือกเอาเฉพาะตัวอักษรที่ต้องการ ดังนั้นหากมีการใส่ตัวเลขตำแหน่งของตัวอักษรที่เกินขนาดของอาร์เรย์ตัวอักษร ก็จะก่อให้เกิดความผิดพลาดประเภทนี้ขึ้นตัวอย่างเช่น ในรูปที่ 2.7 มีการประกาศตัวแปรที่มีค่าเป็น SUT โดยมีความยาวเท่ากับ 3 ตัวอักษร ซึ่งสามารถเข้าถึงในลักษณะอาร์เรย์ได้จากตำแหน่งที่ 0 ถึง 2 เท่านั้นแต่ในบรรทัดที่ 5 มีการเข้าถึงในตำแหน่งที่ 3 จึงทำให้เกิดความผิดพลาดขึ้น

```
1  public class StrIndexEx {
2
3      public static void main(String[] args) {
4          String str = "SUT";
5          System.out.println(str.charAt(3));
6      }
7  }
```

รูปที่ 2.7 ตัวอย่างของซอสโค้ดที่เกิดความผิดพลาดประเภทการใช้งานสตริงเกินขอบเขตที่กำหนด

มหาวิทยาลัยเทคโนโลยีสุรนารี

2.2 ไบต์โค้ดของภาษาจาวา (Java Bytecode)

ข้อมูลเกี่ยวกับไบต์โค้ดที่นำเสนอในหัวข้อนี้จะเป็นข้อมูลของไบต์โค้ดที่ถูกคอมไพล์ด้วย Java 2 SDK Standard Edition v1.2.1 ในอนาคตอาจมีตัวคอมไพล์ที่ใหม่กว่าซึ่งอาจมีบางคำสั่งที่แตกต่างออกไป แต่ก็ยังคงมีลักษณะการทำงานที่คล้ายกันอยู่โดยในที่นี้จะเน้นไปที่การศึกษาเกี่ยวกับวิธีการและหลักการในการประมวลผลคำสั่งของไบต์โค้ดเป็นหลักสำคัญ

ไบต์โค้ด คือ ภาษาสื่อกลางของจาวากับเครื่องคอมพิวเตอร์ เช่นเดียวกับภาษาแอสเซมบลีของภาษาซี และซีพลัสพลัส (C and C++) โดยจาวาไบต์โค้ดเป็นชุดคำสั่งที่ได้จากการคอมไพล์ซอสโค้ดของภาษาจาวา นั่นก็คือโปรแกรมที่ได้จากการคอมไพล์ซอสโค้ดภาษาจาวานั่นเอง ซึ่งจำนวนคำสั่งของไบต์โค้ดจะส่งผลโดยตรงต่อขนาดของโปรแกรม หน่วยความจำที่ใช้ และความเร็วในการประมวลผล ดังนั้นถ้าหากมีโปรแกรมที่มีไบต์โค้ดจำนวนมากเป็นส่วนประกอบก็จะทำให้ต้องใช้หน่วยความจำจำนวนมากขึ้นในการประมวลผล และยังส่งผลให้ความเร็วของโปรแกรมลดลงอีกด้วย (Lievens, W., 2006) ในภาษาจาวาเมื่อทำการเขียนซอสโค้ดแล้วจะถูกบันทึกไว้ในไฟล์ตระกูล .java และต่อมาจะถูกคอมไพล์ให้อยู่ในรูปของไฟล์ตระกูล .class ซึ่งสามารถสร้างไบต์โค้ดจากไฟล์ที่ทำการคอมไพล์แล้วของภาษาจาวา (*.class) ได้ด้วยหลายวิธี ตัวอย่างเช่น การใช้คำสั่งดังรูปที่ 2.8

```
javac Employee.java

javap -c Employee > Employee.bc
```

รูปที่ 2.8 การสร้างไบต์โค้ดของภาษาจาวาผ่าน Command line

จากรูปที่ 2.8 เป็นการใช้คำสั่ง javap ซึ่งเป็นคำสั่งพื้นฐานในการจัดการเกี่ยวกับไบต์โค้ดโดยคำสั่งนี้ยังมีตัวเลือกอื่นๆที่จำเป็นในการสร้างไบต์โค้ดซึ่งได้อธิบายเอาไว้ในเว็บไซต์ของ Oracle Inc. (2004) ดังต่อไปนี้

-l	แสดงหมายเลขบรรทัดและตารางตัวแปรภายใน
-public	แสดงเฉพาะคลาสและตัวแปรที่มีลักษณะเป็นแบบ public
-protected	แสดงเฉพาะคลาสและตัวแปรที่มีลักษณะเป็นแบบ protected
-private	แสดงเฉพาะคลาสและตัวแปรที่มีลักษณะเป็นแบบ private
-s	แสดง type signature
-c	แสดงไบต์โค้ดของโปรแกรม

ในการสร้างวิธีการตรวจหาความผิดพลาดในไบต์โค้ดของภาษาจาวาด้วยวิธีการเปรียบเทียบรูปแบบ (Pattern matching) นั้นต้องอาศัยลักษณะของไบต์โค้ดที่ได้จากการคอมไพล์ซอสโค้ด ซึ่งหนึ่งชุดคำสั่งในซอสโค้ดอาจสามารถแยกย่อยได้เป็นอีกหลายคำสั่งของไบต์โค้ด ดังนั้นจึงจำเป็นที่จะแบ่งไบต์โค้ดออกเป็นชุดๆเพื่อความสะดวกในการเปรียบเทียบโดยใช้วิธีการเปรียบเทียบรูปแบบซึ่งในที่นี้จะอาศัยวิธีการแบ่งชุดคำสั่งไบต์โค้ดจากงานวิจัยของ Lance, D., R. H. Untch, et al. (1999) เข้ามาช่วยในการแบ่งชุดคำสั่ง จึงทำให้ต้องอาศัยผลลัพธ์ที่ได้จากคำสั่งใน javap ตัวเลือกอื่นๆเพื่อใช้ในการจัดรูปแบบไบต์โค้ดให้เหมาะสม

จากรูปที่ 2.8 เป็นการสั่งให้ตัวคอมไพล์ของภาษาจาวา (Java compiler) สร้างไฟล์ไบต์โค้ดขึ้นมาจากซอสโค้ดที่ชื่อ Employee.java ซึ่งผลลัพธ์ที่ได้คือไฟล์ที่บรรจุไบต์โค้ดในชื่อ Employee.bc โดยในตอนนี้ไบต์โค้ดที่ได้สามรถอ่านทำความเข้าใจ และสามารถเข้าใจการทำงานคร่าวๆของโปรแกรมจนอาจนำมาเขียนเป็นผังงาน (Flowchart) ดังที่กล่าวเอาไว้งานวิจัยของ Zhao, G., H. Chen, et al. (2008) แต่ไบต์โค้ดที่ได้ยังไม่สามารถแบ่งออก เป็นชุดคำสั่งย่อยได้เพียงแค่มีการแบ่งชื่อคลาสและเมท็อดเท่านั้น ดังตัวอย่างที่แสดงในรูปที่ 2.9

มหาวิทยาลัยเทคโนโลยีสุรนารี

```
Compiled from Employee.java

class Employee extends java.lang.Object {

public Employee(java.lang.String,int);

public java.lang.String employeeName();

public int employeeNumber();

}

Method Employee(java.lang.String,int)
0 aload_0
1 invokespecial #3 <Method java.lang.Object()>
4 aload_0
5 aload_1
6 putfield #5 <Field java.lang.String name>
9 aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 <Method void storeData(java.lang.String,
int)>
20 return

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn

Method int employeeNumber()
0 aload_0
1 getfield #4 <Field int idNumber>
4 ireturn
```

าวิทยาลัยเทคโนโลยีสุรน

รูปที่ 2.9 ตัวอย่างไบต์โค้ดที่ได้จากการใช้คำสั่ง javap

จากรูปที่ 2.9 เป็นตัวอย่างของไบต์โค้ดอย่างง่าย ประกอบไปด้วยสอง instance หนึ่ง constructor และ สามเมท็อด โดยห้าบรรทัดแรกคือรายละเอียดของไฟล์ซอสโค้ดจาวาที่ใช้สร้างไบต์โค้ด บรรทัดแรกคือชื่อไฟล์ซอสโค้ดที่ใช้ บรรทัดที่สองคือชื่อคลาสซึ่งโดยปรกติแล้วคลาสในภาษาจาวาจะสืบทอดจากคลาส java.lang.Object ส่วนบรรทัดที่เหลืออีกสามบรรทัดจะเป็น constructor และรายชื่อเมท็อดที่มีอยู่ โดยในบรรทัดอื่นๆถัดลงมาจะเป็นไบต์โค้ดที่ทำหน้าที่ประมวลผลในแต่ละเมท็อด ซึ่งจะเห็นว่ามีลักษณะคำสั่งที่สั้นและในหนึ่งบรรทัดจะมีเพียงหนึ่งคำสั่งเท่านั้น

ในคำสั่งปฏิบัติการ (Opcode) ของไบต์โค้ดเราจะสังเกตเห็นว่าอักษรนำหน้าตัวแรกของคำสั่งอาจต่างกันแต่คำสั่งที่ถัดจากนั้นจะเป็นคำสั่งที่เหมือนกัน เช่น aload_0 และ iload_2 จะสามารถแบ่งออกได้เป็นสามส่วนหลักๆคือ อักษรนำหน้า (Prefix) คำสั่ง และค่าคงที่ ดังที่จะแสดงในรูปที่ 2.10

Opcode	Prefix	คำสั่ง	ค่าคงที่
aload_0	a	load	0
iload_2	i	load	2

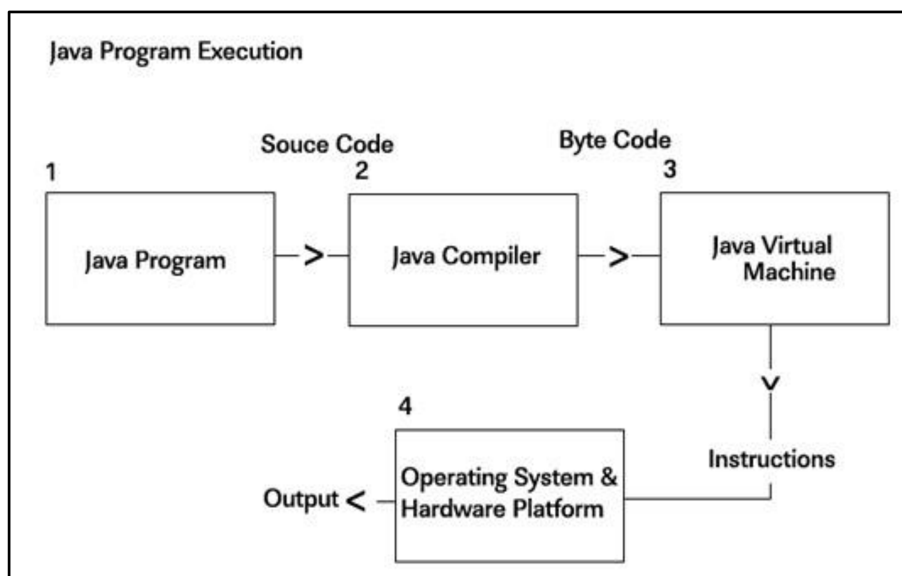
รูปที่ 2.10 ตัวอย่างการแยกคำสั่งปฏิบัติการ (Opcode) ของไบต์โค้ดภาษาจาวา

ในรูปที่ 2.10 อักษรนำหน้า a และ i บ่งบอกถึงประเภทของข้อมูลที่คำสั่งอ้างอิงถึง ในที่นี้คือ a หมายถึงอ้างอิงไปยังข้อมูล Object และ i หมายถึงอ้างอิงถึงข้อมูลประเภท Integer นอกจากนี้ยังมีอักษรนำหน้าอื่นๆอีกที่ได้อธิบายเอาไว้ในบทความของ Lievens, W. (2006) เช่น b คือการอ้างอิงถึงข้อมูลประเภท byte ตัวอักษร c คือการอ้างอิงถึงข้อมูลประเภท char และ d คือการอ้างอิงข้อมูลประเภท double เป็นต้น ในตัวอย่างนี้คำสั่งที่เรียกใช้คือ load หมายถึงการอ่านค่าที่เก็บเอาไว้ในแบบจำลองหน่วยความจำของจาวา โดยตำแหน่งที่ต้องการอ่านค่าจะถูกระบุถัดไปจากเครื่องหมาย _ ซึ่งในที่นี้คือตำแหน่งที่ 0 และ 2

ในไบต์โค้ดของภาษาจาวามีรูปแบบคำสั่งจำนวนมากในการประมวลผล แต่หลักการทำงานของการตีความคำสั่งจะมีลักษณะคล้ายกับตัวอย่างที่แสดงในรูปที่ 2.10 โดยอาจมีบางคำสั่งที่อาจทำงานต่างออกไปบ้างโดยในงานวิจัยนี้จะอ้างอิงคำสั่งจาก Java 2 SDK Standard Edition v1.2.1 เป็นหลัก ซึ่งหากมีการใช้จาวาแพลตฟอร์มที่แตกต่าง หรือในอนาคตมีการพัฒนาเวอร์ชันที่ใหม่กว่าอาจมีคำสั่งอื่นๆเพิ่มเติมและมีคำสั่งที่มีรูปแบบการตีความต่างออกไป

ในการพัฒนาซอฟต์แวร์ด้วยภาษาจาวา เมื่อจะนำออกไปใช้งานต้องได้รับการคอมไพล์โดยส่วนใหญ่จะอยู่ในรูปไฟล์นามสกุล .class ซึ่งบรรจุไบต์โค้ดที่มีลักษณะคล้ายกับภาษาเครื่องในการคอมไพล์โปรแกรมภาษาซีให้เป็นภาษาเครื่อง แต่เนื่องจากจาวามี complier ที่เป็นของภาษาจาวา

เอง และมีเวอร์ชวลแมชีน (Java Virtual Machine (JVM)) ที่ใช้เป็นสภาพแวดล้อมในการทำงานของภาษาเอง จึงทำให้ชุดคำสั่งที่บรรจุอยู่ในไฟล์นามสกุล .class แตกต่างจากภาษาเครื่อง ซึ่งในภาษาจาวามีหลักการคอมไพล์ดังรูปที่ 2.11



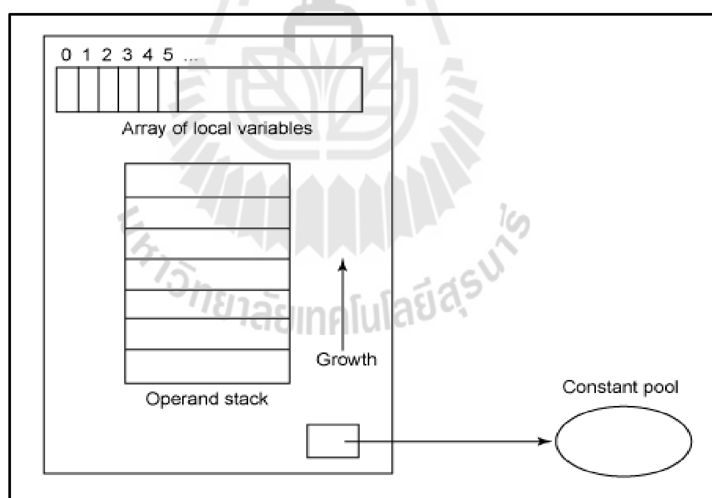
รูปที่ 2.11 การนำโปรแกรมภาษาจาวาไปใช้งาน (Gupta, R., 2006)

จากรูปที่ 2.11 เมื่อซอสโค้ดผ่านการคอมไพล์จากตัวคอมไพล์ของจาวา (Java compiler) เรียบร้อยแล้วผลลัพธ์ที่ได้คือไบต์โค้ด ซึ่งสามารถนำไปรันได้ในหลายระบบปฏิบัติการที่มีการติดตั้งเวอร์ชวลแมชีนเอาไว้ ซึ่งเครื่องมือเวอร์ชวลแมชีนจะทำหน้าที่เป็นตัวแปลภาษาในขั้นตอนการประมวลผลเพื่อแสดงผลลัพธ์ที่ได้จากการประมวลผลซอฟต์แวร์ (Gupta, R., 2006)

เนื่องจากภาษาจาวา และจาวาแพลตฟอร์มมีชื่อที่เหมือนกัน และมักจะพูดถึงพร้อมกันบ่อยๆทำให้คนทั่วไปสับสนว่า ภาษาจาวาและจาวาแพลตฟอร์มคือสิ่งเดียวกัน ที่จริงนั้นทั้งสองสิ่งแม้จะทำงานร่วมกันแต่ก็เป็นสิ่งที่แยกออกจากกัน โดยภาษาจาวาเป็นภาษาที่ใช้สำหรับเขียนโปรแกรม ส่วนจาวาแพลตฟอร์มคือสภาพแวดล้อมสำหรับการใช้งานโปรแกรมภาษาจาวา โดยมีองค์ประกอบหลักๆคือ จาวาเวอร์ชวลแมชีน (Java virtual machine) และ คลังโปรแกรมมาตรฐานจาวา (Java standard library)

“โปรแกรมที่ทำงานบนจาวาแพลตฟอร์มนั้น ไม่จำเป็นจะต้องสร้างด้วยภาษาจาวาเช่น อาจจะใช้ภาษาไพทอน (Python) หรือภาษาอื่นๆ ก็ได้ ส่วนภาษาจาวานั้นก็สามารถนำไปใช้พัฒนาโปรแกรมสำหรับแพลตฟอร์มอื่นได้เช่นเดียวกันเช่น คอมไพเลอร์ gcj สามารถคอมไพล์โปรแกรมที่เขียนด้วยภาษาจาวาให้ทำงานได้โดยไม่ต้องใช้จาวาเวอร์ชวลแมชีน” (http://th.wikipedia.org/wiki/ภาษาจาวา)

เพื่อความเข้าใจในรายละเอียดของไบต์โค้ด ในหัวข้อนี้จะอธิบายให้เห็นถึงการทำงานของจาวาเวอร์ชวลแมชีนว่าทำงานร่วมกับไบต์โค้ดอย่างไร ซึ่งในบทความของ Haggar, P. (2006) ได้อธิบายว่าจาวาเวอร์ชวลแมชีนทำงานโดยอาศัยหลักการของแสตก แต่ละเทรดที่ใช้ประมวลผลของจาวาเวอร์ชวลแมชีนจะจัดเก็บเฟรมคำสั่ง (Frame) โดยเฟรมจะถูกสร้างขึ้นทุกครั้งที่มีการเรียกเมท็อด เฟรมจะประกอบไปด้วยแสตกของคำสั่งดำเนินการ ตัวแปรในระดับท้องถิ่น และยังอ้างอิงไปยังแหล่งเก็บค่าคงตัว (Constant pool) ซึ่งเป็นคลาสของเมท็อดปัจจุบันดังในรูปที่ 2.12



รูปที่ 2.12 ลักษณะของแบบจำลองเฟรมของจาวาเวอร์ชวลแมชีน (Haggar, P., 2006)

อาร์เรย์ของตัวแปรระดับท้องถิ่นยังต้องเรียกใช้ตารางค่าตัวแปร (Local variable table) ที่เก็บค่าพารามิเตอร์ของเมท็อดไว้ และตารางนี้ยังเก็บค่าของตัวแปรระดับท้องถิ่นอื่นๆอีกด้วย พารามิเตอร์ในตำแหน่งแรกจะถูกเก็บไว้ในตำแหน่งที่ 0 โดยถ้าเฟรมนี้เป็นเฟรมที่เก็บ constructor หรือ instance ตำแหน่งถัดไปในตำแหน่งที่ 1 จะเก็บค่าพารามิเตอร์ตัวแรก และในตำแหน่งถัดไป

ตามจำนวนของพารามิเตอร์ สำหรับเมท็อดแบบคงที่ (Static method) ค่าของพารามิเตอร์จะเริ่มถูกเก็บไว้ในตำแหน่งที่ 0 และถัดไปตามลำดับ

2.3 การเปรียบเทียบรูปแบบ (Pattern matching)

รูปแบบ (Pattern) คือ อนุกรมของตัวอักษรที่ใช้สำหรับค้นหาในข้อความ (String) ซึ่งในงานวิจัยนี้ใช้เป็นวิธีการหลักในการค้นหารูปแบบของไบต์โค้ดที่จะทำให้เกิดความผิดพลาด เพราะเนื่องจากคำสั่งของไบต์โค้ดมีรูปแบบที่คงที่และเป็นมาตรฐานจึงทำให้สามรถใช้เทคนิคการเปรียบเทียบรูปแบบในการเปรียบเทียบคำสั่งได้ ในหัวข้อนี้จะนำเสนอให้เห็นถึงการเปรียบเทียบรูปแบบในภาษา Perl ซึ่งภาษานี้รูปแบบที่จะใช้ค้นหาจะอยู่ภายในเครื่องหมายต่างๆ / เช่น /cat/ , /c*t/ และ /ca+t/ เป็นต้น

โดยในภาษา Perl จะใช้เครื่องหมาย = ~ เพื่อเปรียบเทียบว่าในข้อความ (String) มีรูปแบบที่กำหนดประกอบอยู่ด้วยหรือไม่ดังที่แสดงในรูปที่ 2.13

```
$result = $var = ~ /abc/;
```

รูปที่ 2.13 ตัวอย่างการเปรียบเทียบรูปแบบในภาษา Perl

จากรูปที่ 2.13 เป็นการค้นหาว่ามีรูปแบบ abc อยู่ภายในข้อความที่อยู่ในตัวแปร $var หรือไม่โดยผลลัพธ์ที่ได้จะถูกเก็บไว้ในตัวแปร $result ซึ่งค่าที่เป็นไปได้คือ จริงหรือเท็จเท่านั้น (ในภาษา Perl จะแทนด้วยตัวเลข 0 และ 1)

2.3.1 ตัวอักขระพิเศษในการเปรียบเทียบรูปแบบ (Schwartz, R., T. Christiansen, et al., 1997)

ในภาษา Perl ได้กำหนดอักขระพิเศษขึ้นเพื่อใช้แทนกรณีพิเศษต่างๆในการเปรียบเทียบรูปแบบ ซึ่งมีอักขระที่สำคัญที่นิยมใช้กัน ดังนี้

- อักขระ + หมายถึง มีอักขระที่อยู่ข้างหน้าเครื่องหมาย + อย่างน้อยหนึ่งตัวซึ่งถ้าหากใช้เดี่ยวๆไม่มีอักขระนำหน้าจะหมายถึง ไม่เป็นข้อความว่างๆ หรือ มีอักขระประกอบอยู่อย่างน้อยหนึ่งตัวนั่นเอง ดังเช่นตัวอย่างต่อไปนี้

ข้อความที่ตรงกับรูปแบบ /ca+t/ คือ

cat

caat

caaaaaaaat

เป็นต้น

รูปที่ 2.14 ตัวอย่างการใช้อักขระ + ในการเปรียบเทียบรูปแบบ

- อักขระ * หมายถึง ไม่ต้องมีอักขระที่นำหน้าเครื่องหมาย * หรือ มีอักขระที่นำหน้าเครื่องหมายเท่าใดก็ได้ไม่จำกัด เช่น /de*f/ จะมีรูปแบบที่ประกอบอยู่ในข้อความ df, def และ deef เป็นต้น
- เมื่อต้องการใช้อักขระพิเศษให้เป็นอักขระธรรมดาเพื่อนำไปเปรียบเทียบรูปแบบจะต้องใช้อักขระ \ นำหน้าอักขระดังกล่าว เช่น /*+/ หมายถึง ภายในข้อความต้องประกอบไปด้วย * อย่างน้อยหนึ่งตัว เป็นต้น
- อักขระ . หมายถึง เป็นอักขระใดๆก็ได้ยกเว้นอักขระในการขึ้นบรรทัดใหม่ เช่น /d.f/ มีรูปแบบตรงกับข้อความ ddf, def และ dduf เป็นต้น
- อักขระ ? หมายถึง เป็นอักขระใดๆก็ได้หนึ่งตัว เช่น /b?r/ เป็นรูปแบบที่ประกอบอยู่ในข้อความ bar, ber และ bir เป็นต้น

อักขระ [อนุกรมอักขระ] หมายถึง จะต้องเป็นอักขระที่มีอยู่ในอนุกรมอักขระที่กำหนดอยู่ภายในเครื่องหมาย [ ] เท่านั้น เช่น d[0123456789]f จะมีรูปแบบที่อยู่ภายในข้อความ d2f, d5f และ d0f เป็นต้น

มหาวิทยาลัยเทคโนโลยีสุรนารี

2.3.2 Anchoring pattern (Schwartz, R., T. Christiansen, et al., 1997)

- อักขระ ^ และ $ ในการเปรียบเทียบรูปแบบโดยเครื่องหมาย ^ จะใช้เพื่อทำให้แน่ใจว่าอักขระที่กำหนดอยู่หน้าสุดในข้อความที่นำมาเปรียบเทียบ และอักขระ $ จะใช้เพื่อให้แน่ใจว่าอักขระที่กำหนดอยู่เป็นอักขระที่อยู่ท้ายสุดของข้อความ เช่น /^def/ หมายถึง จะต้องเป็นข้อความที่มีอักขระสามตัวแรกขึ้นต้นด้วย def และ /def$/ หมายถึง จะต้องเป็นข้อความที่ลงท้ายด้วยอักขระสามตัวคือ def เป็นต้น
- เมื่ออักขระ ^ ปรากฏเป็นอักขระตัวแรกหลังอักขระ [([^อนุกรมอักขระ]) หมายถึง เป็นอักขระใดๆก็ได้ยกเว้นอักขระที่ประกอบอยู่ในอนุกรมอักขระที่กำหนด เช่น /d[^eE]f/ จะมีรูปแบบที่ประกอบอยู่ในข้อความ dcf, dgf และ dvf เป็นต้น
- อักขระ | จะใช้คั่นกลางระหว่างอนุกรมอักขระ (/อนุกรมอักขระ1|อนุกรมอักขระ2/) เพื่อเป็นการแทนว่าข้อความที่นำมาเปรียบเทียบรูปแบบนั้นจะต้องประกอบไปด้วยอนุกรมอักขระ1 หรือประกอบไปด้วยอนุกรมอักขระ2อย่างใดอย่างหนึ่ง เช่น /def|ghi/ หมายถึง ข้อความที่จะนำมาเปรียบเทียบรูปแบบจะต้องเป็นข้อความที่ประกอบไปด้วยอนุกรมอักขระ def หรือ ghi อย่างใดอย่างหนึ่งเท่านั้น
- อักขระ \b และอักขระ \B สองอักขระนี้จะใช้ในความหมายที่ต่างกัน โดยถ้าเป็นอักขระ \b หมายถึง ข้อความที่นำมาเปรียบเทียบจะต้องขึ้นต้นด้วยอนุกรมข้อความที่กำหนดหรือลงท้ายด้วยอนุกรมข้อความที่กำหนดเช่น /\bdef/ หมายถึง ข้อความที่นำมาเปรียบเทียบต้องขึ้นต้น หรือ ลงท้ายด้วย def ในทางตรงกันข้ามอักขระ \B หมายถึง ข้อความที่นำมาเปรียบเทียบรูปแบบจะต้องไม่ขึ้นต้นหรือลงท้ายด้วยอนุกรมอักขระที่กำหนด

2.3.3 อักขระที่กำหนดไว้ล่วงหน้าในการเปรียบเทียบรูปแบบ (Humbad, S. N., 2004)

- \d หมายถึง เป็นตัวเลขจำนวนเต็มใดๆก็ได้ [0-9]
- \D หมายถึง เป็นอักขระใดๆก็ได้ที่ไม่ใช่ตัวเลขจำนวนเต็ม [^0-9]
- \w หมายถึง เป็นอักขระปรกติใดๆ [_0-9a-zA-Z]
- \W หมายถึง เป็นอักขระใดๆที่ไม่ใช่อักขระปรกติ [^_0-9a-zA-Z]
- \s หมายถึง เป็นอักขระที่เป็นเครื่องหมายวรรคตอน [\r\t\n\f]

- \S หมายถึง เป็นอักขระที่ไม่เป็นเครื่องหมายวรรคตอน [^\r\t\n\f]
- (อักขระพิเศษ) ใช้เพื่อจดจำสิ่งที่ทำการเปรียบเทียบ ซึ่งสามารถอ้างอิงได้โดยใช้ \ หมายเลข ได้ตามลำดับ เช่น /fred(.)barney\1/ จะมีรูปแบบอยู่ในข้อความ fredxbarneyx แต่ไม่อยู่ในข้อความ fredxbarneyy เป็นต้น
- ในการเปรียบเทียบจำนวนอักขระที่ปรากฏจะใช้อักขระพิเศษ { } ซึ่งระหว่างอักขระนี้จะเป็นตัวเลขที่บอกจำนวนของอักขระที่จะปรากฏ เช่น /de{1,3}f/ จะมีรูปแบบอยู่ในข้อความ def, deef และ deeef ส่วน /de{3}f/ จะมีรูปแบบอยู่ในข้อความ deeef เท่านั้น และ /de{3,}f/ หมายถึงจะต้องประกอบด้วยอักขระ e ตั้งแต่สามตัวขึ้นไป เป็นต้น

2.3.4 ลำดับความสำคัญในการเปรียบเทียบรูปแบบ

- () การจดจำสิ่งที่เปรียบเทียบ
- + * ? +? *? {n,m} จำนวนอักขระที่ปรากฏ
- ^ $ \b \B Pattern anchors
- |

มหาวิทยาลัยเทคโนโลยีสุรนารี

2.4 ตารางคำสั่งปฏิบัติการของไบต์โค้ด (Java Bytecode instruction table)

หากต้องการที่จะสร้างวิธีการตรวจสอบความผิดพลาดจากไบต์โค้ดแล้ว จำเป็นที่จะต้องเข้าใจรายละเอียดของคำสั่งว่าแต่ละคำสั่งหมายถึงอะไรและทำงานอย่างไร ซึ่งหากเข้าใจดีแล้วจะทำให้สามารถที่จะเข้าใจถึงรูปแบบของคำสั่งที่ทำให้เกิดความผิดพลาดและสามารถนำมาสร้างเป็นวิธีการในการค้นหาได้ในที่สุด โดยในการนำเสนอคำสั่งของไบต์โค้ดในหัวข้อนี้จะอ้างอิงคำสั่งจากจาวาเวอร์ชวลแมชีนเวอร์ชัน 1.6 เท่านั้น ซึ่งรายละเอียดของคำสั่งจะแสดงตามลำดับตัวอักษรได้ดังตารางในภาคผนวก ก. (Harrison, T., 2006)

2.5 ฐานข้อมูล H2 (H2 Database)

ในการวิจัยครั้งนี้จะต้องทำการจัดแบ่งชุดคำสั่งเพื่อเตรียมสำหรับการเปรียบเทียบรูปแบบ เมื่อพิจารณาแล้วผู้วิจับพบว่าการจัดเก็บชุดคำสั่งในรูปแบบของข้อมูลในฐานข้อมูลเหมาะสมที่สุด เนื่องจากลักษณะข้อมูลจะถูกจัดแบ่งออกเป็นระเบียนข้อมูล (Records) ซึ่งจะเหมาะกับการแยกแต่ละชุดคำสั่งออกเป็นแต่ละระเบียนข้อมูล และนอกจากนี้ยังสามารถใช้ภาษาสอบถามข้อมูล (Query) ที่มีความสารถในการเปรียบเทียบรูปแบบเช่นคำสั่ง Like ในการเปรียบเทียบอีกด้วย เมื่อพิจารณาฐานข้อมูลที่เหมาะสมกับงานวิจัย ผู้วิจัยได้เลือกฐานข้อมูล H2 มาใช้ในการวิจัยเนื่องจากฐานข้อมูลนี้มีคุณสมบัติตรงตามที่ผู้วิจัยกล่าวมา และยังสามารถทำงานในโหมดฝังกับโปรแกรม (Embedded mode) อีกทั้งยังเป็นฐานข้อมูลที่เปิดเผยซอสโค้ดอีกด้วย

รูปที่ 2.15 สัญลักษณ์ของฐานข้อมูล H2 (Muller, T., 2006)

Muller, T. (2006) ผู้ที่พัฒนาฐานข้อมูล H2 ได้อธิบายเอาไว้ว่าฐานข้อมูล H2 เป็นระบบจัดการฐานข้อมูลแบบความสัมพันธ์ (Relational database) ถูกพัฒนาขึ้นด้วยภาษาจาวา สามารถทำงานได้ทั้งแบบฝังไปกับโปรแกรม (Embedded) และแบบลูกข่าย – แม่ข่าย (Client - Server) ซึ่งมีขนาดประมาณ 1 เมกะไบต์

ฐานข้อมูล H2 มีลักษณะเป็นโอเพ่นซอร์สถูกเผยแพร่ครั้งแรกเมื่อปี 2004 แต่ เป็นที่รู้จักมากในปี 2005 โดยผู้ริเริ่มพัฒนาคือ Thomas Muller ต่อมามีการปรับปรุงแก้ไข และ ออกเวอร์ชันใหม่มาเรื่อยๆ ภายใต้การกำกับดูแลของ Mozilla ในชื่อย่อโครงการ MPL ฐานข้อมูล H2 สนับสนุนมาตรฐานภาษา SQL เป็นหลัก ซึ่งภาษา SQL เป็นที่นิยมใช้กันในระบบจัดการฐานข้อมูลต่างๆ นอกจากนี้ฐานข้อมูล H2 ยังมีส่วนเชื่อมต่อซอฟต์แวร์ที่รองรับมาตรฐานหลายตัวเช่น JDBC PostgreSQL และ MySQL เป็นต้น

2.5.1 คุณสมบัติเด่นของฐานข้อมูล H2

- เป็นฐานข้อมูลที่มีความเร็วสูงในปัจจุบัน (พ.ศ.2554)
- มีลักษณะเป็นโอเพ่นซอร์ส
- เขียนด้วยภาษาจาวาทั้งหมด
- สนับสนุนมาตรฐาน SQL และ JDBC
- ทำงานได้ทั้งแบบฝังกับโปรแกรม และ ลูกข่าย – แม่ข่าย
- รองรับการประมวลผลแบบคู่ขนาน
- มีความปลอดภัยสูง
- รองรับการเชื่อมต่อหลายๆการเชื่อมต่อพร้อมกัน
- มีขนาดเล็ก (ประมาณ 1 เมกะไบต์)
- รองรับการทำงานทั้งแบบระบบไฟล์ และบนหน่วยความจำ (Memory)

2.5.2 คุณสมบัติของฐานข้อมูล H2 เมื่อเทียบกับกับฐานข้อมูลอื่นๆ

การเปรียบเทียบคุณสมบัติในหัวข้อนี้จะใช้ H2 1.3, Apache Derby 10.8, HSQLDB 2.2, MySQL 5.5, PostgreSQL 9.0 เป็นฐานข้อมูลที่ใช้เปรียบเทียบ

ตารางที่ 2.1 การเปรียบเทียบคุณสมบัติของฐานข้อมูล H2 กับฐานข้อมูลอื่น (Muller, T., 2006)

Feature	H2	Derby	HSQLDB	MySQL	PostgreSQL
Pure Java	Yes	Yes	Yes	No	No
Embedded Mode (Java)	Yes	Yes	Yes	No	No
In-Memory Mode	Yes	Yes	Yes	No	No
Explain Plan	Yes	Yes	Yes	Yes	Yes
Built-in Clustering / Replication	Yes	Yes	No	Yes	Yes
Encrypted Database	Yes	Yes *10	Yes *10	No	No

ตารางที่ 2.1 การเปรียบเทียบคุณสมบัติของฐานข้อมูล H2 กับฐานข้อมูลอื่น (Muller, T., 2006) (ต่อ)

Feature	H2	Derby	HSQLDB	MySQL	PostgreSQL
Linked Tables	Yes	No	Partially *1	Partially *2	No
ODBC Driver	Yes	No	No	Yes	Yes
Fulltext Search	Yes	No	No	Yes	Yes
Files per Database	Few	Many	Few	Many	Many
Row Level Locking	Yes *9	Yes	Yes *9	Yes	Yes
Multi Version Concurrency	Yes	No	Yes	Yes	Yes
Multi-Threaded Processing	No *11	Yes	Yes	Yes	Yes
Role Based Security	Yes	Yes *3	Yes	Yes	Yes
Updatable Result Sets	Yes	Yes *7	Yes	Yes	Yes
Sequences	Yes	Yes	Yes	No	Yes
Temporary Tables	Yes	Yes *4	Yes	Yes	Yes
Information Schema	Yes	No *8	Yes	Yes	Yes
Case Insensitive Columns	Yes	Yes	Yes	Yes	Yes *6
Custom Aggregate Functions	Yes	No	Yes	Yes	Yes
CLOB/BLOB Compression	Yes	No	No	No	No
Footprint (jar/dll size)	~1 MB *5	~2 MB	~1 MB	~4 MB	~6 MB

*1 HSQLDB สนับสนุนเฉพาะตารางแบบข้อความ

*2 MySQL สนับสนุนการเชื่อมโยงตารางภายใต้ชื่อตาราง 'federated tables'.

*3 Derby จะมีการสนับสนุนด้านความปลอดภัย และการตรวจสอบรหัสผ่านจากการตั้งค่าเท่านั้น

*4 Derby จะสนับสนุนเฉพาะตารางแบบชั่วคราวที่เป็นโกบอลเท่านั้น

*5 ในไฟล์นามสกุล .jar ของฐานข้อมูล H2 จะเก็บข้อมูลการดีบักไว้ด้วย แต่ฐานข้อมูลตัวอื่นจะไม่มีการจัดเก็บ

*6 PostgreSQL สนับสนุนการชี้ตำแหน่งแบบฟังก์ชันเท่านั้น

*7 Derby จะสามารถปรับปรุงเซตของผลลัพธ์ได้ถ้าไม่มีการจัดเรียงการสอบถามข้อมูล (Query) เท่านั้น

*8 Derby ไม่สนับสนุนมาตรฐานนี้

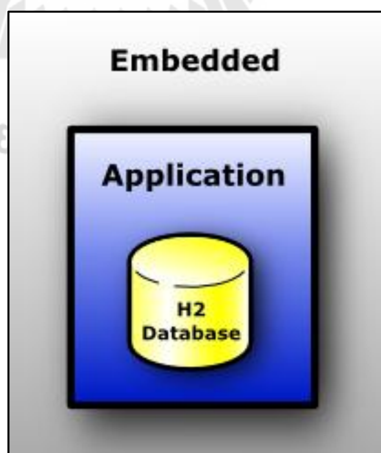
*9 เมื่อมีการใช้ MVCC (Multi version concurrency) เท่านั้น

*10 Derby และ HSQLDB ไม่มีการซ่อนรูปแบบข้อมูลที่ดี

*11 การประมวลผลแบบหลายเทรดไม่ถูกใช้ในการตั้งค่าแบบปรกติ และไม่สนับสนุนเมื่อใช้ MVCC.

2.5.3 รูปแบบการเชื่อมต่อของฐานข้อมูล H2

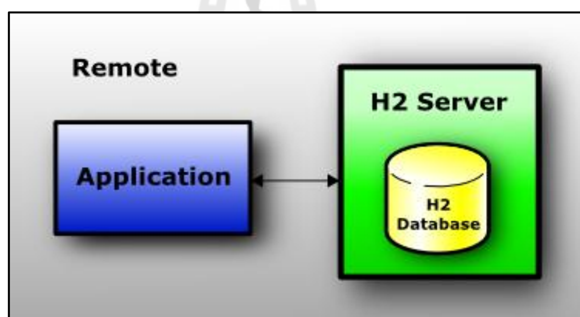
โหมดฝังกับโปรแกรม (Embedded Mode) ในโหมดนี้ฐานข้อมูลจะถูกเชื่อมต่อโดย JDBC ซึ่งจะมีความง่ายและรวดเร็วเป็นอย่างมาก อีกทั้งยังสามารถเชื่อมต่อฐานข้อมูลหลายๆฐานข้อมูลพร้อมๆกันหรือเปิดการเชื่อมต่อหลายๆการเชื่อมต่อในหนึ่งฐานข้อมูลก็สามารถทำได้อย่างรวดเร็ว แต่การทำงานในโหมดนี้ฐานข้อมูลจะถูกจำกัด (Lock) ให้สามารถเชื่อมต่อได้โดยหนึ่งซอฟต์แวร์เท่านั้น หากซอฟต์แวร์อื่นจะทำการเชื่อมต่อจะต้องรอให้ซอฟต์แวร์แรกจบการเชื่อมต่อก่อนเป็นลำดับต่อไปเรื่อยๆ



รูปที่ 2.16 การทำงานของฐานข้อมูล H2 ในโหมดฝังกับโปรแกรม (Muller, T., 2006)

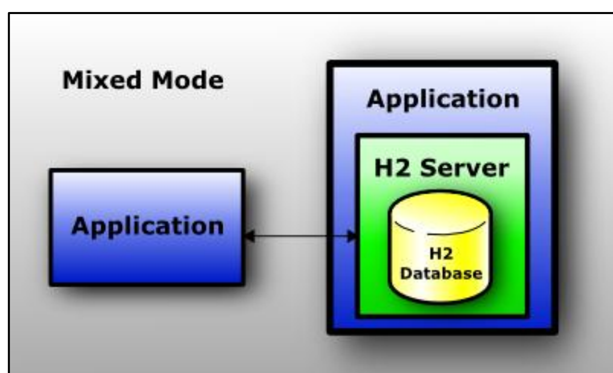
โหมดแม่ข่าย (Server mode) เมื่อใช้รูปแบบการเชื่อมต่อในโหมดแม่ข่ายซอฟต์แวร์จะเข้าถึงฐานข้อมูลจากระยะไกลด้วยส่วนเชื่อมต่อ JDBC หรือ ODBC ซึ่งแม่ข่ายจะทำงานบนสภาวะแวดล้อมแยกกับเครื่องลูกข่ายหรืออาจคนละเครื่องคอมพิวเตอร์ การเชื่อมต่อหลายการเชื่อมต่อจากหลายซอฟต์แวร์จากหลายๆสถานที่สามารถทำได้พร้อมกันในเวลาเดียว ซึ่งการทำงานภายในแม่ข่ายจะใช้วิธีการเดียวกับการเชื่อมต่อโหมดแนบไปกับซอฟต์แวร์ (Embedded mode) ภายใต้การจัดการของตัวจัดการแม่ข่าย H2

การเชื่อมต่อในโหมดนี้จะมีความเร็วที่ต่ำกว่าโหมดฝังไปกับซอฟต์แวร์ (Embedded mode) เนื่องจากข้อมูลจะต้องถูกส่งผ่านโปรโตคอล TCP/IP แต่จะมีข้อดีในแง่ของการเชื่อมต่อพร้อมๆกันจากหลายๆแหล่งพร้อมกันหรือหลายๆการเชื่อมต่อพร้อมๆกันก็สามารถทำได้



รูปที่ 2.17 การทำงานของฐานข้อมูล H2 ในโหมดแม่ข่าย (Muller, T., 2006)

โหมดผสม (Mixed mode) โหมดนี้เป็นการผสมผสานระหว่างโหมดฝังไปกับซอฟต์แวร์และโหมดแม่ข่าย โดยจะมีซอฟต์แวร์แรกเชื่อมต่อฐานข้อมูลในโหมดฝังไปกับซอฟต์แวร์พร้อมกับเริ่มการทำงานฐานข้อมูลในโหมดแม่ข่าย (ซอฟต์แวร์ และฐานข้อมูลจะถูกประมวลผลคนละสภาวะแวดล้อมหรือคนละโปรเซส) ซึ่งสามารถเข้าถึงฐานข้อมูลได้ทั้งแบบภายใน (Local) และจากภายนอก แต่หากมีการเข้าถึงพร้อมๆกัน ในขณะที่มีการเข้าถึงจากภายใน (มีการเชื่อมต่อในโหมดฝังไปกับซอฟต์แวร์อยู่ก่อนแล้ว) จะทำให้ความเร็วในการเข้าถึงข้อมูลลดลงเล็กน้อย



รูปที่ 2.18 การทำงานของฐานข้อมูล H2 ในโหมดผสม (Muller, T., 2006)

2.5.4 การกำหนด URL เพื่อกำหนดโหมดการเชื่อมต่อ

ฐานข้อมูล H2 สนับสนุนการเชื่อมต่ออย่างหลากหลาย ซึ่งการเชื่อมต่อจะถูกกำหนดด้วย URL ที่แตกต่างกันดังต่อไปนี้

ตารางที่ 2.2 URL ที่ใช้ในการกำหนดการเชื่อมต่อฐานข้อมูล H2 (Muller, T., 2006)

รูปแบบการตั้งค่า	รูปแบบ URL และตัวอย่าง
โหมดแนบไปกับซอฟต์แวร์ (Embedded)	jdbc:h2:[file:][<path>]<databaseName> jdbc:h2:~/test jdbc:h2:file:/data/sample jdbc:h2:file:C:/data/sample (Windows only)
โหมดบนหน่วยความจำ (Private)	jdbc:h2:mem:
โหมดบนหน่วยความจำ (Named)	dbc:h2:mem:<databaseName> jdbc:h2:mem:test_mem
โหมดแม่ข่ายผ่าน TCP/IP	jdbc:h2:tcp://<server>[:<port>]/[<path>]<databaseName> jdbc:h2:tcp://localhost/~/test jdbc:h2:tcp://dbserv:8084/~/sample jdbc:h2:tcp://localhost/mem:test
โหมดแม่ข่ายผ่าน SSL/TLS	jdbc:h2:ssl://<server>[:<port>]/<databaseName> jdbc:h2:ssl://localhost:8085/~/sample;

มหาวิทยาลัยเทคโนโลยีสุรนารี

ตารางที่ 2.2 URL ที่ใช้ในการกำหนดการเชื่อมต่อฐานข้อมูล H2 (Muller, T., 2006) (ต่อ)

รูปแบบการตั้งค่า	**รูปแบบ URL และตัวอย่าง**
ใช้การเข้ารหัสไฟล์	jdbc:h2:<url>;CIPHER=[AES|XTEA] jdbc:h2:ssl://localhost/~/test;CIPHER=AES
กำหนดเมท็อดการสงวนไฟล์	jdbc:h2:<url>;FILE_LOCK={FILE|SOCKET|NO} jdbc:h2:file:~/private;CIPHER=XTEA;FILE_LOCK=SOCKET
กำหนดให้ใช้ได้เมื่อฐานข้อมูลถูกพบเท่านั้น	jdbc:h2:<url>;IFEXISTS=TRUE jdbc:h2:file:~/sample;IFEXISTS=TRUE
กำหนดให้ไม่มีการปิดฐานข้อมูลเมื่อ VM จบการทำงาน	jdbc:h2:<url>;DB_CLOSE_ON_EXIT=FALSE
กำหนดชื่อผู้ใช้และรหัสผ่าน	jdbc:h2:<url>[;USER=<username>][;PASSWORD=<value>] jdbc:h2:file:~/sample;USER=sa;PASSWORD=123
กำหนดระดับการดีบัก	jdbc:h2:<url>;TRACE_LEVEL_FILE=<level 0..3> jdbc:h2:file:~/sample;TRACE_LEVEL_FILE=3
ให้ยอมรับการตั้งค่าที่ไม่รู้จัก	jdbc:h2:<url>;IGNORE_UNKNOWN_SETTINGS=TRUE
กำหนดโหมดการเข้าถึงไฟล์	jdbc:h2:<url>;ACCESS_MODE_DATA=rws
ฐานข้อมูลในไฟล์ตระกูล zip	jdbc:h2:zip:<zipFileName>!/<databaseName> jdbc:h2:zip:~/db.zip!/test
กำหนดโหมดความเข้ากันได้	jdbc:h2:<url>;MODE=<databaseType> jdbc:h2:~/test;MODE=MYSQL
เริ่มการเชื่อมต่อใหม่อัตโนมัติ	jdbc:h2:<url>;AUTO_RECONNECT=TRUE jdbc:h2:tcp://localhost/~/test;AUTO_RECONNECT=TRUE
โหมดผสมแบบอัตโนมัติ	jdbc:h2:<url>;AUTO_SERVER=TRUE jdbc:h2:~/test;AUTO_SERVER=TRUE
กำหนดขนาดเพจของฐานข้อมูล	jdbc:h2:<url>;PAGE_SIZE=512
เปลี่ยนการตั้งค่าอื่นๆ	jdbc:h2:<url>;<setting>=<value>[;<setting>=<value>...] jdbc:h2:file:~/sample;TRACE_LEVEL_SYSTEM_OUT=3

2.5.5 การเคลื่อนย้าย และเปลี่ยนชื่อฐานข้อมูล

ในฐานข้อมูล H2 ทำงานบนพื้นฐานของระบบไฟล์ซึ่งเป็นไฟล์ที่ไม่มีรูปแบบตายตัว ชื่อฐานข้อมูล และตำแหน่งของฐานข้อมูลไม่ได้ถูกจัดเก็บไว้ภายในไฟล์ฐานข้อมูลเมื่อไม่มีการเชื่อมต่อใดๆผู้ใช้สามารถย้ายตำแหน่งหรือเปลี่ยนชื่อไฟล์ฐานข้อมูลได้ ซึ่งการกำหนดตำแหน่งของไฟล์ฐานข้อมูลจะระบุด้วย URL ดังที่ได้นำเสนอไปในหัวข้อที่ผ่านมา ดังนั้นถึงแม้ว่าจะมีการย้ายตำแหน่งไฟล์ไปยังตำแหน่งต่างๆหรือในต่างระบบปฏิบัติการ ก็ไม่ก่อให้เกิดปัญหาในการทำงานแต่อย่างใด ซึ่งไฟล์ที่เกี่ยวข้องกับฐานข้อมูล H2 จะมีดังต่อไปนี้

ตารางที่ 2.3 ไฟล์ที่เกี่ยวข้องกับฐานข้อมูล H2 (Muller, T., 2006)

ชื่อไฟล์	คำอธิบาย	จำนวนไฟล์
test.h2.db	ไฟล์ฐานข้อมูลบรรจุข้อมูลตารางต่างๆ และข้อมูลทั้งหมดของฐานข้อมูล Format: <database>.h2.db	1 ไฟล์ต่อฐานข้อมูล
test.lock.db	ไฟล์สงวนฐานข้อมูลจะถูกสร้างขึ้นอัตโนมัติเมื่อมีการใช้ฐานข้อมูลและจะหายไปเมื่อไม่มีการเชื่อมต่อ Format: <database>.lock.db	1 ไฟล์ต่อฐานข้อมูล (ถ้าถูกใช้)
test.trace.db	ไฟล์ติดตามผล (จะมีถ้าตั้งค่าให้มีการติดตามฐานข้อมูล) บรรจุข้อมูลการประมวลผลของข้อมูล Format: <database>.trace.db ถูกเปลี่ยนชื่อเป็น <database>.trace.db.old เมื่อไฟล์มีขนาดใหญ่	1 ไฟล์ต่อฐานข้อมูล หรือไม่มี
test.lobs.db/*	จะมีไฟล์นี้เมื่อ BLOB หรือ CLOB ค่ามีขนาดใหญ่กว่าที่กำหนด. Format: <id>.<tableId>.lob.db	1 ไฟล์ต่อหนึ่งวัตถุขนาดใหญ่
test.123.temp.db	ไฟล์ชั่วคราวบรรจุผลลัทธ์ที่มีขนาดใหญ่ชั่วคราว Format: <database>.<id>.temp.db	1 ไฟล์ต่อหนึ่งวัตถุ

2.5.6 โหมดความเข้ากันได้กับฐานข้อมูลอื่นๆ

ในฐานข้อมูล H2 ยังมีโหมดที่ทำให้สามารถทำงานร่วมกับฐานข้อมูลต่างๆได้ ซึ่งในปัจจุบันมีความสามารถทำงานร่วมกับฐานข้อมูลต่างๆดังรายชื่อต่อไปนี้

- DB2
- Derby
- HSQLDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

2.5.7 การเชื่อมต่อฐานข้อมูล H2

ในหัวข้อนี้จะยกตัวอย่างการเชื่อมต่อฐานข้อมูล H2 ด้วย JDBC เนื่องจากเป็นวิธีที่สะดวกและรวดเร็ว ซึ่งจะเป็นวิธีที่ถูกนำไปใช้ในงานวิจัยครั้งนี้ โดยการเชื่อมต่อจะต้องกำหนดตัวแปรสามตัว คือ ชื่อผู้ใช้ รหัสผ่าน และ URL ที่ระบุโหมดการเชื่อมต่อ ซึ่งมีลักษณะดังตัวอย่างต่อไปนี้

```
Class.forName("org.h2.Driver");

String url = "jdbc:h2:~ ";

String user = "USER";

String pwds = "PASSWORD";

conn = DriverManager.getConnection(url, user, pwds);
```

รูปที่ 2.19 ตัวอย่างคำสั่งในการเชื่อมต่อฐานข้อมูล H2

2.6 โปรแกรม FindBugs

ในงานวิจัยนี้จะนำเอาโปรแกรม FindBugs เป็นเครื่องมืออ้างอิงในการทดสอบเครื่องมือตัวอย่างที่จะทำการสร้างขึ้น โดยใช้ FindBugs ค้นหาจำนวนและประเภทของความผิดพลาดที่เกิดขึ้นกับซอสโค้ดที่นำมาทดสอบ และใช้เครื่องมือตัวอย่างที่ทำการพัฒนาขึ้นค้นหาความผิดพลาดที่เกิดขึ้น โดยการใช้ไบต์โค้ดในการทดสอบแทนซอสโค้ด จากนั้นนำผลลัพธ์ที่ได้มาอ้างอิงกับจำนวนและประเภทของความผิดพลาดที่เกิดขึ้นจากผลลัพธ์ของการทดสอบด้วยโปรแกรม FindBugs ว่าครบถ้วนและถูกต้องหรือไม่

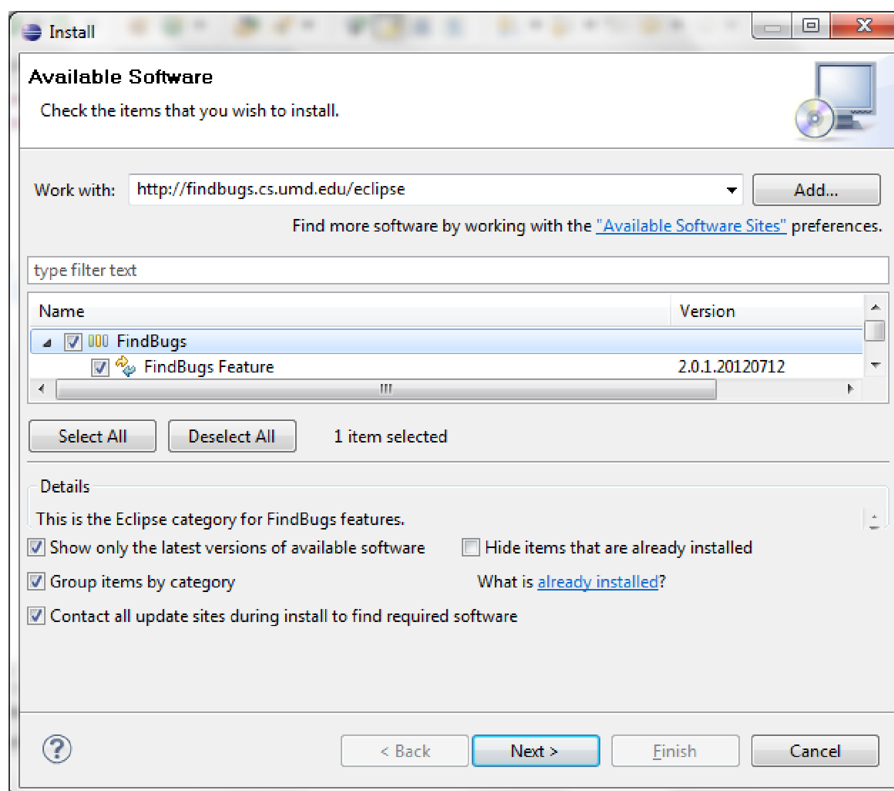
FindBugs (Hovemeyer, D., B. Pugh, et al., 2012) ใช้เทคนิคการวิเคราะห์โปรแกรมแบบคงที่ (Static analysis) เพื่อค้นหาจุดบกพร่องในโค้ดของภาษาจาวา FinfBugs เป็นโปรแกรมที่อนุญาตให้ใช้งานได้โดยไม่ต้องเสียค่าใช้จ่าย โปรแกรม FindBugs ถูกพัฒนาขึ้นครั้งแรกที่มหาวิทยาลัยแมรี่แลนด์ (University of Maryland) โดย David Hovemeyer ซึ่งนักวิจัยท่านนี้ได้ศึกษาและพัฒนาโปรแกรม FindBugs เพื่อใช้ เป็นวิทยานิพนธ์ในระดับปริญญาเอก และต่อมาภายหลังโปรแกรมนี้ได้เป็นที่รู้จักมากยิ่งขึ้นในวงการนักพัฒนาโปรแกรมด้วยจาวา ต่อมาได้มีกลุ่มนักพัฒนารวมตัวกันเพื่อร่วมกันพัฒนาโปรแกรม FindBugs จนถึงปัจจุบันซึ่งบุคคลที่รับผิดชอบหลักในการพัฒนาโปรแกรมนี้คือ Bill Pugh โดยในปัจจุบัน (กรกฎาคม 2555) โปรแกรม FindBugs เวอร์ชันล่าสุดที่ถูกนำมาเผยแพร่คือเวอร์ชัน 2.0.1

FindBugs มีความต้องการ JRE (หรือ JDK) เวอร์ชัน 1.5.0 ขึ้นไปในการประมวลผลโปรแกรม แต่อย่างไรก็ตาม FindBugs สามารถวิเคราะห์โปรแกรมที่ถูกคอมไพล์ด้วยจาวาตั้งแต่เวอร์ชัน 1.0 ถึง 1.8 ซึ่งครอบคลุมเกือบทุกเวอร์ชันของภาษาจาวา

โปรแกรม FindBugs สามารถทำงานร่วมกับโปรแกรม Eclipse ได้เป็นอย่างดีในลักษณะของส่วนเสริมความสามารถ (Plug-in) ของโปรแกรม Eclipse โดยในขณะที่ทำการเขียนโปรแกรมภาษาจาวาโดยใช้โปรแกรม Eclipse อยู่ โปรแกรมเมอร์สามารถรันโปรแกรม FindBugs ที่เป็นส่วนเสริมความสามารถโปรแกรมควบคู่ไปพร้อมกับการรันโปรแกรมที่กำลังเขียนอยู่ได้ โดยในที่นี้จะยกตัวอย่างวิธีการติดตั้งส่วนเสริมความสามารถโปรแกรมและการใช้งานโปรแกรมดังนี้

2.6.1 การติดตั้งโปรแกรมส่วนเสริม FindBugs ลงบน Eclipse

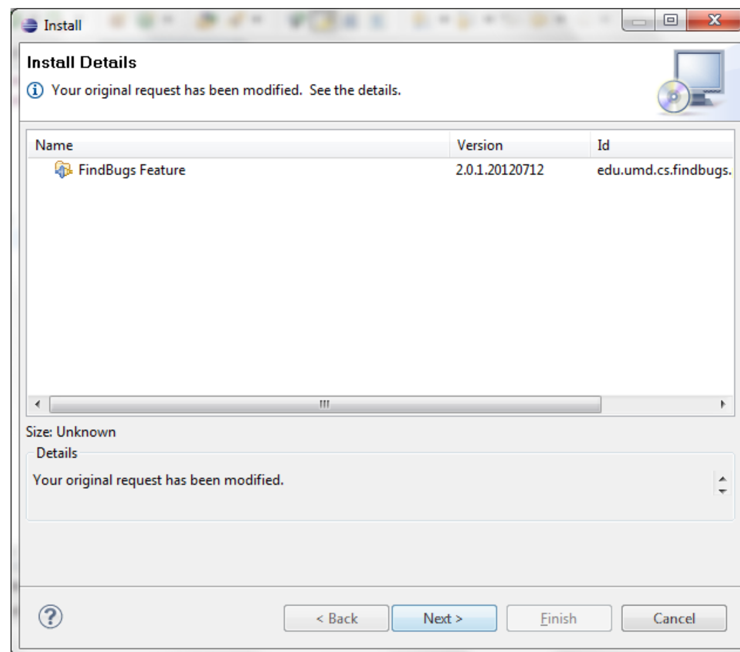
1. เปิดโปรแกรม Eclipse ขึ้นมา โดยในหน้าต่างหลักของโปรแกรมคลิกที่เมนู Help จากนั้นเลือกที่เมนูย่อย Install New Software… จะปรากฏหน้าต่าง Install ขึ้นมาดังรูปที่ 2.20



วิทยาลัยเทคโนโลยีสุรนารี

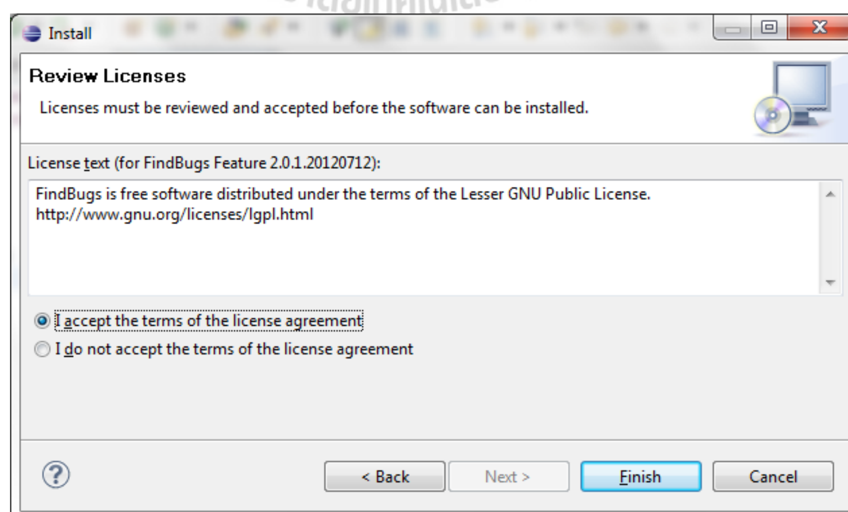
รูปที่ 2.20 หน้าต่างการติดตั้งส่วนเสริมความสามารถ FindBugs ลงบนโปรแกรม Eclipse

2. ในหน้าต่าง Install (รูปที่ 2.20) ในช่อง Work with: ให้กรอก http://findbugs.cs.umd.edu /eclipse จากนั้นจะปรากฏโปรแกรม FindBugs ขึ้นในกล่องรายการ ให้ทำเครื่องหมายถูกในช่องสี่เหลี่ยมหน้ารายการ FindBugs จากนั้นคลิกที่ปุ่ม Next จากนั้นรอจนกระทั่งทำการติดตั้งเครื่องมือเสร็จ (ในขั้นตอนนี้จำเป็นที่ต้องเชื่อมต่อกับอินเตอร์เน็ต) จะปรากฏหน้าต่าง Install Detail ขึ้นมาดังรูปที่ 2.21 ให้คลิกที่ปุ่ม Next



รูปที่ 2.21 หน้าต่างแจ้งเตือนหลังการติดตั้งส่วนเสริม FindBugs เสร็จ

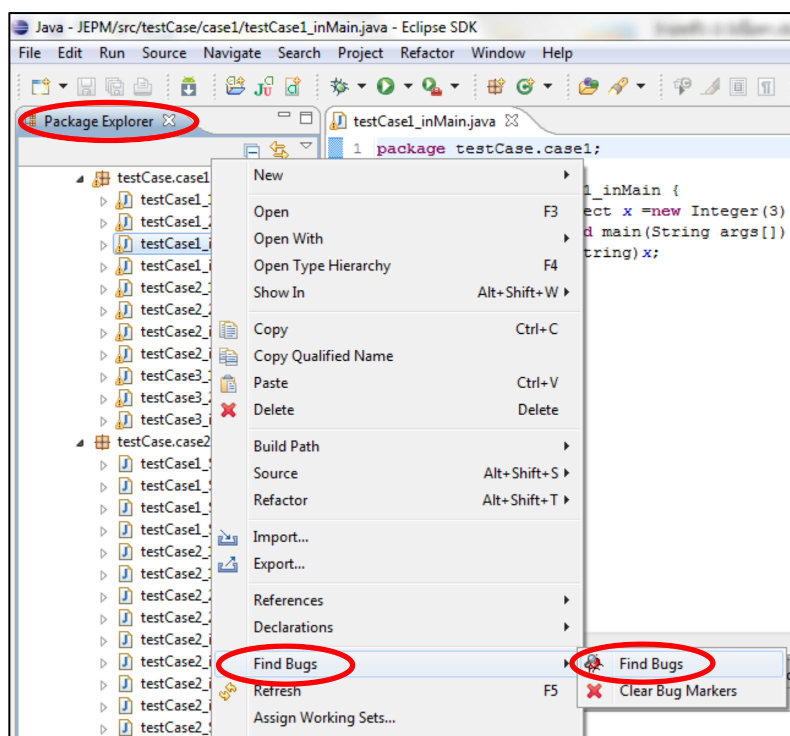
3. จากนั้นจะปรากฎหน้าต่าง Review Licenses ขึ้นมาดังรูปที่ 2.22 ในหน้าต่างนี้ให้ทำการยอมรับเงือนไขการใช้งานโดยเลือกที่ I accept the terms of licenses agreement จากนั้นคลิกที่ปุ่ม Finish เป็นอันเสร็จขั้นตอนการติดตั้งส่วนเสริม FindBugs



รูปที่ 2.22 หน้าต่างแสดงข้อตกลงเบื้องต้นในการใช้งานส่วนเสริม FindBugs

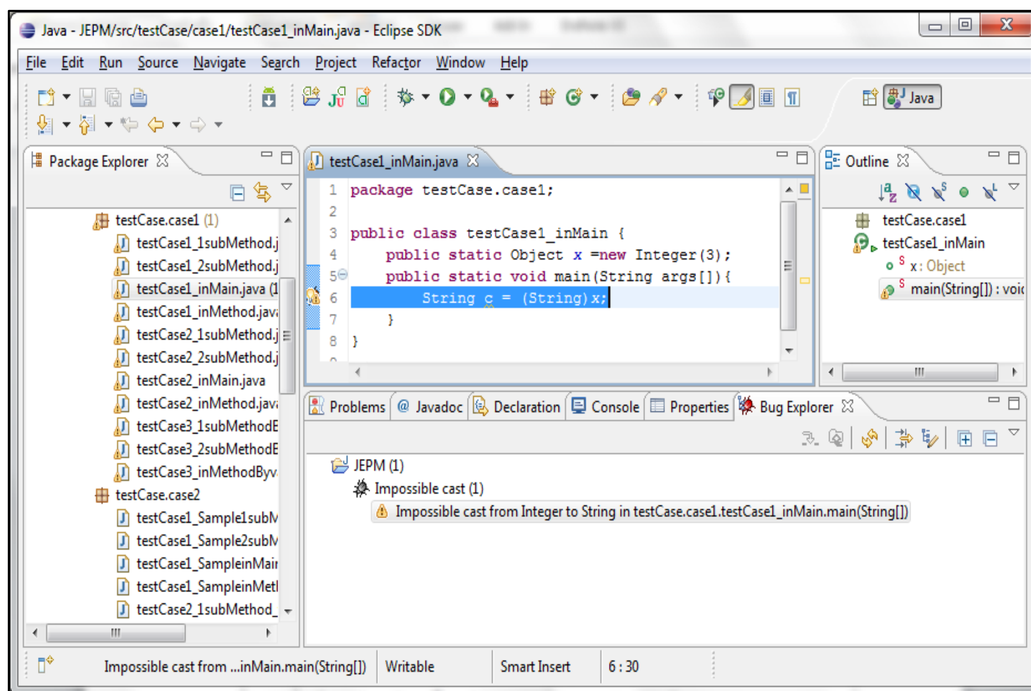
2.6.2 การใช้งานโปรแกรมส่วนเสริม FindBugs บน Eclipse

1. เปิดโปรเจคที่ต้องการทดสอบด้วยโปรแกรม Eclipse ให้หน้าต่าง Package Explorer คลิกขวาที่ไฟล์ซอสโค้ดที่ต้องการทดสอบ เลือกที่เมนู Find Bugs ในเมนูย่อยเลือกที่เมนู Find Bugs ดังรูปที่ 2.23



รูปที่ 2.23 การเริ่มต้นใช้งานโปรแกรมส่วนเสริม FindBugs

2. รอจนกระทั่งประมวลผลเสร็จจากนั้นจะปรากฏหน้าต่าง Bug Explorer ขึ้นมาดังที่แสดงในรูปที่ 2.24 ในหน้าต่างนี้จะแจ้งรายละเอียดของความผิดพลาดที่ถูกพบ โดยในตัวอย่างนี้ได้เกิดความผิดพลายประเภทการแปลงวัตถุอย่างไม่ถูกต้องในบรรทัดที่ 6 ของซอสโค้ด ซึ่งเป็นการพยายามแปลงวัตถุประเภท Integer ให้อยู่ในรูปแบบของวัตถุประเภท String ซึ่งไม่สามารถทำได้



รูปที่ 2.24 ผลลัพธ์ของการทดสอบซอสโค้ดด้วยส่วนเสริม FindBugs

ในงานวิจัยนี้จะทำการศึกษาส่วนเสริม FindBugs เนื่องจากเป็นเครื่องมือตรวจหาความผิดพลาดในภาษาจาวาที่ได้รับความนิยมในปัจจุบัน เพื่อดูลักษณะการทำงานและการรายงานผลของเครื่องมือส่วนเสริม โดยจะใช้ผลลัพธ์จากเครื่องมือนี้ในการอ้างอิงถึงจำนวนความผิดพลาดที่เกิดขึ้นจริง แต่จะไม่นำเอาเครื่องมือส่วนเสริม FindBugs มาเปรียบเทียบกับเครื่องมือตัวอย่างที่ได้พัฒนาขึ้นในงานวิจัย ในส่วนของประสิทธิภาพและความถูกต้องเพียงแต่ใช้ผลลัพธ์ในการอ้างอิงจำนวนและประเภทของความผิดพลาดที่เกิดขึ้นเท่านั้น

2.7 งานวิจัยที่เกี่ยวข้อง

ในการศึกษาและพัฒนาวิธีการตรวจหาความผิดพลาดด้วยการเปรียบเทียบรูปแบบ ผู้วิจัยได้ทำการศึกษาค้นคว้างานวิจัยในอดีตที่เกี่ยวข้องกับการค้นหาความผิดพลาดในภาษาจาวาดังสรุปในตารางที่ 2.4 และมีรายละเอียดดังต่อไปนี้

งานวิจัยของ Hovemeyer and Pugh (2004) ได้นำเสนอลักษณะและวิธีการทำงานของเครื่องมือ FindBugs โดยเลือกตัวอย่างประเภทความผิดพลาดที่น่าสนใจจำนวน 13 ประเภทดังที่แสดงในรูปที่ 2.25 แต่ในเนื้อหางานวิจัยจะกล่าวถึงจริงเพียงแค่ 6 ประเภทเท่านั้น ในงานวิจัยของ Hovemeyer and Pugh อธิบายถึงการค้นหาความผิดพลาดโดยใช้ Bug pattern และทำการทดสอบเปรียบเทียบ FindBugs กับเครื่องมือ 2 ตัว คือ KLOC และ PMD โดยใช้ข้อมูลทดสอบเป็นซอสโค้ดของ Eclipse 3.0 และ rt.jar 1.5.0 build 59 (J2SE core) ซึ่งผู้วิจัยได้สรุปผลการทดสอบว่า เครื่องมือ FindBugs สามารถค้นหาจำนวนความผิดพลาดรวมได้น้อยกว่า PMD เนื่องจากมีการตัดรายงานผลที่ไม่มีความจำเป็นต้องแจ้งเตือนบางส่วนออกไป แต่หากมองในแง่ของการนำไปใช้งานแล้วเครื่องมือ FindBug จะให้รายงานผลที่นำไปใช้ได้ถูกต้องมากกว่า

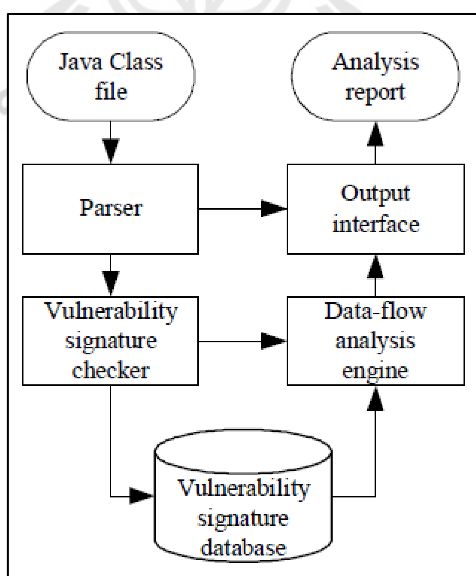
Code	Description
CN	Cloneable Not Implemented Correctly
DC	Double Checked Locking
DE	Dropped Exception
EC	Suspicious Equals Comparison
Eq	Bad Covariant Definition of Equals
HE	Equal Objects Must Have Equal Hashcodes
IS2	Inconsistent Synchronization
MS	Static Field Modifiable By Untrusted Code
NP	Null Pointer Dereference
NS	Non-Short-Circuit Boolean Operator
OS	Open Stream
RCN	Redundant Comparison to Null
RR	Read Return Should Be Checked
RV	Return Value Should Be Checked
Se	Non-serializable Serializable Class
UR	Uninitialized Read In Constructor
UW	Unconditional Wait
Wa	Wait Not In Loop

รูปที่ 2.25 ประเภทของความผิดพลาดที่นำมาศึกษาในงานวิจัยของ Hovemeyer and Pugh (2004)

งานวิจัยของ Murphy, Kim, Kaiser and Cannon (2003) ได้อธิบายว่าความผิดพลาดในการพัฒนาซอฟต์แวร์มี 3 อย่าง คือ ความผิดพลาดที่เกิดระหว่างการคอมไพล์ (Semantic and syntactic) ความผิดพลาดที่เกิดจากความผิดพลาดของตรรกะ (logical) และความผิดพลาดที่เกิดขณะรันโปรแกรม (Runtime Exception) ซึ่งในงานวิจัยชิ้นนี้จะให้ความสำคัญไปที่ความผิดพลาด 2 ประเภทแรก และอธิบายถึงการทำงานของเครื่องมือ Backstop ซึ่งเป็นเครื่องมือที่วิเคราะห์ความผิดพลาดของซอฟต์แวร์ในลักษณะของการวิเคราะห์แบบยืดหยุ่น (Dynamic analysis) โดยใน

งานวิจัยชิ้นนี้ยกตัวอย่างการค้นหาความผิดพลาดประเภทความผิดพลาดของตรรกะที่ได้ทบสอบกับซอสโค้ดโปรแกรมตัวเลข Fibonacci ของนักศึกษาจำนวน 17 คน ผลลัพธ์ปรากฏว่าพบความผิดพลาดของตรรกะในซอสโค้ดของนักศึกษา 8 คน และได้กล่าวถึงการวิจัยในอนาคตที่จะทำให้เครื่องมือ Backstop สามารถวิเคราะห์ความผิดพลาดประเภทที่เกิดขึ้นขณะรันโปรแกรมได้

งานวิจัยของ Zhao, Chen and Wang (2008) ได้นำเสนอขั้นตอนการวิเคราะห์ไบต์โค้ดของภาษาจาวาโดยใช้เทคนิค Data-flow เพื่อค้นหาช่องโว่ (Vulnerability) ที่อาจเกิดขึ้นในโปรแกรมภาษาจาวาด้วยไบต์โค้ดของโปรแกรมซึ่งมีลักษณะการทำงานดังที่แสดงในรูปภาพที่ 2.26 โดยเริ่มจากการกำหนดลักษณะลำดับการทำงานที่ไม่ปลอดภัย จากนั้นวิเคราะห์ไบต์โค้ดเพื่อตรวจหาว่ามีลำดับการทำงานที่ไม่ปลอดภัยอยู่ภายในไบต์โค้ดหรือไม่ ในงานวิจัยชิ้นนี้ได้ยังได้นำเสนอการทำงานร่วมกับฐานข้อมูลเพื่อใช้จัดเก็บ Signature Database ซึ่งเป็นฐานข้อมูลที่เก็บเอกลักษณ์ของลำดับการทำงานที่ไม่ปลอดภัยเอาไว้ และในลำดับสุดท้ายผู้วิจัยได้สรุปการวิจัยว่า เมื่อสามารถตรวจหาการทำงานที่ไม่ปลอดภัยได้แล้วก็จะทำให้ผู้ใช้มีความปลอดภัยในการใช้งานซอฟต์แวร์ที่พัฒนาด้วยภาษาจาจาสูงขึ้น



รูปที่ 2.26 แผนภาพการตรวจหาช่องโหว่จากไบต์โค้ดจาวาของ Zhao, Chen and Wang (2008)

งานวิจัยของ Lance, Untch and Wahl (1999) อธิบายถึงการวิเคราะห์ไบต์โค้ดเพื่อนำไปสร้าง CFG (Control flow graph) ที่แสดงการทำงานอย่างคร่าวๆของโปรแกรม โดยเริ่มจากการนำเอาไบต์โค้ดมาแบ่งออกเป็นชุดคำสั่งด้วยเทคนิค three-address intermediate ที่นำมาจากงานวิจัยของ Aho, Sethi and Ullman (1986) แล้วแทนแต่ละชุดคำสั่งด้วยโหนดของ CFG จากนั้นหาความสัมพันธ์ระหว่างโหนดและสร้าเส้นเชื่อมโหนด ในงานวิจัยชิ้นนี้ได้นำเสนอโครงของเครื่องมือ JAristotle โดยผู้วิจัยกล่าวว่าจะพัฒนาให้เป็นเครื่องมือที่สามารถสร้าง CFG อัตโนมัติได้จากไบต์โค้ดของภาษาจาวา ซึ่งเมื่อสามารถสร้าง CFG จากไบต์โค้ดได้แล้วก็จะทำให้ทราบถึงการทำงานของโปรแกรมได้โดยไม่ต้องอาศับซอสโค้ด นอกจากนั้นผู้วิจัยยังได้ออกแบบเครื่องมือ JAristotle ให้ทำงานร่วมกับฐานข้อมูลในการจัดเก็บไบต์โค้ดเพื่อนำไปวิเคราะห์และสร้าง CFG ต่อไป

งานวิจัยของ William Pugh (2007) เป็นงานวิจัยต่อเนื่องจากงานวิจัยของ Hovemeyer and Pugh (2004) อธิบายเกี่ยวกับการทำงานที่เพิ่มขึ้นของโปรแกรม FindBugs ที่มีการเพิ่มความสามารถใหม่ๆเข้าไปโดยผู้วิจจัยได้อธิบายเทคนิคที่ใช้ในเครื่องมือ FindBugs จำนวน 4 เทคนิคดังนี้

1. เทคนิคการเปรียบเทียบรูปแบบ (Local pattern matching) เป็นการเปรียบเทียบโดยการค้นหาคำสั่งที่อาจทำให้เกิดความผิดพลาดภายในซอสโค้ด ซึ่งรูปแบบของซอสโค้ดที่ถูกระบุว่าก่อให้เกิดความผิดพลาดจะถูกกำหนดเอาไว้ก่อน
2. เทคนิคการวิเคาะห์การทำงานของโปรแกรม (Intraprocedural dataflow analysis) เป็นวิธิการพี่พัฒนาขึ้นเพื่อตรวจหาความผิดพลาดประเภท Null pointer exception และ Type cast exception
3. เทคนิคการสรุปเมท็อด (Interprocedural method summaries) เป็นเทคนิคที่ใช้ในการค้นหาความผิดพลาดประเภทที่เกิดจากการส่งผ่านค่าตัวแปรระหว่างเมท็อด
4. เทคนิคการวิเคราห์บริบท (Context sensitive interprocedural analysis) เป็นเทคนิคที่ใช้เพื่อป้องกันการทำ SQL injection และ cross site scripting

ในงานวิจัยชิ้นนี้ผู้วิจัยได้กล่าวถึงการนำเอาเครื่องมือ FindBugs ไปใช้อย่างแพร่หลาย และอธิบายถึงประสิทธิภาพการทดสอบที่ทำให้ FindBug มีประสิทธิภาพสูงกว่าเครื่องมือทดสอบซอฟต์แวร์ภาษาจาวาตัวอื่นๆ

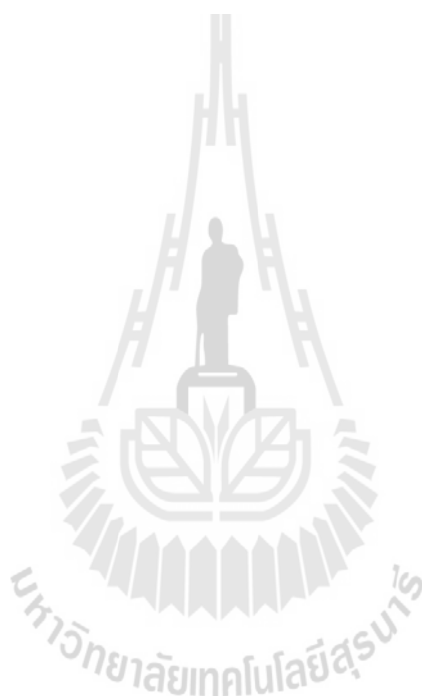
ตารางที่ 2.4 แสดงการสรุปเปรียบเทียบประเด็นต่างๆ ของงานวิจัยที่เกี่ยวข้อง โดยบทความวิจัยที่เกี่ยวข้องประกอบด้วย “ก” แทนงานวิจัยของ Hovemeyer and Pugh (2004) “ข” แทนงานวิจัยของ Murphy, Kim, Kaiser and Cannon (2003) “ค” แทนงานวิจัยของ Zhao, Chen and Wang (2008) “ง” แทนงานวิจัยของ Lance, Untch and Wahl (1999) “จ” แทนงานวิจัยของ William Pugh (2007) และ “ฉ” แทนงานวิจับเรื่อง การพัฒนาวิธีการตรวจหาความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวา (งานวิจัยของวิทยานิพนธ์ฉบับนี้)

ตารางที่ 2.4 สรุปเปรียบเทียบงานวิจัยที่เกี่ยวข้องกับการการพัฒนาวิธีการตรวจหาความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวา

กระบวนการทำงาน	**งานวิจัยที่เกี่ยวข้อง**					
	ก	**ข**	**ค**	**ง**	**จ**	**ฉ**
การหาความผิดพลาด						
วิธีการวิจัยเกี่ยวข้องกับไบต์โค้ด			✔	✔		✔
วิธีการทดสอบใช้ไบต์โค้ดเป็นกรณีทดสอบ				✔		✔
วิธีการวิเคราะห์ความผิดพลาดเป็นแบบคงที่ (Statc analysis)	✔		✔	✔	✔	✔
วิธีการเปรียบเทียบเกี่ยวข้องกับการเปรียบเทียบรูปแบบ (Pattern matching)	✔				✔	✔
วิธีการถูกนำเสนอด้วยภาษาจาวา	✔		✔	✔	✔	✔
วิธีการมีการกำหนดต้นแบบความผิดพลาด	✔		✔		✔	✔
การพัฒนา						
พัฒนาด้วยภาษาจาวาทั้งหมด	✔		✔	✔	✔	✔
พัฒนาโดยใช้แนวคิดการเปรียบเทียบรูปแบบ	✔	✔			✔	✔
พัฒนาในลักษณะคลังโปรแกรม (Library)						✔
พัฒนาให้เชื่อมต่อกับฐานข้อมูล			✔		✔	✔
ส่วนอธิบายความ						
มีการอธิบายถึงความเป็นมาของผลลัพธ์ที่ได้	✔	✔		✔		✔
ส่วนติดต่อผู้ใช้						
สามารถตอบโต้กับผู้ใช้งานได้ (User interface)	✔	✔	✔		✔	

ตารางที่ 2.4 สรุปเปรียบเทียบงานวิจัยที่เกี่ยวข้องกับการการพัฒนาวิธีการตรวจหาความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวา (ต่อ)

การประยุกต์ใช้						
วิจัยเพื่อทดสอบประสิทธิภาพ	✔				✔	✔
วิจัยเพื่อวางแผนพัฒนาระบบ		✔	✔	✔		✔
มีการประยุกต์ใช้กับการทดสอบความผิดพลาดจริง	✔				✔	



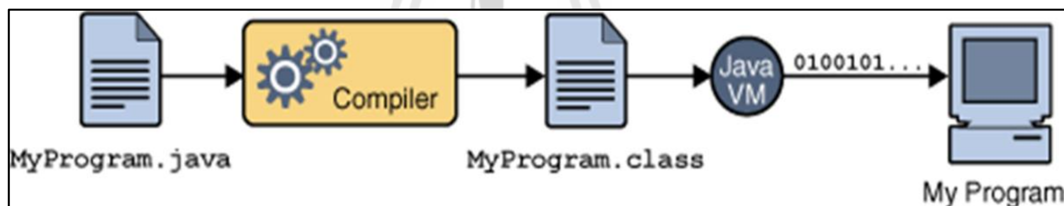
บทที่ 3
วิธีดำเนินการวิจัย

3.1 วิธีวิจัย

ในงานวิจัยนี้เป็นการค้นหาวิธีการตรวจหาความผิดพลาดของโปรแกรมภาษาจาวาโดยใช้ไบต์โค้ด โดยขั้นตอนการศึกษาจะเริ่มต้นด้วยการศึกษาเกี่ยวกับไบต์โค้ดซึ่งมีรายละเอียดดังนี้

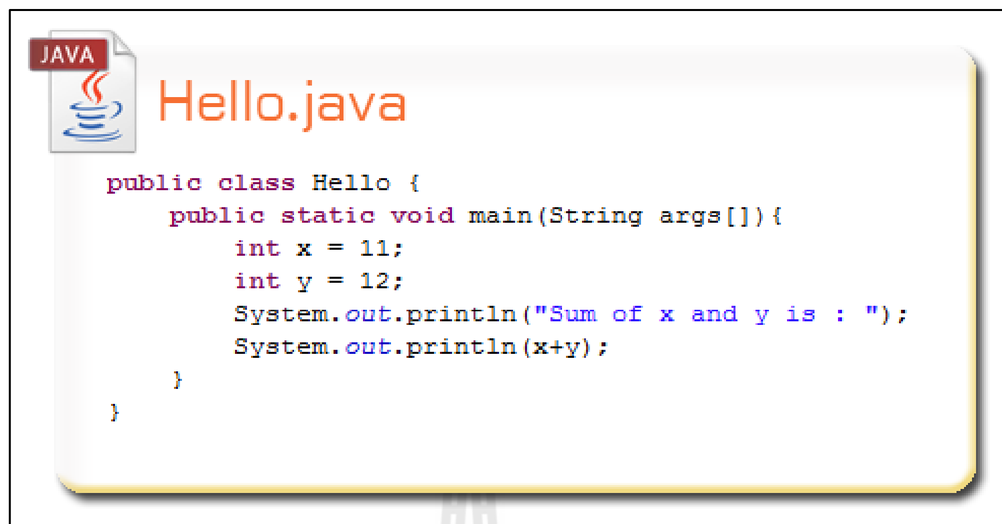
3.1.1 การทำงานของไบต์โค้ด

ในขั้นตอนการคอมไพล์โปรแกรมภาษาจาวาผลลัพธ์สุดท้ายจะได้ไฟล์นามสกุล .class ออกมาซึ่งภายในบรรจุไบต์โค้ดเอาไว้ ดังรูปที่ 3.1



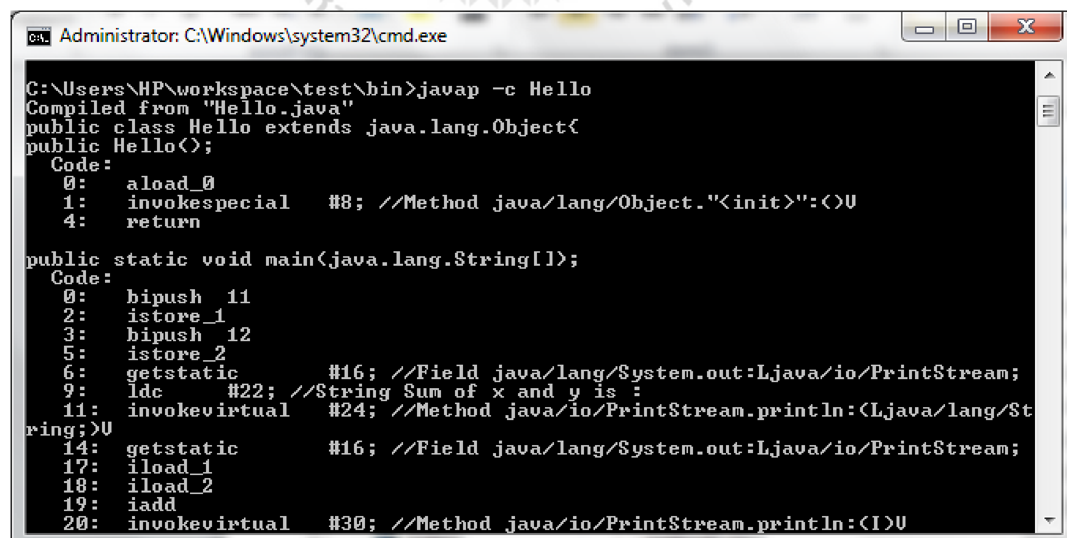
รูปที่ 3.1 การคอมไพล์โปรแกรมในภาษาจาวา (Sun Microsystems, Inc., 2006)

จากรูปที่ 3.1 ในไฟล์ MyProgram.class จะมีไบต์โค้ดบรรจุอยู่ภายในซึ่งไม่สามารถเปิดอ่านและทำความเข้าใจได้ ซึ่งหากต้องการแปลงให้อยู่ในรูปแบบของไบต์โค้ดที่สามารถอ่านแล้วเข้าใจได้ต้องใช้คำสั่ง “javap –c ชื่อไฟล์ที่ต้องการ” เพื่อทำการจัดรูปแบบของไบต์โค้ด โดยคำสั่ง javap จะมีอยู่แล้วในจาวาเวอร์ชวลแมชีนสามมารถเรียกใช้ได้เช่นเดียวกับคำสั่ง javac (คำสั่งในการคอมไพล์ของจาวา)



รูปที่ 3.2 ตัวอย่างซอสโค้ดของภาษาจาวา

ในขั้นตอนนี้ผู้วิจัยจะยกตัวอย่างโปรแกรมภาษาจาวาอย่างง่ายชื่อ Hello.java ซึ่งมีซอสโค้ดของโปรแกรมดังเช่นที่แสดงในรูปที่ 3.2 จากนั้นจะทำการคอมไพล์ซอสโค้ดนี้ให้อยู่ในรูปของไฟล์นามสกุล .class เพื่อนำไปรันด้วยคำสั่ง javap ดังรูปที่ 3.3



รูปที่ 3.3 ผลลัพธ์เมื่อใช้คำสั่ง javap –c

จากรูปที่ 3.3 เป็นคำสั่งของไบต์โค้ดที่ได้จากไฟล์ Hello.class จะสังเกตว่าจะมีคำสั่งที่แบ่งออกอย่างเป็นรูปแบบและสามารถอ่านทำความเข้าใจได้ ซึ่งผู้วิจัยจะนำมาศึกษาถึงวิธีการทำงานของชุดคำสั่งของไบต์โค้ดโดยสามารถแบ่งข้อมูลที่มีประโยชน์ได้ดังนี้

Hello.class (Bytecode)

```
Compiled from "Hello.java"
public class Hello extends java.lang.Object{
public Hello();
  Code:
   0:   aload_0
   1:   invokespecial   #8; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   bipush  11
   2:   istore_1
   3:   bipush  12
   5:   istore_2
   6:   getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   9:   ldc     #22; //String Sum of x and y is :
   11:  invokevirtual   #24; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   14:  getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   17:  iload_1
   18:  iload_2
   19:  iadd
   20:  invokevirtual   #30; //Method java/io/PrintStream.println:(I)V
   23:  return

}
```

รูปที่ 3.4 ส่วนประกอบต่างๆในไบต์โค้ด

1. แหล่งที่มาของการคอมไพล์ โดยสังเกตจากประโยค Compiled from “ชื่อไฟล์.java” ซึ่งหากบรรทัดใดมีคำสั่งนี้ประกอบอยู่จะสามารถนำมาหาชื่อไฟล์ก่อนการคอมไพล์ได้ซึ่งในที่นี้คือ Hello.java

2. ชื่อคลาส หากบรรทัดนั้นประกอบด้วยคำว่า class และประโยค extends java.lang .Object จะสามารถนำมาหาชื่อของคลาสได้ ซึ่งในที่นี้คือคลาส Hello

3. เมท็อด หากคำสั่งบรรทัดใดประกอบไปด้วยเครื่องหมาย () และ ; แสดงว่าบรรทัดนั้นจะเป็นการประกาศชื่อเมท็อด โดยในที่นี้ มี 2 เมท็อดคือ Hello ซึ่งเป็นเมท็อดที่มีชื่อเดียวกับชื่อคลาส (Default constructor) และเมท็อด main ซึ่งเป็นเมท็อดเริ่มต้นของโปรแกรม

4. คำสั่งแต่ละบรรทัดภายในเมท็อด ซึ่งลักษณะคำสั่งจะถูกอธิบายในหัวข้อที่ 3.1.2
5. คำสั่ง return ซึ่งเป็นการระบุว่าสิ้นสุดการทำงานของเมท็อดนั้นแล้ว

3.1.2 ลักษณะคำสั่งของไบต์โค้ด

ในหัวข้อที่ 3.1.1 ได้ทำการแบ่งส่วนประกอบของไบต์โค้ดที่ได้จากการรันคำสั่ง javap –c ซึ่งส่วนประกอบที่สำคัญที่สุดคือคำสั่งภายในเมท็อดแต่ละบรรทัด โดยเมื่อสังเกตแล้วจะพบว่ามีลำดับ และสัญลักษณ์ที่สำคัญ ดังนี้

```
0 :        bipush   11
```

รูปที่ 3.5 ตัวอย่างคำสั่งแบบปรกติของไบต์โค้ด

1. คำสั่งแบบปรกติ เป็นคำสั่งที่ประกอบด้วย หมายเลขลำดับคำสั่ง คำสั่ง และตัวแปร ซึ่งหมายเลขบรรทัดจะอยู่ในตำแหน่งต้นบรรทัดและปิดด้วยเครื่องหมาย : จากรูปที่ 3.5 ในที่นี้หมายเลขลำดับคำสั่งคือ 0 ต่อมาคือคำสั่งไบต์โค้ด ซึ่งเป็นคำสั่งที่ใช้ดำเนินการจะอยู่ถัดจากหมายเลขบรรทัดซึ่งจะถูกคั่นด้วยเครื่องหมาย : ในที่นี้คือคำสั่ง bipush หมายถึงการใส่ตัวแปร integer ลงไปในสแตก (Stack) และสุดท้ายคือตัวแปร (บางคำสั่งอาจไม่มีตัวแปร) ซึ่งจะเป็นข้อมูลที่จะนำไปใช้ประมวลผลร่วมกับคำสั่งในที่นี้คือ 11 รวมแล้วคือคำสั่งลำดับที่ 0 จะทำการใส่ตัวแปรแบบ Integer ที่มีค่า 11 ลงไปในสแตก

```
2 :        istore_1
```

รูปที่ 3.6 ตัวอย่างคำสั่งที่อ้างอิงถึงตำแหน่งข้อมูลของไบต์โค้ด

2. คำสั่งที่อ้างอิงถึงตำแหน่งข้อมูล คำสั่งประเภทนี้จะมีจุดสังเกตโดยภายในคำสั่งจะคั่นระหว่างคำสั่งและตัวแปรด้วยเครื่องหมาย _ ซึ่งในคำสั่งปรกติจะใช้เครื่องหมายวรรคตอนในการแบ่งแยกคำสั่ง เมื่อดูจากรูปที่ 3.6 คำสั่งคือ istore หมายถึง การเก็บตัวแปร integer ไปยังตำแหน่งที่กำหนดไว้ต่อจากเครื่องหมาย _ ซึ่งในที่นี้คือ ตำแหน่งที่ 1

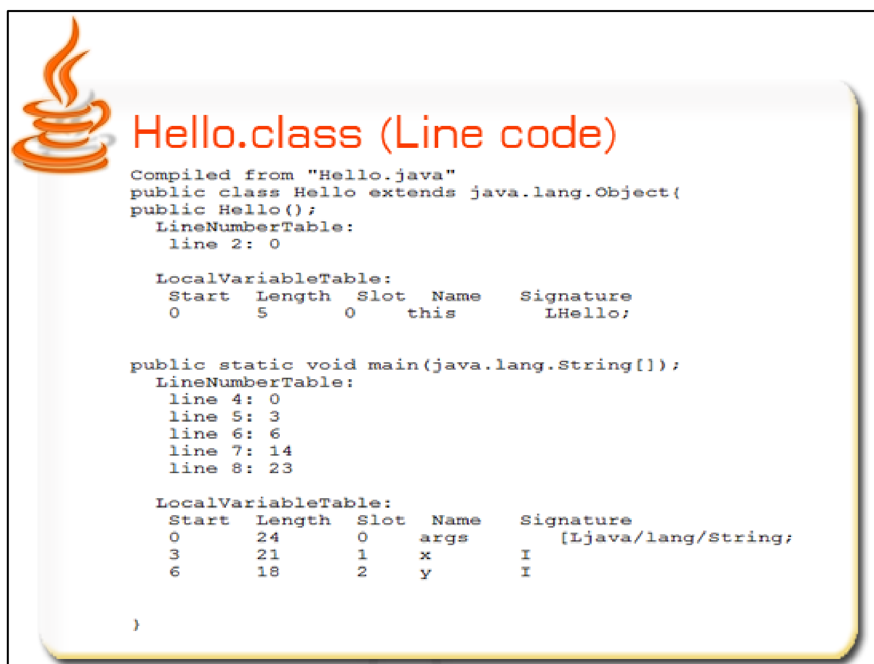
```
11:     invokevirtual     #24; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

รูปที่ 3.7 ตัวอย่างคำสั่งที่อ้างอิงถึงคำสั่งภายใน

3. คำสั่งที่อ้างอิงถึงคำสั่งภายใน คำสั่งประเภทนี้มักมีเครื่องหมาย # และ // ประกอบอยู่ภายในบรรทัด ซึ่งสิ่งที่อยู่ด้านหลังเครื่องหมาย # คือหมายเลขของคำสั่งที่จะนำมาใช้ และหลังเครื่องหมายจะเป็นการอธิบายว่าคำสั่งหมายเลขดังกล่าวคืออะไร จากรูปที่ 3.7เป็นการเรียกใช้คำสั่ง invokevirtual ซึ่งเป็นคำสั่งในการเรียกใช้คำสั่งภายในจาวาเวอร์ชวลแมชีน ในที่นี้คือคำสั่งหมายเลข 24 ซึ่งเปรียบได้กับการใช้คำสั่ง System.out.println() ในซอสโค้ดนั่นเอง

3.1.3 การแบ่งชุดคำสั่งของไบต์โค้ด

ในการทำงานของภาษาจาวาเมื่อคอมไพล์คำสั่งในซอสโค้ด 1 บรรทัด จะได้คำสั่งในไบต์โค้ดจำนวนหนึ่ง (ในที่นี้เรียกว่าชุดคำสั่ง) ซึ่งในขณะนี้ยังไม่สามารถที่จะระบุได้ว่าหนึ่งชุดคำสั่งประกอบไปด้วยคำสั่งหมายเลขลำดับใดบ้าง โดยหากต้องการทราบจะต้องอาศัยผลลัพธ์ที่ได้จากการรันไฟล์นามสกุล .class ที่ต้องการด้วยคำสั่ง “javap –l ชื่อไฟล์” ซึ่งผลลัพธ์ที่ได้จะมีลักษณะดังรูปที่ 3.8



รูปที่ 3.8 ผลลัพธ์การรันคำสั่ง javap –l

จากรูปที่ 3.8 แสดงข้อความผลลัพธ์จำนวนหนึ่งโดยในงานวิจัยจะเรียกผลลัพธ์นี้ว่า ไลน์โค้ด (Line number) ซึ่งภายในจะมีส่วนประกอบที่สำคัญที่จำเป็นในบทนี้คือส่วนของหมายเลขบรรทัด ซึ่งหากบรรทัดใดเป็นการระบุการจำแนกชุดคำสั่งของไบต์โค้ดแล้ว ในบรรทัดนั้นจะขึ้นต้นด้วยข้อความ line และตามด้วยหมายเลข ดังแสดงในรูปที่ 3.9

```
line 4 : 0
line 5 : 3
```

รูปที่ 3.9 ตัวอย่างคำสั่งของไลน์โค้ด

จากรูปที่ 3.9 เป็นตัวอย่างบางส่วนของไลน์โค้ดซึ่งจากตัวอย่างจะสามารถบอกได้ว่าคำสั่งในซอสโค้ดหมายเลขบรรทัดที่ 4 คือคำสั่งในไบต์โค้ดเริ่มที่คำสั่งลำดับที่ 0 ถึงคำสั่งที่อยู่ก่อนลำดับ

ที่ 3 ซึ่งระบุเป็นจุดเริ่มต้นของบรรทัดถัดไป เมื่อนำไลน์โค้ดมาแบ่งชุดคำสั่งของไบต์โค้ดแล้วจะสามารถแบ่งได้ดังรูปที่ 3.10

Hello.class (Bytecode)

```
Compiled from "Hello.java"
public class Hello extends java.lang.Object{
public Hello();
  Code:
   0:   aload_0
   1:   invokespecial   #8; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   bipush  11
   2:   istore_1
   3:   bipush  12
   5:   istore_2
   6:   getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   9:   ldc     #22; //String Sum of x and y is :
   11:  invokevirtual   #24; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   14:  getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   17:  iload_1
   18:  iload_2
   19:  iadd
   20:  invokevirtual   #30; //Method java/io/PrintStream.println:(I)V
   23:  return
}
```

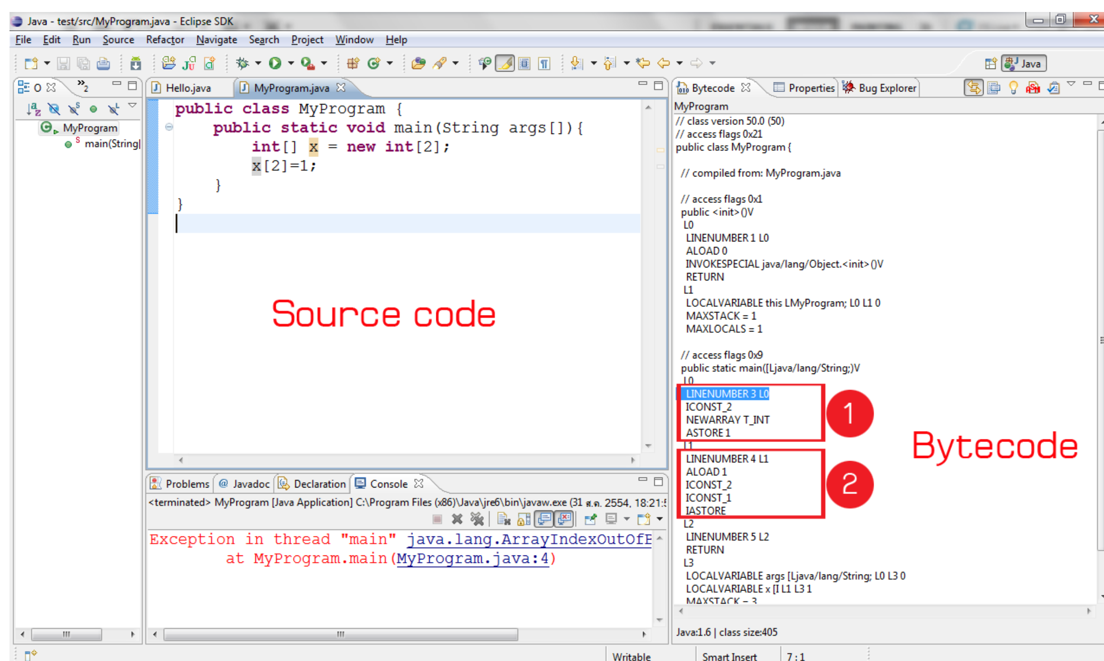
รูปที่ 3.10 การแบ่งชุดคำสั่งของไบต์โค้ดโดยอาศัยไลน์โค้ด

ในขั้นตอนนี้ผู้วิจัยสามารถแบ่งชุดคำสั่งภายในไบต์โค้ดได้ และยังพบอีกว่าสามารถอ้างอิงไปถึงหมายเลขบรรทัดของซอสโค้ดได้ ซึ่งในขั้นตอนต่อไปจะเป็นการนำเอาชุดคำสั่งที่ได้ไปหารูปแบบของชุดคำสั่งที่อาจทำให้เกิดความผิดพลาด เพื่อสร้างเป็นวิธีการตรวจหาความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวาต่อไป

3.1.4 การค้นหารูปแบบการเกิดความผิดพลาดของชุดคำสั่งไบต์โค้ด

ในขั้นตอนนี้จะเป็นการใช้โปรแกรมประยุกต์ Eclipse ร่วมกับส่วนเสริมความสามารถ Bytecode outline ในการศึกษารูปแบบของชุดคำสั่งที่จะทำให้เกิดความผิดพลาดในภาษาจาวาซึ่งมีขั้นตอนดังนี้

1. สร้างซอสโค้ด คือ การสร้าง หรือ นำเอา (อาจได้จากการค้นหาบนเว็บไซต์) ซอสโค้ดที่ทำให้เกิดความผิดพลาดมารันด้วยโปรแกรม Eclipse เพื่อหาว่าตำแหน่งใดที่เกิดความผิดพลาดขึ้น โดยในที่นี้จะยกตัวอย่างซอสโค้ดอย่างง่ายที่ทำให้เกิดความผิดพลาดประเภทการอ้างอิงถึงอาร์เรย์เกินขอบเขตที่กำหนด (Array index out of bound) ดังรูปที่ 3.11



รูปที่ 3.11 ตัวอย่างโปรแกรม Eclipse กับส่วนเสริม Bytecode outline

จากรูปที่ 3.11 จะเห็นว่าในซอสโค้ดบรรทัดที่ 4 เกิดความผิดพลาดขึ้นบนคำสั่ง x[2]=1 เนื่องจากเข้าถึงอาร์เรย์ในตัวแปร x เกินขนาดที่กำหนดไว้ในบรรทัดที่ 3 ซึ่งเมื่อเปรียบเทียบกับไบต์โค้ดแล้วคำสั่งที่ทำให้เกิดความผิดพลาดในซอสโค้ดบรรทัดที่ 4 เมื่อแปลงเป็นชุดคำสั่งไบต์โค้ดแล้วจะได้ชุดคำสั่งไบต์โค้ดหมายเลข 2 และซอสโค้ดบรรทัดที่ 3 คือชุดคำสั่งไบต์โค้ดหมายเลข 1 โดยในงานวิจัยนี้จะเรียกชุดคำสั่งที่ถูกแบ่งแล้วว่า “แพทเทิร์น (Pattern)” ซึ่งในตัวอย่างนี้จะเกิดความผิดพลาดขึ้นหากขนาดที่กำหนดไว้ในตัวแปรคำสั่งไบต์โค้ดชุดที่ 1 น้อยกว่าตัวแปรที่ชุดคำสั่งที่ 2 เข้าถึง

2. หาชุดคำสั่งที่ทำให้เกิดความผิดพลาด และชุดคำสั่งเงื่อนไข ในที่นี้ชุดคำสั่งที่ทำให้เกิดความผิดพลาดคือชุดคำสั่งหมายเลข 2 และชุดคำสั่งเงื่อนไขคือชุดคำสั่งชุดที่ 1 ดังภาพต่อไปนี้

```
1  LINENUMBER 3 L0
2  ICONST_2
3  NEWARRAY T_INT
4  ASTORE 1
```
1

```
1  LINENUMBER 4 L1
2  ALOAD 1
3  ICONST_2
4  ICONST_1
5  IASTORE
```
2

รูปที่ 3.12 ตัวอย่างชุดคำสั่งของไบต์โค้ด

จากภาพตัวอย่างด้านบนในชุดคำสั่งที่ 1 เป็นการสร้างตัวแปรแบบอาร์เรย์ของ integer ที่มีขนาดเท่ากับ 2 โดยมีการทำงานของชุดคำสั่งแยกย่อยแต่ละบรรทัด ดังนี้

บรรทัดที่ 1 : เป็นการอ้างอิงว่าชุดคำสั่งไบต์โค้ดนี้คือคำสั่งในซอสโค้ดบรรทัดที่ 3

บรรทัดที่ 2 : สร้างค่าคงที่ประเภท integer ในที่นี้คือ 2

บรรทัดที่ 3 : สร้างอาร์เรย์ที่มีชนิดเป็น integer โดยให้มีขนาดเท่ากับค่าคงที่ที่ประกาศไว้ในบรรทัดที่แล้วในตัวอย่างนี้คือ 2 นั่นเอง

บรรทัดที่ 4 : เก็บอาร์เรย์นี้ไว้ในหน่วยความจำจำลองของเวอร์ชวลแมชีนตำแหน่งที่ 1

ต่อมาในชุดคำสั่งที่ 2 เป็นชุดคำสั่งในการเข้าถึงอาร์เรย์ในตำแหน่งที่ 2 ซึ่งมีรายละเอียดการทำงานแต่ละบรรทัดตามลำดับดังนี้

บรรทัดที่ 1 : เป็นการอ้างอิงว่าชุดคำสั่งไบต์โค้ดนี้คือคำสั่งในซอสโค้ดบรรทัดที่ 4

บรรทัดที่ 2 : ดึงข้อมูลอาร์เรย์ที่เก็บไว้ในหน่วยความจำลองของเวอร์ชวลแมชีนตำแหน่งที่ 1 ขึ้นมา (จะสังเกตเห็นว่าอ้างอิงตำแหน่งเดียวกันกับชุดคำสั่งที่ 1)

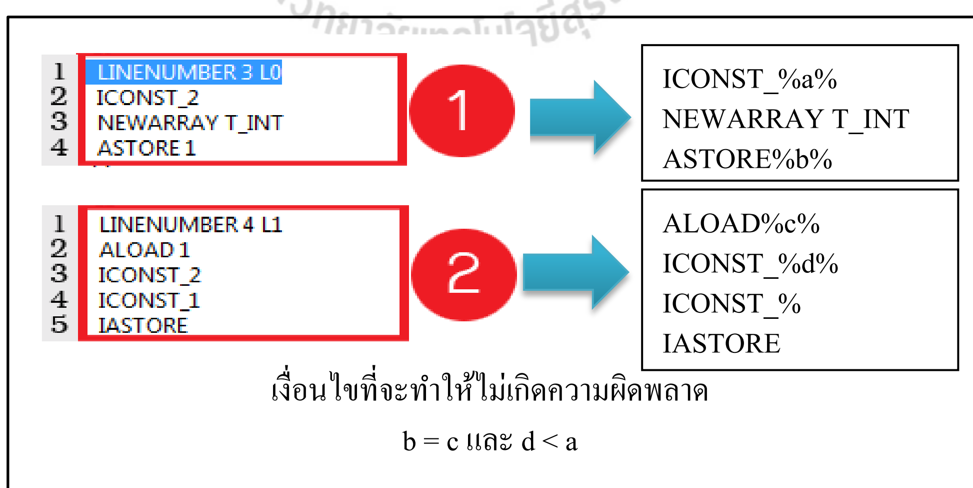
บรรทัดที่ 3 : สร้างค่าคงที่ประเภท integer ในที่นี้คือ 2 ในที่นี้ใช้เพื่ออ้างถึงตำแหน่งในอาร์เรย์ที่ทำการดึงข้อมูลขึ้นมาในบรรทัดที่แล้วซึ่งทำให้เกิดความผิดพลาด เนื่องจากอาร์เรย์ที่ดึงขึ้นมาสามารถเข้าถึงได้ในตำแหน่ง 0 และ 1 เท่านั้น

บรรทัดที่ 4 : สร้างค่าคงที่ประเภท integer ในที่นี้คือ 1 (ในที่นี้ใช้เพื่อใส่ค่าลงในอาร์เรย์ตำแหน่งที่ระบุเอาไว้ในบรรทัดที่แล้วนั่นคือในตำแหน่งที่ 2)

บรรทัดที่ 5 : นำอาร์เรย์ของ integer ที่ถูกใส่ค่าแล้วเก็บลงในตำแหน่งเดิม

จากตัวอย่างนี้เราจะพบว่าชุดคำสั่งของไบต์โค้ดชุดที่ 1 ในตำแหน่งบรรทัดที่ 2 จะสัมพันธ์กับคำสั่งไบต์โค้ดชุดที่สองในตำแหน่งบรรทัดที่ 3 ซึ่งถ้าหากค่าที่กำหนดในชุดคำสั่งที่ 2 มีค่าน้อยกว่าหรือเท่ากับค่าที่กำหนดในชุดที่ 1 แล้วจะทำให้เกิดความผิดพลาดประเภทการเข้าถึงอาร์เรย์เกินขอบเขตที่กำหนดได้ ซึ่งจะนำแพทเทิร์นทั้งสองตัวนี้ไปใช้เพื่อเป็นต้นแบบในการค้นหาความผิดพลาดประเภทนี้กับไบต์โค้ดของโปรแกรมอื่นๆได้

3. จัดเก็บแพทเทิร์นลงในฐานข้อมูลตรวจสอบ เมื่อทราบถึงลักษณะของแพทเทิร์นและเงื่อนไขที่ทำให้เกิดความผิดพลาดแล้วในขั้นตอนนี้จะเป็นการแปลงแพทเทิร์นเพื่อเก็บในฐานข้อมูลตรวจสอบโดยลักษณะการแปลงข้อมูลสามารถแสดงได้ดังภาพต่อไปนี้



รูปที่ 3.13 การแปลงชุดคำสั่งไบต์โค้ดเพื่อจัดเก็บในฐานข้อมูลตรวจสอบ

จากรูปภาพเมื่อได้ชุดคำสั่งที่มีการแปลงรูปแบบแล้วข้อมูลจะถูกบันทึกลงในฐานข้อมูลตรวจสอบ โดยในฐานข้อมูลตรวจสอบจะมีการจัดเก็บข้อมูลดังรูปที่ 3.14

ID	Check_Pattern	Condition_Pattern	Rule
1	ICONST_%a% NEWARRAY T_INT ASTORE%b%	ALOAD%c% ICONST_%d% ICONST_% IASTORE	b = c d < a

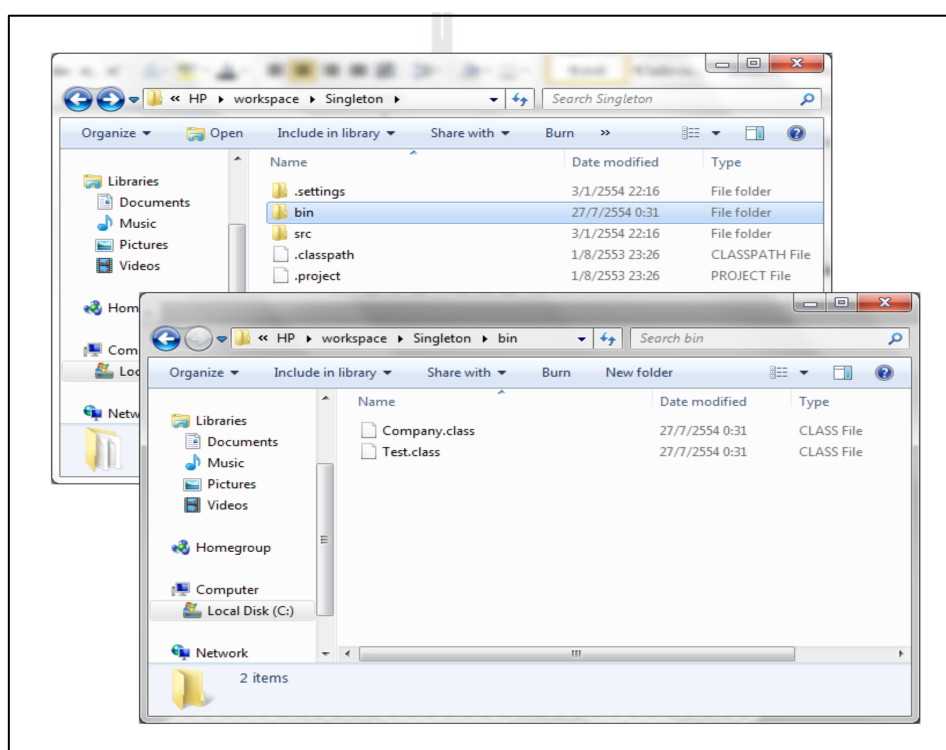
รูปที่ 3.14 ลักษณะข้อมูลที่ถูกจัดเก็บอยู่ในฐานข้อมูลตรวจสอบ

ในฐานข้อมูลตรวจสอบจะประกอบไปด้วยฟิลด์ข้อมูลที่ใช้เก็บชุดคำสั่งและข้อมูลอื่นๆเพื่อใช้ในการตรวจหาความผิดพลาด ซึ่งสามารถเพิ่มข้อมูลชุดคำสั่งของไบต์โค้ดที่อาจทำให้เกิดความผิดพลาดใหม่เข้าไปได้เมื่อมีการแปลงรูปแบบของชุดคำสั่งแล้ว โดยในฐานข้อมูลตรวจสอบนี้มีฟิลด์ข้อมูลที่สำคัญอยู่สามฟิลด์คือ Check_Pattern ฟิลด์นี้จัดเก็บชุดคำสั่งเริ่มต้นที่ใช้ค้นหาว่าพบคำสั่งชุดนี้ในโปรแกรมหรือไม่โดยหากพบชุดคำสั่งนี้แล้วต่อมาจะต้องค้นหาชุดคำสั่งในฟิลด์ที่สองชื่อ Condition_Pattern ซึ่งข้อมูลทั้งสองฟิลด์ที่กล่าวมานี้จะมีความสัมพันธ์กัน และในขั้นตอนสุดท้ายจะต้องเข้าสู่กระบวนการตรวจสอบกฎความถูกต้องซึ่งกฎที่ได้จะถูกบันทึกไว้ในฟิลด์ Rule

3.1.5 การสร้างเครื่องมือตัวอย่างสำหรับการค้นหาความผิดพลาดในภาษาจาวาด้วยไบต์โค้ด

ในหัวข้อนี้จะเป็นการสร้างเครื่องมือตัวอย่างเพื่อทดสอบการหาความผิดพลาดโดยใช้แพทเทิร์นที่ได้นำเสนอไปในหัวข้อที่แล้วร่วมกับฐานข้อมูล H2 โดยใช้ภาษาจาวาในการพัฒนาซึ่งเครื่องมือตัวอย่างนี้จะมีลักษณะเป็นคลังโปรแกรม (Library) ที่ทำงานร่วมกับโปรแกรมภาษาจาวาอื่นๆโดยอาศัยไฟล์ตระกูล .class จากโปรแกรมที่เรียกใช้คลังโปรแกรมเป็นแหล่งข้อมูลนำเข้าของเครื่องมือ ซึ่งจะทำให้สามารถเรียกใช้เครื่องมือไปพร้อมๆกับการรันโปรแกรมที่ต้องการได้ โดยเครื่องมือที่สร้างขึ้นจะใช้ฐานข้อมูล H2 เพื่อเก็บแพทเทิร์นของไบต์โค้ดที่จะทำให้เกิดความผิดพลาด และเก็บข้อมูลนำเข้าที่ได้จากการประมวลผลแล้ว ซึ่งข้อได้เปรียบอีกประการของการเก็บ

ข้อมูลไว้ในฐานข้อมูลคือ สามารถที่จะเข้าถึงข้อมูลได้โดยการสอบถามข้อมูล (Query) โดยใช้ภาษา SQL ในภาษานี้จะมีคำสั่งในการสอบถามข้อมูลที่เอื้อต่อการเปรียบเทียบรูปแบบ เช่นคำสั่ง LIKE ที่สามารถหาข้อความที่มีลักษณะคล้ายกันได้ เป็นต้น อีกทั้งยังสามารถลดระยะเวลาในการเข้าถึงข้อมูล เนื่องจากสามารถสอบถามข้อมูลได้โดยใช้คำสั่ง SQL ทำให้สามารถเข้าถึงข้อมูลที่ต้องการได้ทันที ซึ่งจะส่งผลให้เครื่องมือที่สร้างขึ้นไม่ต้องเสียเวลากับการจัดการเรื่องการค้นหาข้อมูล และฐานข้อมูล H2 ยังมีคุณสมบัติพิเศษต่างๆที่ช่วยเสริมความสามารถของเครื่องมืออีกด้วย



รูปที่ 3.15 ตัวอย่างโครงสร้างของโปรเจค (Java project) ที่ถูกสร้างโดย Eclipse

นอกจากนี้เครื่องมือที่สร้างขึ้นยังสามารถใช้ร่วมกับโปรแกรมประยุกต์ที่ใช้ในการพัฒนาโปรแกรมภาษาจาวา เช่น Eclipse และ NetBeans เนื่องจากโปรแกรมประยุกต์เหล่านี้มีการแยกไฟล์ตระกูล .class เอาไว้ในโฟลเดอร์ bin ดังเช่นตัวอย่างในรูปที่ 3.15 ซึ่งเครื่องมือที่สร้างขึ้นสามารถอาศัยข้อดีนี้เพื่อเข้าถึงไฟล์ตระกูล .class เพื่อใช้เป็นแหล่งข้อมูลนำเข้าได้ ดังที่จะนำเสนอขั้นตอนการทำงานของเครื่องมือตัวอย่างดังต่อไปนี้

3.1.6 ขั้นตอนการทำงานของเครื่องมือตรวจหาความผิดพลาดของภาษาจาวาโดยใช้ไบต์โค้ด

1. เตรียมข้อมูล

นำรูปแบบแพทเทิร์นที่ทำให้เกิดความผิดพลาดในภาษาจาวา และข้อมูลอื่นๆที่จำเป็นที่ได้จากการศึกษาจัดเก็บไว้ในฐานข้อมูลสำหรับตรวจสอบ โดยในงานวิจัยนี้จะเรียกฐานข้อมูลนี้ว่า ฐานข้อมูลตรวจสอบ (Check pattern database)

2. สร้างแพทเทิร์นจากแหล่งข้อมูลนำเข้า

2.1 นำเข้าข้อมูลไบต์โค้ดจากไฟล์ตระกูล .class

2.2 สร้างไบต์โค้ด และไลน์โค้ดจากคำสั่ง javap

2.3 สร้างแพทเทิร์นของไบต์โค้ดด้วยวิธีการแบ่งชุดคำสั่งโดยอาศัยไบต์โค้ด และไลน์โค้ดที่ได้ในขั้นตอนที่ 2

2.4 นำข้อมูลแพทเทิร์นที่ได้ทั้งหมดจัดเก็บลงในฐานข้อมูลซึ่งในงานวิจัยนี้จะเรียกฐานข้อมูลนี้ว่าฐานข้อมูลนำเข้า (Input pattern database)

3. ค้นหาแพทเทิร์นที่อาจก่อให้เกิดความผิดพลาด

3.1 เปรียบเทียบข้อมูลแพทเทิร์นในฐานข้อมูลนำเข้า ว่ามีรูปแบบของแพทเทิร์นที่อยู่ในฐานข้อมูลตรวจสอบประกอบอยู่ด้วยหรือเปล่า

3.2 หากพบแพทเทิร์นใดในฐานข้อมูลนำเข้ามีรูปแบบเหมือนกับแพทเทิร์นในฐานข้อมูลตรวจสอบให้เก็บหมายเลขอ้างอิงลงในตารางชั่วคราวเพื่อตรวจสอบต่อไป

4. ตรวจสอบเงื่อนไขการเกิดความผิดพลาด

นำข้อมูลในแต่ละฟิลด์ของตารางชั่วคราวที่เก็บข้อมูลแพทเทิร์นที่อาจก่อให้เกิดความผิดพลาดในโปรแกรมมาตรวจสอบเงื่อนไขที่กำหนดเอาไว้ ซึ่งหากแพทเทิร์นที่กำลังพิจารณาอยู่มีความถูกต้องตามเงื่อนไขทุกข้อแล้วแพทเทิร์นนั้นจะไม่ก่อให้เกิดความผิดพลาด แต่ถ้าหากไม่ถูกต้องตามเงื่อนไขใดเงื่อนไขหนึ่งแล้ว แพทเทิร์นนั้นจะเป็นแพทเทิร์นที่ก่อให้เกิดความผิดพลาดและถูกรายงานผลให้แก่ผู้ใช้ทราบ

มหาวิทยาลัยเทคโนโลยีสุรนารี

3.2 เครื่องมือที่ใช้ในการวิจัย

1. เครื่องคอมพิวเตอร์สำหรับพัฒนาเครื่องมือตัวอย่างในการตรวจหาความผิดพลาดขณะโปรแกรมประมวลผลในภาษาจาวา โดยเครื่องคอมพิวเตอร์ที่ใช้มีรายละเอียดดังนี้

- หน่วยประมวลผลกลาง : Intel®Core™ i3-380UM
- หน่วยความจำสำรอง : 2 GB Dual-channel 1333MHz DDR3 SDRAM
- หน่วยความจำหลัก : 500GB 5400RPM SATA Hard Drive
- อุปกรณ์เสริมอื่นๆ เช่น เมาส์ แป้นพิมพ์ เรื่องพิมพ์ เป็นต้น

2. ระบบปฏิบัติการและโปรแกรมประยุกต์สำหรับศึกษา และพัฒนาเครื่องมือ

- ระบบปฏิบัติการ : Windows 7 Ultimate 64 bit operating system
- เว็บบราวส์เซอร์ : Mozilla Firefox 6.0
- โปรแกรมจัดการข้อความ : Notepad ++ 5.9.2
- โปรแกรมมาตรฐานของภาษาจาวา : Java Development Kit (JDK) 6.0
- โปรแกรมพัฒนาโปรแกรมภาษาจาวา : Eclipse 3.6.1
- ส่วนเสริมความสารมารถโปรแกรม : Bytecode Outline 2.1.0
- ฐานข้อมูล : H2 1.3.159
- ระบบจัดการฐานข้อมูล : H2 Console

3.3 การเก็บรวบรวมข้อมูล

ในงานวิจัยนี้ผู้วิจัยได้เก็บรวมรวมข้อมูลเพื่อใช้ประกอบการสร้างวิธีการค้นหาวิธีการตรวจหาความผิดพลาดของโปรแกรมภาษาจาวาโดยใช้ไบต์โค้ดโดยมีแหล่งที่มาของข้อมูล 2 แหล่งหลักๆดังนี้

3.3.1 การเก็บรวบรวมข้อมูลจากสื่ออิเล็กทรอนิกส์บนอินเทอร์เน็ต

มีงานวิจัยของนักวิจัยท่านอื่นที่มีความเกี่ยวข้อง และเป็นประโยชน์กับงานวิจัยนี้ซึ่งผู้วิจัยได้นำเอางานวิจัยเหล่านั้นมาศึกษา โดยงานวิจัยที่ได้ศึกษาจะมีแหล่งที่มาจากเว็บไซต์ 2 เว็บไซต์ คือ

www.ieee.org และ www.acm.org โดยทั้งสองเว็บไซต์เป็นแหล่งเก็บรวบรวมงานวิจัยขนาดใหญ่ของโลก เนื่องจากการสร้างวิธีการค้นหาวิธีการตรวจหาความผิดพลาดของโปรแกรมภาษาจาวาโดยใช้ไบต์โค้ดต้องทำการศึกษาหารูปแบบของไบต์โค้ดที่จะทำให้เกิดความผิดพลาดซึ่งบนอินเทอร์เน็ต มีแหล่งข้อมูลเกี่ยวกับความผิดพลาดของภาษาจาวาที่ผู้วิจัยสามารถนำเอาความผิดพลาดบางตัวมาหารูปแบบของไบต์โค้ดที่จะทำให้เกิดความผิดพลาดขึ้นได้

3.3.2 การเก็บรวบรวมข้อมูลจากการทดลองด้วยโปรแกรมประยุกต์

ในงานวิจัยนี้ได้มีการใช้งานโปรแกรมประยุกต์ Eclipse และส่วนเสริม Bytecode Outline ซึ่งทำให้ผู้วิจัยสามารถสร้างซอสโค้ดพร้อมกับดูไบต์โค้ดได้พร้อมกันทำให้สามารถระบุได้ว่าไบต์โค้ดส่วนใดที่ก่อให้เกิดความผิดพลาดในโปรแกรม และ นำมาสร้างเป็นรูปแบบเพื่อใช้ในการตรวจหาต่อไปได้

3.4 การวิเคราะห์ข้อมูล

ในงานวิจัยนี้ข้อมูลที่ถือว่าเป็นหัวใจหลักของงานวิจัยคือไบต์โค้ด ซึ่งผู้วิจัยได้ทำการวิเคราะห์ไบต์โค้ดโดยมีลำดับดังนี้

3.4.1 วิเคราะห์ลักษณะคำสั่งของไบต์โค้ด ในขั้นตอนการวิเคราะห์นี้จะเป็นการสังเกตลักษณะคำสั่งของไบต์โค้ดว่าในหนึ่งบรรทัดของชุดคำสั่งประกอบไปด้วยอะไรบ้าง และ มีสัญลักษณ์อะไรในการแบ่งแยกคำสั่ง หรือ ตัวแปร

3.4.2 วิเคราะห์การแบ่งชุดคำสั่งของไบต์โค้ด ในขั้นตอนนี้จะเป็นการศึกษาและสังเกตว่าในไบต์โค้ดมีการแบ่งชุดคำสั่งอย่างไร ซึ่งพบว่า ต้องอาศัยคำสั่ง javap ในการสร้างข้อมูลการแบ่งบรรทัดคำสั่งเพื่อนำมาใช้แบ่งชุดคำสั่งของไบต์โค้ด

3.4.3 วิเคราะห์รูปแบบของไบต์โค้ดที่อาจก่อนให้เกิดความผิดพลาด โดยการนำเอารูปแบบที่ทำการแบ่งชุดคำสั่งเรียบร้อยแล้วมาหาว่ารูปแบบใดที่มีความสัมพันธ์กับการเกิดความผิดพลาดของโปรแกรมภาษาจาวา แล้วนำไปบันทึกไว้เพื่อใช้เป็นรูปแบบในการตรวจสอบความผิดพลาดต่อไป

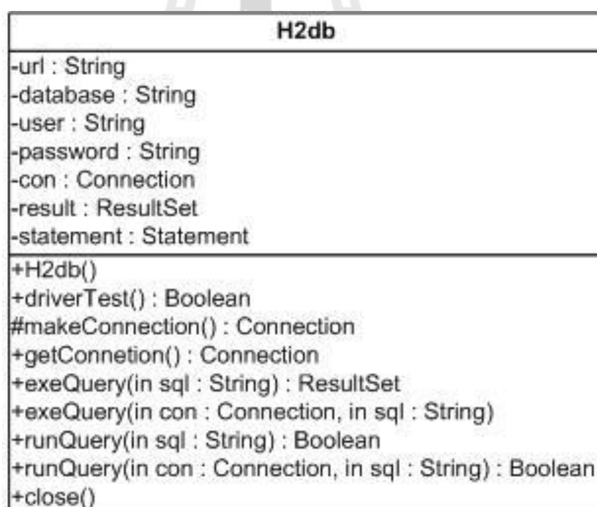
มหาวิทยาลัยเทคโนโลยีสุรนารี

3.5 การออกแบบวิธีการตรวจหาความผิดพลาดในภาษาจาวาโดยใช้ไบต์โค้ดเพื่อสร้างเครื่องมือตัวอย่าง

ในขั้นตอนนี้จะเป็นการออกแบบการทำงานของเครื่องมือเพื่อตรวจหาความผิดพลาดในภาษาจาวาด้วยไบต์โค้ด โดยแสดงโครงสร้างและวิธีการทำงานด้วยแผนภาพ UML เพื่อให้ผู้ศึกษาสามารถเข้าใจได้ง่ายซึ่งส่วนแรกจะแสดงคลาสไดอะแกรม (Class diagram) และอธิบายหน้าที่ของแต่ละคลาสดังต่อไปนี้

3.1.7 คลาส H2db

เป็นคลาสที่มีหน้าที่จัดการเกี่ยวกับการเชื่อมต่อกับฐานข้อมูล H2 โดยในคลาสนี้จะจัดเก็บชื่อผู้ใช้ รหัสผ่าน ชื่อฐานข้อมูล และรายละเอียดอื่นๆที่จำเป็นในการเชื่อต่อกับฐานข้อมูลโดยหากมีการเข้าถึงฐานข้อมูล เช่น แทรกข้อมูล แก้ไขข้อมูลหรือลบข้อมูลจะต้องดำเนินการผ่านคลาสนี้



รูปที่ 3.16 โครงสร้างของคลาส H2db

3.1.8 คลาส DBFunc

เป็นคลาสที่มีหน้าที่หลักสองอย่างคือ การจัดเก็บไบต์โค้ดที่จัดรูปแบบแล้วลงในฐานข้อมูล และการเทียบข้อมูลรูปแบบความผิดพลาดที่อ่านได้จากข้อมูลนำเข้ากับข้อมูลรูปแบบความผิดพลาดที่เก็บไว้ในฐานข้อมูลตรวจสอบ โดยการทำงานทั้งสองอย่างดำเนินการผ่านคลาส H2db

DBFunc
+h2 : H2db +result : LinkedList<ReportStruct> +temp : LinkedList<tempStruct>
+DBFunc() +screen() +addCode(in code : String, in id : String, in isExc : Integer, in haveCon : Boolean) +genSQL(in oldSql : String) : String +existPattern(in pattern : String) : Boolean +getPatternId(in pattern : String) : Boolean +getExceptionType(in id : String) : String +haveRule(in ruleId : String) : Boolean +getRule(in ruleId : String) : ArrayList<String>

รูปที่ 3.17 โครงสร้างของคลาส DBFunc

3.1.9 คลาส IFile

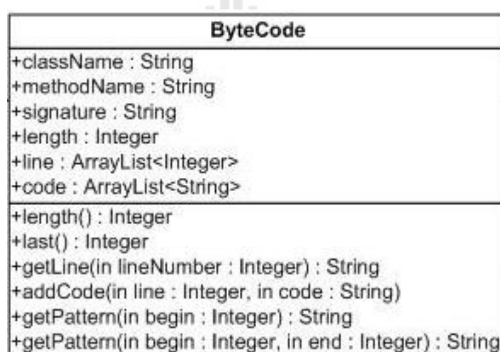
เป็นคลาสที่มีหน้าที่จัดการเกี่ยวกับไฟล์นามสกุล .class ที่นำมาเป็นข้อมูลนำเข้าโดยทำการแปลงข้อมูลที่จัดเก็บอยู่ในไฟล์ตระกูล .class ที่มีลักษณะเป็นไบต์โค้ดที่ไม่สามารถอ่านทำความเข้าใจได้ให้อยู่ในรูปแบบที่สามารถอ่านทำความเข้าใจได้โดยใช้เครื่องมือ javap ที่เป็นเครื่องมือพื้นฐานของเวอร์ชวลแมชีนอยู่แล้ว จากนั้นอ่านข้อมูลผลลัพธ์ที่ได้ในลักษณะของข้อความแล้วนำมาจัดแบ่งเก็บในลักษณะของโครงสร้างข้อมูลแบบอาร์เรย์ลิสต์เพื่อเตรียมสำหรับการเปรียบเทียบรูปแบบต่อไป

IFile
+line : ArrayList<String> -count : Integer
+IFile(in path : String) +IFile(in path : String, in type : String) +readFile(in path : String) +readFile(in path : String, in type : String) +length() : Integer

รูปที่ 3.18 โครงสร้างของคลาส IFile

3.1.10 คลาส ByteCode

เป็นคลาสที่มีหน้าที่จัดรูปแบบข้อมูลที่ได้จากคลาส IFile ให้อยู่ในรูปแบบของไบต์โค้ดที่มีการแบ่งออกเป็นชุดคำสั่งโดยอาศัยไลน์โค้ดในการแบ่ง โดยเมื่อทำการแบ่งชุดคำสั่งแล้วจะทำการจัดเก็บชุดคำสั่งที่กำกับด้วยหมายเลขลำดับบรรทัดของชุดคำสั่งอย่างชัดเจนเพื่อใช้อ้างอิงในการเปรียบเทียบรูปแบบไบต์โค้ด และนอกจากนี้ในคลาสนี้จะทำการเก็บข้อมูลที่สำคัญต่างๆ เช่น ชื่อคลาส ชื่อเมท็อด และ Signature ของเมท็อด เป็นต้น เพื่อใช้ในการระบุถึงชุดคำสั่งในตำแหน่งที่ถูกต้องโดยหนึ่งวัตถุ (Instance) ของคลาสนี้จะเทียบได้กับ 1 ชุดคำสั่งของไบต์โค้ด

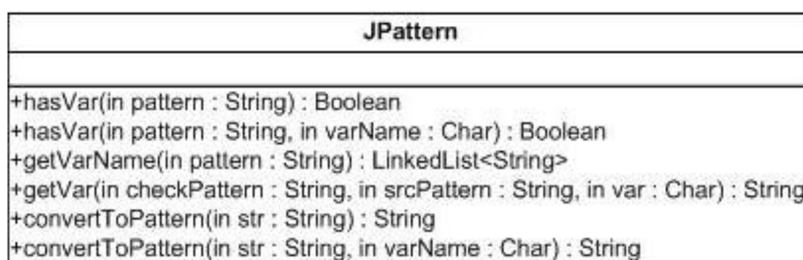


รูปที่ 3.19 โครงสร้างของคลาส ByteCode

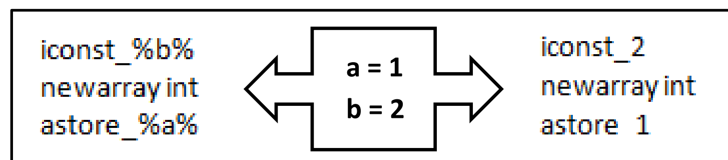
าวิทยาลัยเทคโนโลยีสุร

3.1.11 คลาส JPattern

เป็นคลาสที่มีหน้าที่จัดการเกี่ยวกับการค้นหาค่าของตัวแปรจากแพทเทิร์นหลังจากที่ทำการเปรียบเทียบรูปแบบของไบต์โค้ดแล้ว



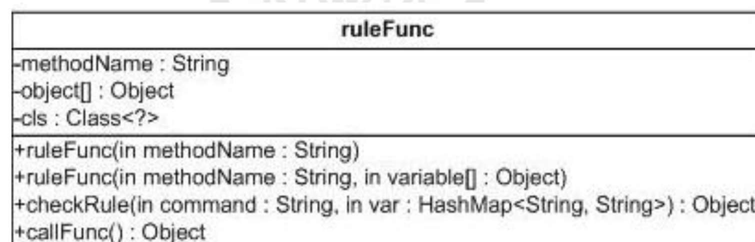
รูปที่ 3.20 โครงสร้างของคลาส JPattern



รูปที่ 3.21 ลักษณะการทำงานของคลาส JPattern

3.1.12 คลาส ruleFunc

เป็นคลาสที่มีหน้าที่ทดสอบความถูกต้องของเงื่อนไขการเกิดความผิดพลาดโดยในคลาสนี้จะมีการนำเอาค่าของตัวแปรที่ได้จากคลาส JPattern มาดำเนินการเพื่อทดสอบว่าเป็นไปตามกฎหรือไม่ โดยกฎของความผิดพลาดคือเงื่อนไขที่ใช้เพื่อตรวจสอบว่าแพทเทิร์นที่กำลังพิจารณาเป็นแพทเทิร์นที่ทำให้เกิดความผิดพลาดหรือไม่ ซึ่งกฎมีลักษณะเป็นการเปรียบเทียบหรือดำเนินการทางคณิตศาสตร์ เช่น มากกว่า น้อยกว่า หรือ เป็นตัวเลข เป็นต้น

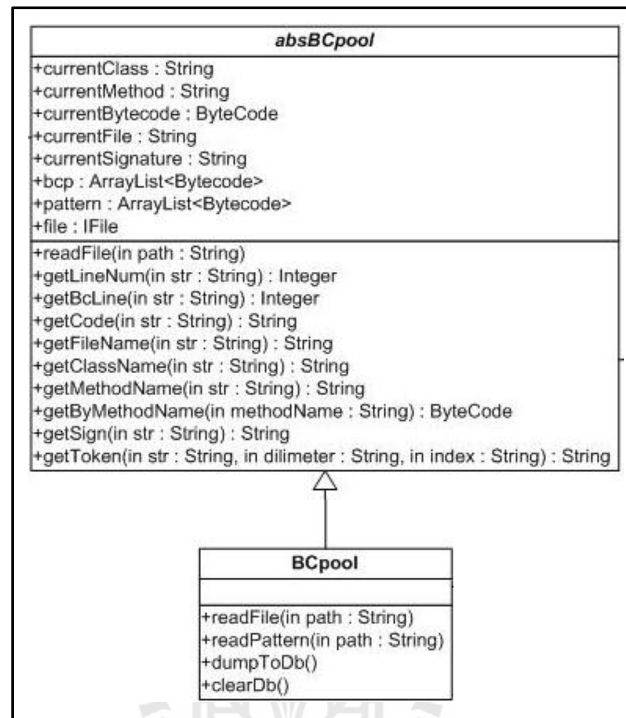


รูปที่ 3.22 โครงสร้างของคลาส ruleFunc

3.1.13 คลาส BCpool

เป็นคลาสหลักที่รวบรวมการทำงานหลักๆของการตรวจสอบความผิดพลาดด้วยไบต์โค้ดเอาไว้ โดยคลาส BCpool จะถ่ายทอดมาจากคลาส asbBCpool ซึ่งเป็นคลาสที่เก็บข้อมูลไบต์โค้ดที่ได้จากการนำเข้าเอาไว้ในตัวแปรชื่อ bcp และมีเมท็อดการทำงานปลีกย่อยในการค้นหาความผิดพลาด เพื่อให้ง่ายและรวบรัดขั้นตอนการทำงานผู้วิจัยจึงให้คลาส BCpool ถ่ายทอดมาจากคลาส asbBCpool เพื่อให้คลาส BCpool รวบรวมการขั้นตอนการทำงานให้เป็นคำสั่งที่ใช้งานง่ายเพื่อความ

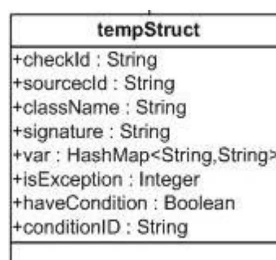
สะดวกในการเรียกใช้ โดยอาจกล่าวได้ว่าคลาส BCpool นี้เป็นจุดเริ่มต้นในการทำงานของเครื่องมือตรวจหาความผิดพลาดโดยใช้ไบต์โค้ด



รูปที่ 3.23 โครงสร้างของคลาส BCpool และ asbBCpool

3.1.14 คลาส tempStruct

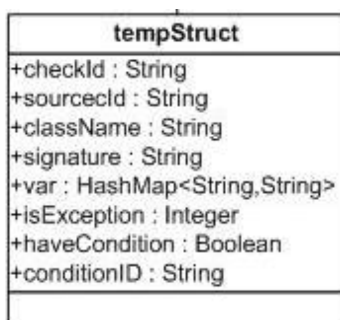
เป็นคลาสที่มีหน้าที่เป็นโครงสร้างข้อมูลที่ใช้เก็บชุกคำสั่งไบต์โค้ดและรายละเอียดของชุดคำสั่งไบต์โค้ดเอาไว้ชั่วคราวเพื่อใช้ในการนำเข้าไปประมวลผลเปรียบเทียบกับฐานข้อมูลตรวจ



รูปที่ 3.24 โครงสร้างของคลาส tempStruct

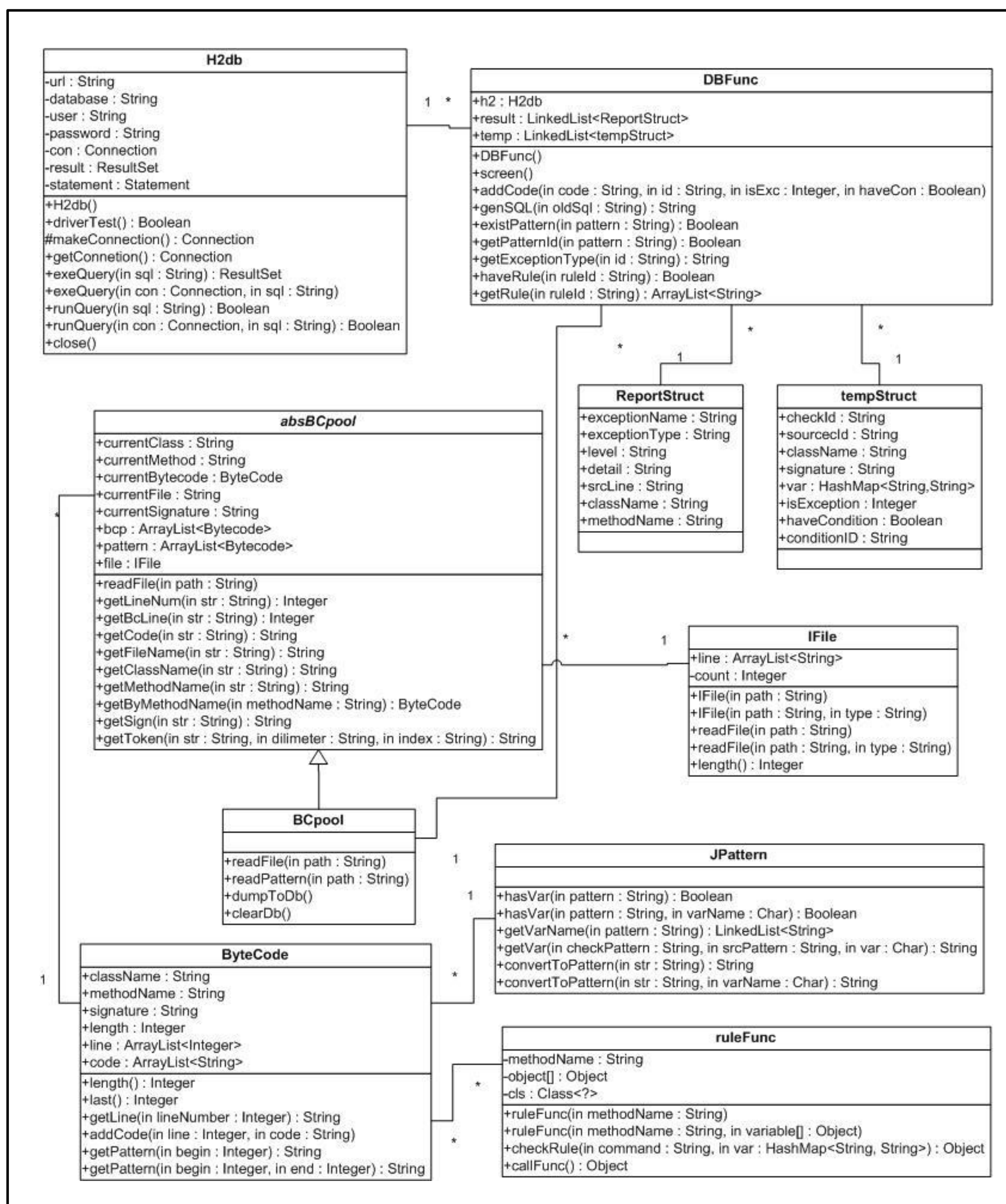
3.1.15 คลาส ReportStruct

เป็นคลาสที่มีหน้าที่เป็นโครงสร้างข้อมูลที่ใช้จัดเก็บผลลัพธ์จากการประมวลผลเพื่อใช้ในการแสดงข้อมูลให้ผู้ใช้งานทราบ โดยมีการจัดเก็บข้อมูลตำแหน่งบรรทัดที่เกิดความผิดพลาดขึ้นในซอสโค้ด ประเภทของความผิดพลาด และข้อมูลที่สำคัญอื่นๆ



รูปที่ 3.25โครงสร้างของคลาส tempStruct

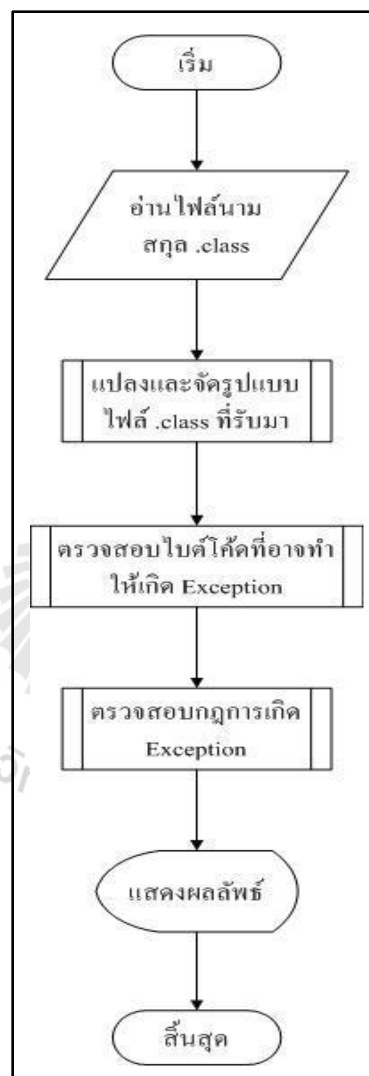
ในงานวิจัยนี้ผู้วิจัยได้ออกแบบการทำงานของเครื่องมือให้มีการแยกการทำงานออกเป็นส่วนๆ เพื่อให้ผู้ศึกษาสามารถทำความเข้าใจได้ง่าย อีกทั้งในบางกรณีที่มีผู้ศึกษาต้องการนำเอาบางส่วนการทำงานของเครื่องมือไปประยุกต์ใช้ก็สามารถทำได้โดยการเลือกเอาเฉพาะส่วนที่เกี่ยวข้องไปใช้งาน ซึ่งเมื่อนำเอาคลาสทั้งหมดมาใช้งานร่วมกันจะสามารถแสดงความสัมพันธ์ได้ดังคลาสไดอะแกรมดังต่อไปนี้



รูปที่ 3.26 คลาสไดอะแกรมของเครื่องมือตรวจหาความผิดพลาดของภาษาจาวาด้วยไบต์โค้ด

3.6 การทำงานของเครื่องมือตรวจหาความผิดพลาดในภาษาจาวาด้วยไบต์โค้ด

เนื้อหาส่วนนี้จะแสดงลำดับขั้นตอนการประมวลผลของเครื่องมือตรวจหาความผิดพลาดในภาษาจาวาด้วยไบต์โค้ด โดยใช้แผนภาพผังงาน (Flowchart) ในการแสดงการทำงานซึ่งลักษณะการทำงานคร่าวๆของเครื่องมือนี้สามารถแสดงได้ดังรูปที่ 3.27



รูปที่ 3.27 ผังงานแสดงการทำงานโดยรวมของเครื่องมือตรวจหาหาความผิดพลาดในภาษาจาวาด้วยไบต์โค้ด

จากรูปที่ 3.27 จะเห็นว่าการทำงานของเครื่องมือจะเริ่มต้นจากการนำเอาไฟล์นามสกุล .class มาเป็นข้อมูลนำเข้าของเครื่องมือ ต่อมาจะมีการแปลงและจัดรูปแบบไฟล์ดังกล่าวให้อยู่ในรูปไบต์โค้ดที่สามารถอ่านทำความเข้าใจได้ ลำดับถัดมาเครื่องมือจะทำการตรวจหารูปแบบของไบต์โค้ดที่อาจทำให้เกิดความผิดพลาดโดยในขั้นตอนนี้จะทำการเทียบรูปแบบของไบต์โค้ดที่ได้จากข้อมูลนำเข้ากับฐานข้อมูลตรวจสอบที่ได้ทำการเตรียมไว้ ต่อมาก็จะทำการตรวจสอบกฎการเกิดความผิดพลาด โดยหากเป็นไปตามกฎทั้งหมดแสดงว่าชุดคำสั่งไบต์โค้ดชุดนั้นไม่ก่อให้เกิดความผิดพลาด แต่หากมีการละเมิดกฎข้อใดข้อหนึ่งนั่นแสดงว่าชุดคำสั่งไบต์โค้ดชุดนั้นก่อให้เกิดความผิดพลาดนั่นเอง

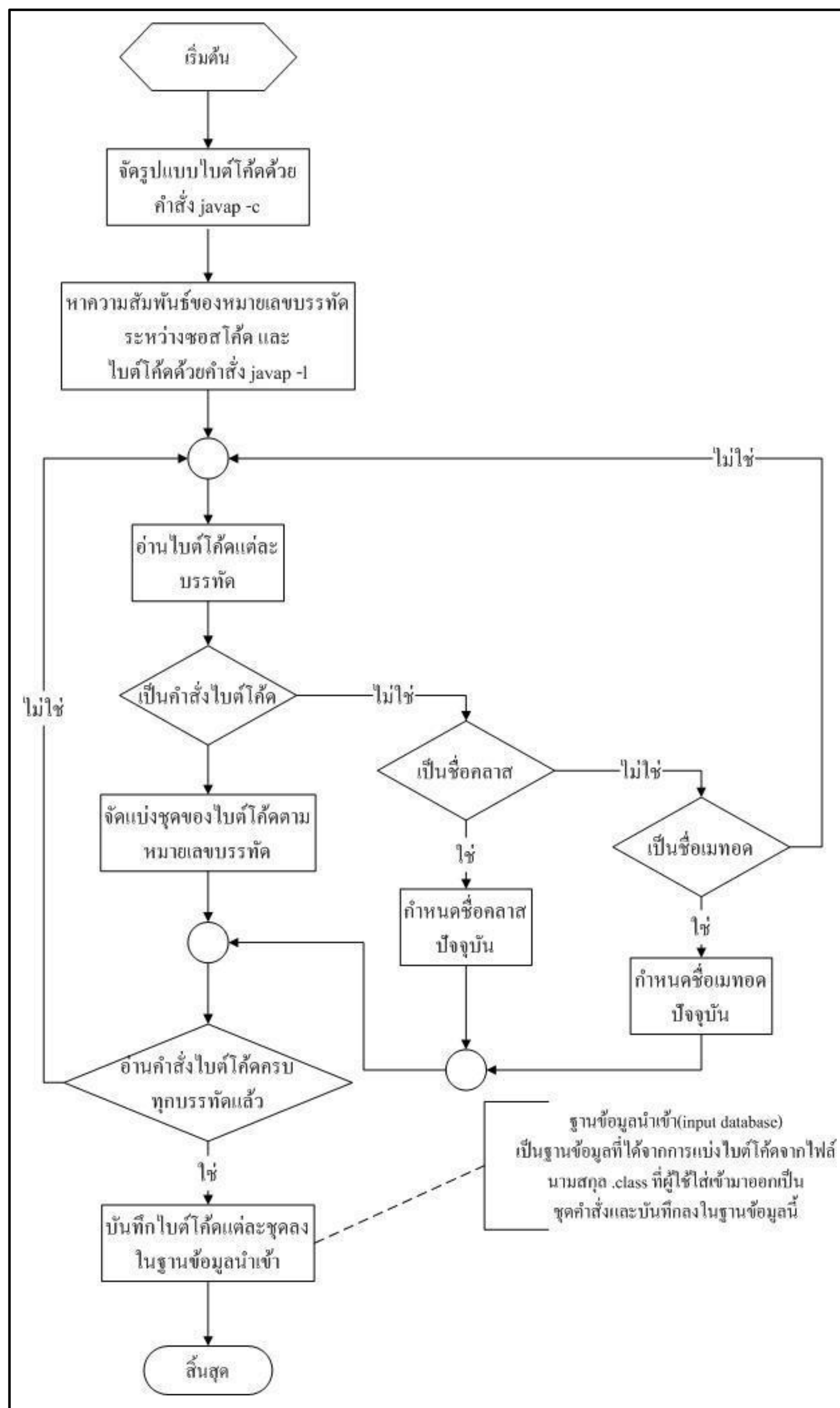
ในรูปที่ 3.27 เป็นเพียงการทำงานคร่าวๆของเครื่องมือ ซึ่งจะเห็นว่าจะมีบางส่วนที่แทนด้วยสัญลักษณ์การประมวลผลย่อย (Predefined process) โดยในแต่ละการประมวลผลย่อยจะมีรายละเอียดการทำงานภายใน ซึ่งในที่นี่ประกอบด้วยสามส่วนย่อย ดังนี้

1. ขั้นตอนการแปลงและจัดรูปแบบไฟล์นามสกุล .class (รูปที่ 3.28) เริ่มจากการนำเอาข้อมูลที่ได้จากไฟล์นามสกุล .class มาแปลงโดยเครื่องมือ javap ที่มีมาพร้อมกับเวอร์ชวลแมชีน เพื่อให้อยู่ในรูปแบบที่สามารถอ่านทำความเข้าใจได้พร้อมกับใช้ตัวเลือกในการแสดงหมายเลยบรรทัดชุดคำสั่งเพื่อนำมาหาความสัมพันธ์และแยกชุดคำสั่ง หลังจากนั้นทำการพิจารณาไบต์โค้ดแต่ละบรรทัดเพื่อแยกองค์ประกอบ โดยหากเป็นไบต์โค้ดที่ไม่ใช่ชุดคำสั่งจะถูกนำมาพิจารณาหาข้อมูลที่สำคัญ เช่น ชื่อคลาส ชื่อเมท็อด และSignature เป็นต้น และหากเป็นชุดคำสั่งไบต์โค้ดจะถูกนำมาแบ่งโดยอาศัยความสัมพันธ์ที่ได้จากการแสดงหมายเลขชุดคำสั่งที่ได้กระทำไปก่อนหน้านี้ จากนั้นเมื่อทำการพิจารณาครบทุกคำสั่งแล้วข้อมูลที่ได้จะถูกนำไปบันทึกในฐานข้อมูลนำเข้า

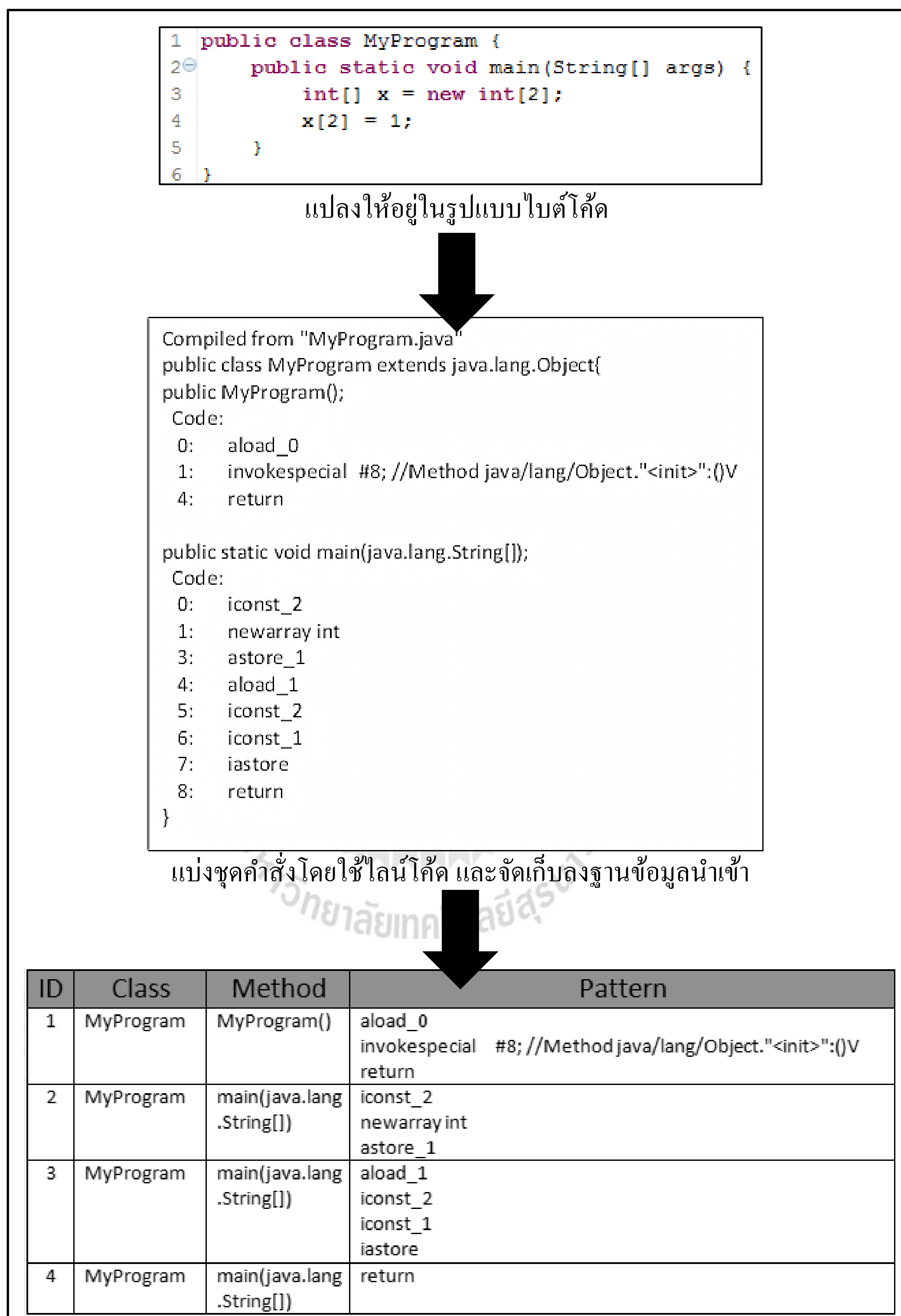
2. ขั้นตอนการตรวจหาไบต์โค้ดที่อาจทำให้เกิดความผิดพลาด (รูปที่ 3.30) เริ่มจากการนำข้อมูลที่ถูกบันทึกไว้ในฐานข้อมูลนำเข้ามาค้นหาเปรียบเทียบกับข้อมูลในฐานข้อมูลตรวจสอบซึ่งในงานวิจัยนี้ได้อาศัยคำสั่งจากภาษา SQL ในการค้นหาความเปรียบเทียบรูปแบบ ถ้าหากชุดคำสั่งไบต์โค้ดใดจากฐานข้อมูลนำเข้าตรงกับชุดคำสั่งในฐานข้อมูลตรวจสอบเครื่องมือจะจัดเก็บชุดคำสั่งนั้น โดยกำหนดสถานะเป็นต้องทำการวิเคราะห์เพื่อเข้าไปตรวจสอบในขั้นตอนตรวจสอบกฎการเกิดความผิดพลาดในขั้นตอนถัดไป

3. ขั้นตอนการตรวจสอบกฎการเกิดความผิดพลาด (รูปที่ 3.32) เริ่มจากนำเอาข้อมูลที่มีสถานะเป็นต้องตรวจสอบมาตรวจสอบกฎการเกิดความผิดพลาดที่ระบุไว้ในฐานข้อมูลตรวจสอบ ถ้าหากมีการละเมิดกฎข้อได้ข้อหนึ่งจะทำให้ชุดคำสั่งไบต์โค้ดที่กำลังพิจารณาถูกกำหนดสถานะเป็นไบต์โค้ดที่ทำให้เกิดความผิดพลาดทันที

1. ขั้นตอนการแปลงและจัดรูปแบบไฟล์นามกุล .class



รูปที่ 3.28 ขั้นตอนการแปลงและจัดรูปแบบไฟล์นามสกุล .class



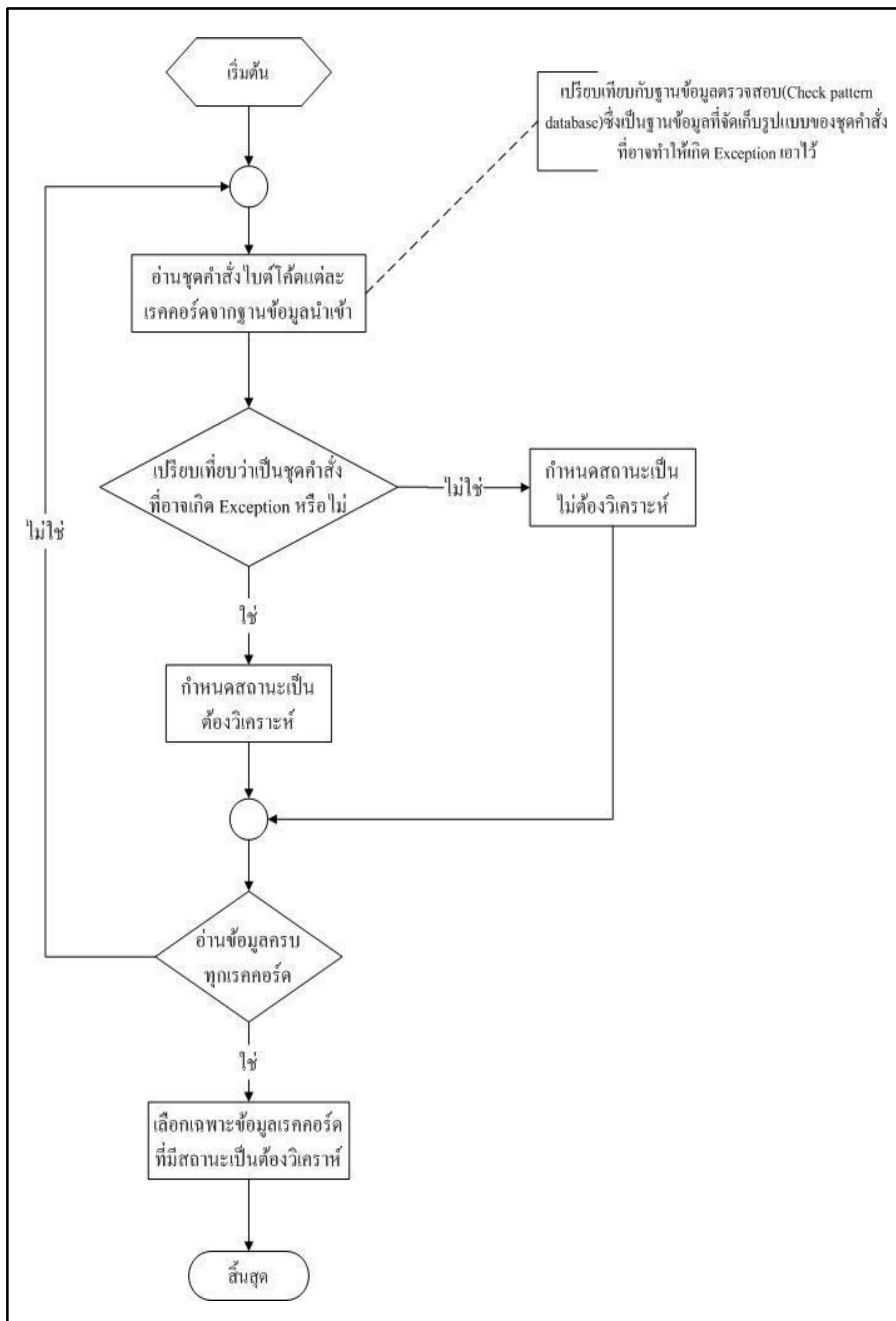
ID	Class	Method	Pattern

1	MyProgram	MyProgram()	aload_0
			invokespecial #8; //Method java/lang/Object."<init>":()V
			return

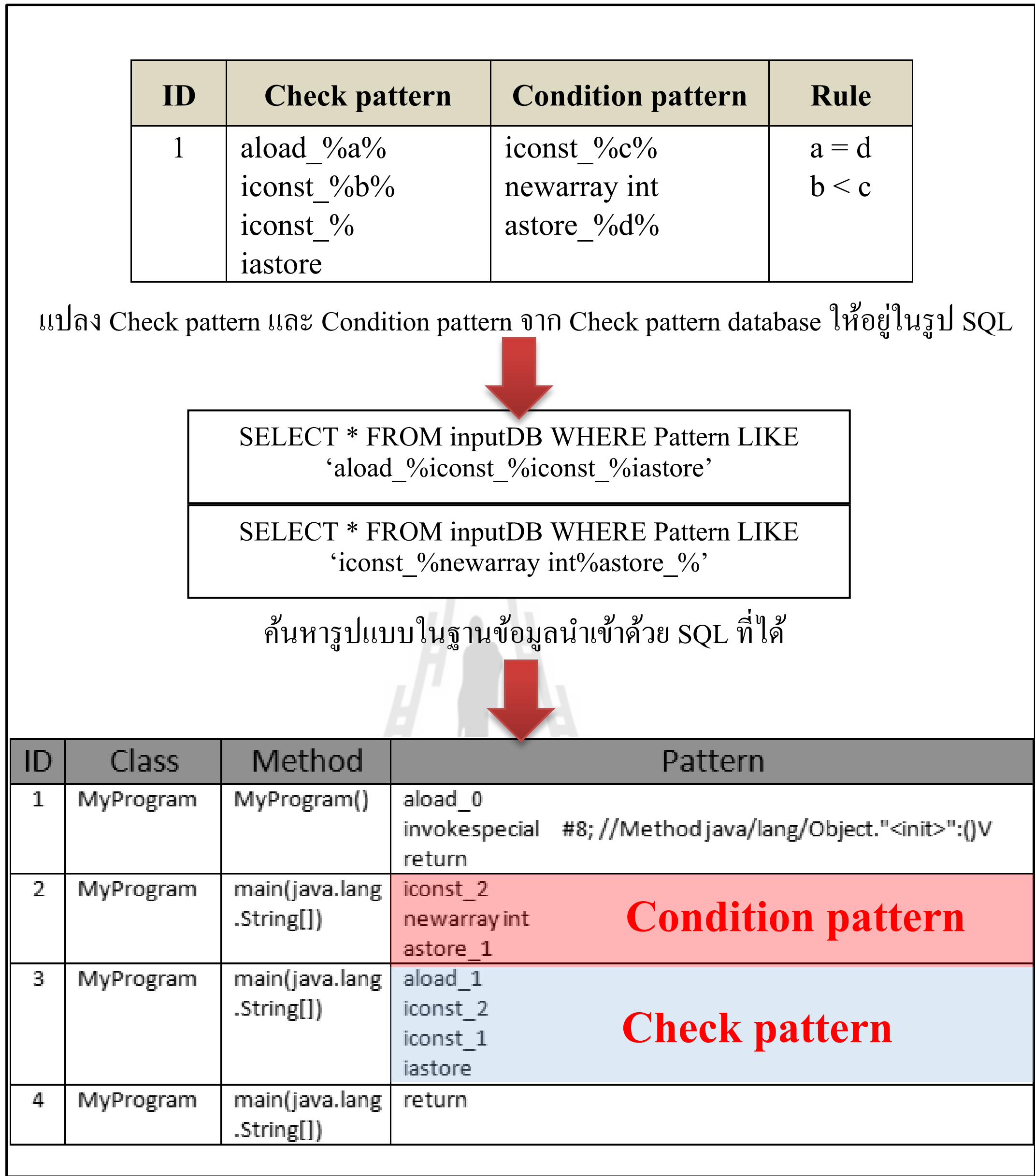
2	MyProgram	main(java.lang.String[])	iconst_2 newarray int astore_1
3	MyProgram	main(java.lang.String[])	aload_1 iconst_2 iconst_1 iastore
4	MyProgram	main(java.lang.String[])	return

รูปที่ 3.29 ตัวอย่างการทำงานและลักษณะข้อมูลที่ได้จากขั้นตอนการแปลงและจัดรูปแบบไฟล์นามสกุล .class

2. ขั้นตอนการตรวจหาไบต์โค้ดที่อาจทำให้เกิดความผิดพลาด

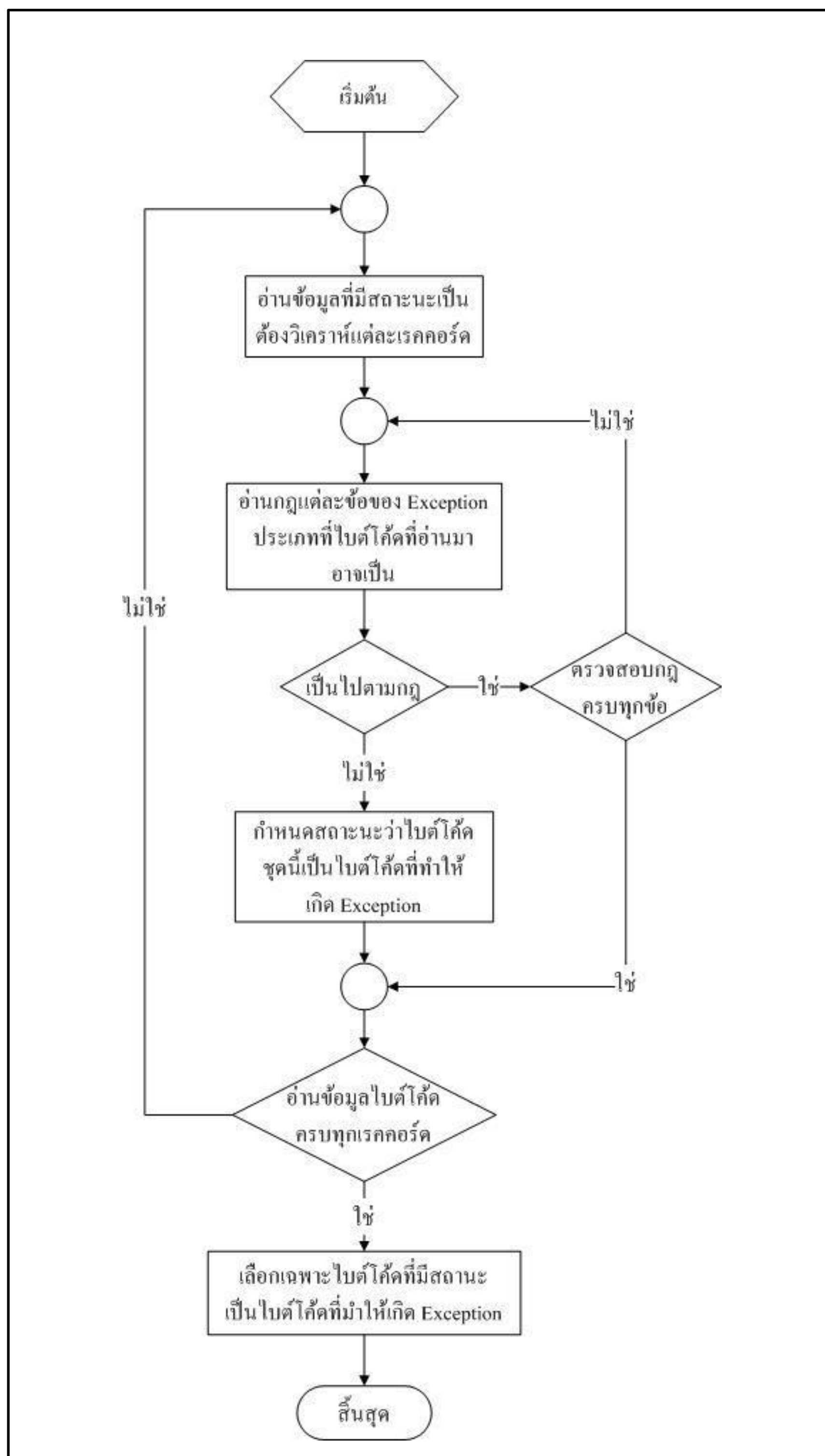


รูปที่ 3.30 ขั้นตอนการตรวจหาไบต์โค้ดที่อาจทำให้เกิดความผิดพลาด

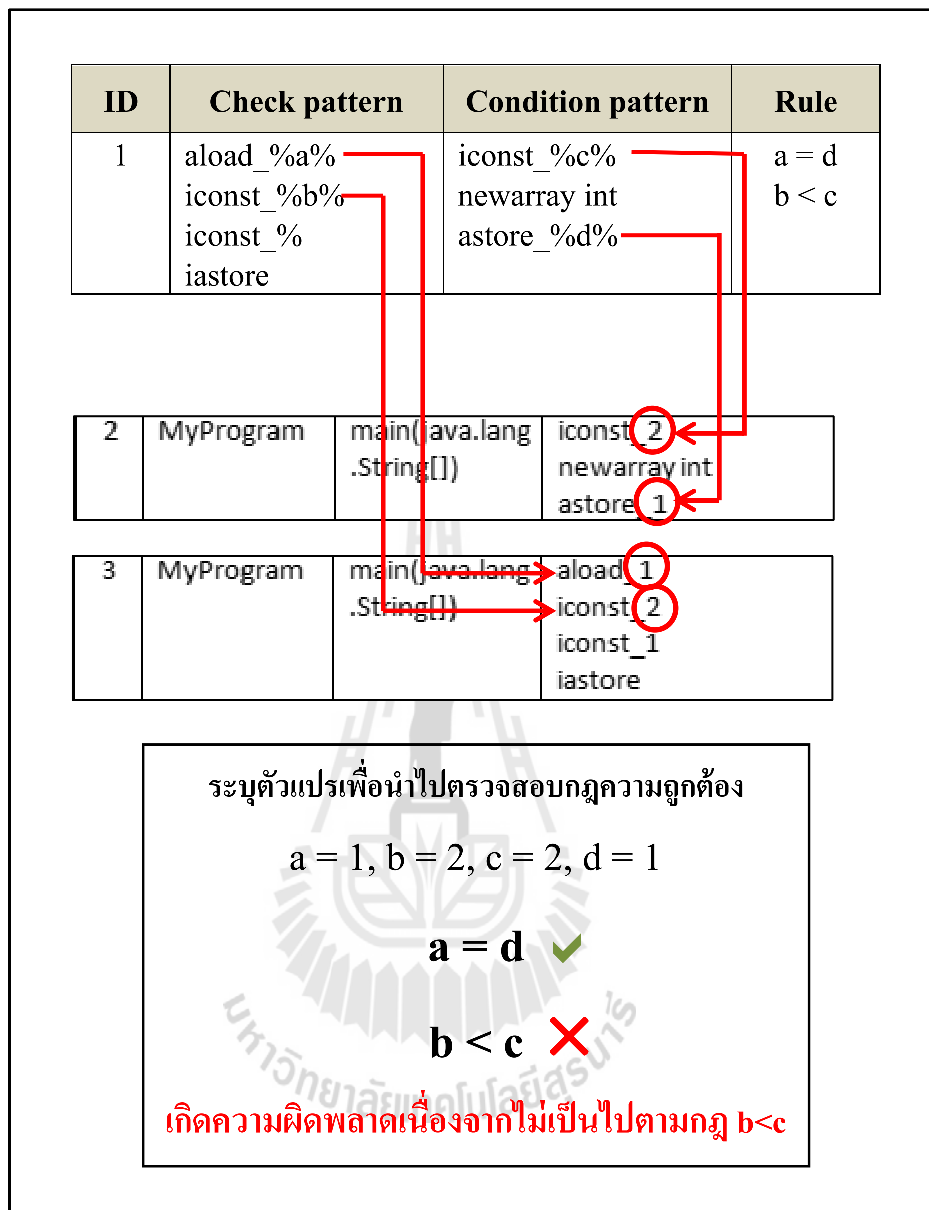


รูปที่ 3.31 ตัวอย่างการทำงานและลักษณะข้อมูลที่ได้จากขั้นตอนการตรวจหาไบต์โค้ดที่อาจทำให้เกิดความผิดพลาด

3. ขั้นตอนการตรวจสอบกฎการเกิดความผิดพลาด



รูปที่ 3.32 ขั้นตอนการตรวจสอบกฎการเกิดความผิดพลาด

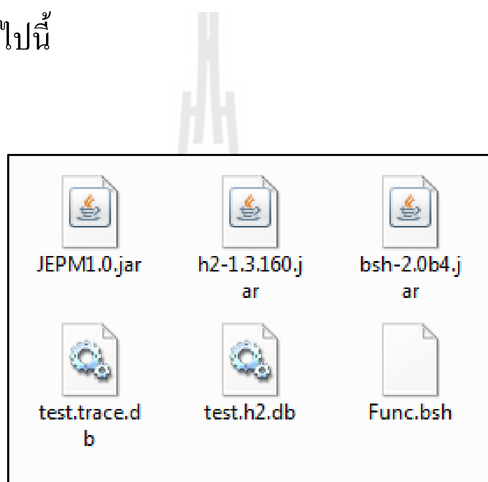


รูปที่ 3.33 ตัวอย่างการทำงานและลักษณะข้อมูลที่ได้จากขั้นตอนการตรวจสอบกฎการเกิดความผิดพลาด

วิธีการที่ผู้วิจัยได้นำเสนอนี้เป็นวิธีการที่ใช้กับไบต์โค้ดในภาษาจาวาซึ่งมีขั้นตอนการทำงานดังที่แสดงในผังงาน โดยหากนักวิจัยท่านอื่นสนใจสามารถประยุกต์ใช้วิธีการเดียวกันนี้กับข้อมูลรูปแบบอื่นๆได้ เช่น ข้อมูลไบนารีโค้ด เป็นต้น นอกจากนี้ผู้วิจัยยังออกแบบการทำงานย่อยต่างๆออกเป็นคลาสอย่างชัดเจน ซึ่งผู้สนใจสามารถนำเอาส่วนที่สนใจไปประยุกต์ใช้ต่อไปได้

3.7 เครื่องมือตัวอย่างและการนำไปใช้งาน

หัวข้อนี้จะกล่าวถึงเครื่องมือตัวอย่างที่ผู้วิจัยได้พัฒนาโดยใช้โครงสร้างและการทำงานดังที่ได้ออกแบบเอาไว้ในหัวข้อที่ 3.6 ซึ่งผู้ใช้ได้ตั้งชื่อเครื่องมือตัวอย่างนี้ว่า JEPM (Java Exception checking by Pattern Matching) โดยผลิตภัณฑ์ที่ได้จะอยู่ในรูปแบบไฟล์นามสกุล .jar ซึ่งในขณะนี้มีการเผยแพร่เครื่องมือตัวอย่างเวอร์ชัน 1.0 โดยส่วนประกอบของเครื่องมือจะประกอบไปด้วยไฟล์คลังโปรแกรมจำนวน 3 ไฟล์ (JEPM1.0.jar,bsh-2.0b4.jar,h2-1.3.160.jar) และไฟล์ฐานข้อมูลและไฟล์เก็บฟังก์ชันที่เรียกใช้งานอีกจำนวน 3 (test.h2.db,test.trace.db,Func.bsh) ดังแสดงในรูปที่ 3.34 ซึ่งแต่ละไฟล์มีหน้าที่ดังต่อไปนี้



รูปที่ 3.34 ส่วนประกอบของเครื่องมือตัวอย่าง JEPM1.0

1. bsh-2.0b4.jar เป็นคลังโปรแกรมของ BeanShell ซึ่งจะเป็นส่วนที่ช่วยอำนวยความสะดวกในการจัดการเกี่ยวกับการตรวจสอบกฎความถูกต้องของเครื่องมือตัวอย่าง

2. h2-1.3.160.jar เป็นคลังโปรแกรมของฐานข้อมูล H2 ซึ่งจะช่วยอำนวยความสะดวกในการติดต่อกับฐานข้อมูลเพื่อจัดการและจัดเก็บข้อมูลต่างๆที่ได้จากเครื่องมือตัวอย่าง

3. JEPM1.0.jar เป็นไฟล์หลักของเครื่องมือตัวอย่างซึ่งจะจัดการเกี่ยวกับการค้นหาความผิดพลาดและการแสดงผลโดยจะใช้คลังโปรแกรมทั้งสองตัวที่กล่าวมาก่อนหน้านี้ช่วยในการประมวลผล

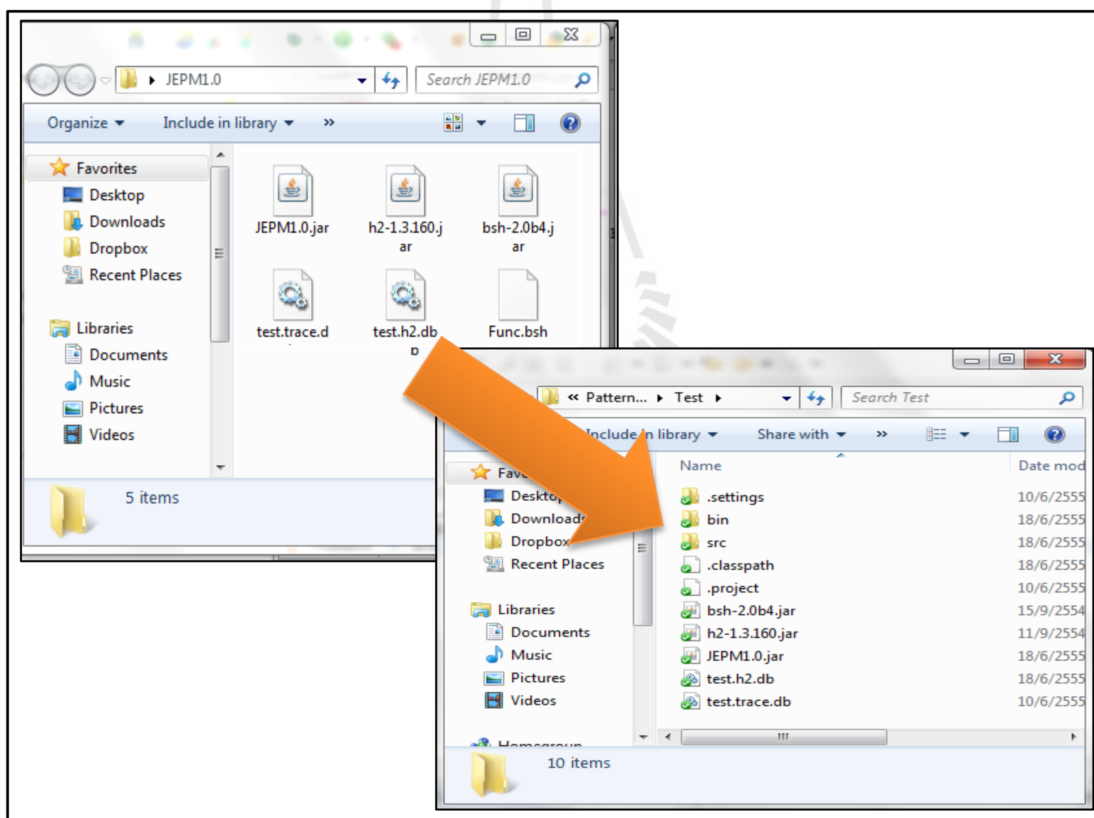
4. test.h2.db เป็นไฟล์ฐานข้อมูลที่ใช้ร่วมกับคลังโปรแกรมของฐานข้อมูล H2 โดยภายในจะจัดเก็บข้อมูลต่างๆที่ได้จากการประมวลผลของเครื่องมือตัวอย่างเอาไว้

5. test.trace.db เป็นไฟล์ที่จัดเก็บประวัติการทำงานของฐานข้อมูล H2 ซึ่งจะต้องปรากฏควบคู่กับไฟล์ test.h2.db เสมอ

6. Func.bsh เป็นไฟล์ที่จัดเก็บฟังก์ชันการทำงานที่ Beanshell จะนำไปใช้ในการตรวจสอบกฎความถูกต้อง ซึ่งสามารถแก้ไข หรือ เพิ่มฟังก์ชันใหม่ได้

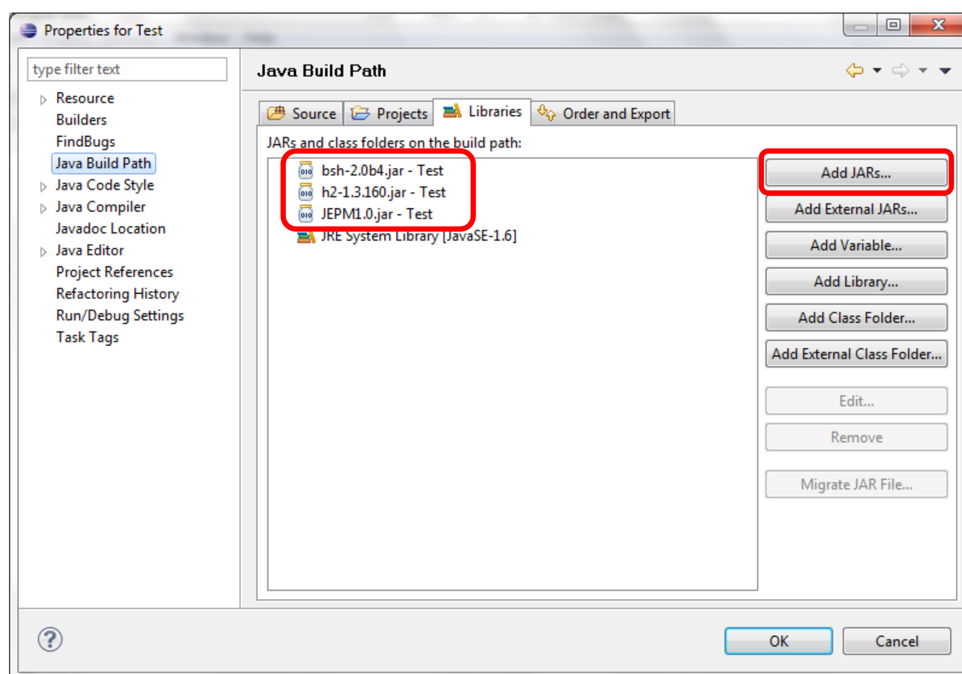
- **การนำเอาเครื่องมือตัวอย่างไปใช้ในลักษณะของคลังโปรแกรม** (Library)

1. คัดลอกไฟล์เครื่องมือตัวอย่างทั้งหมดไปไว้ในโฟลเดอร์ของโปรเจคจาวาที่ต้องการทดสอบ



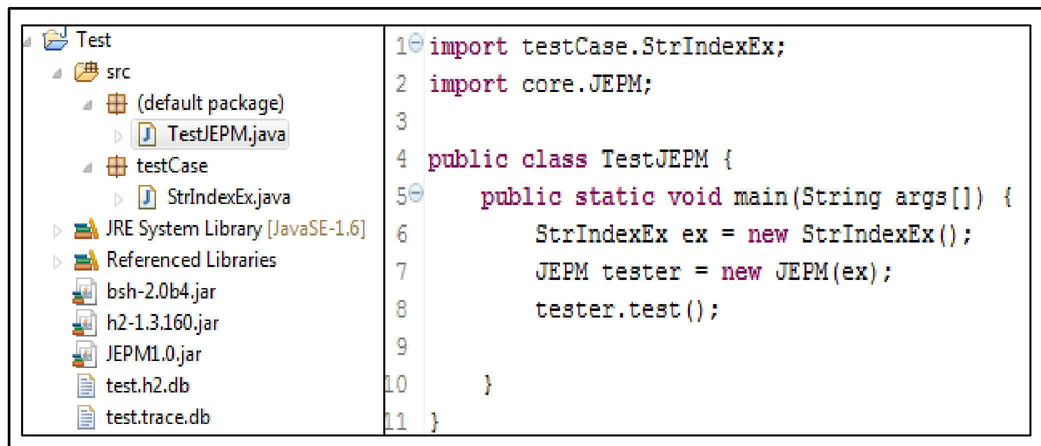
รูปที่ 3.35 การคัดลอกไฟล์ของเครื่องมือตัวอย่าง

2. เปิดโปรเจคจาวาขึ้นมาโดยในที่นี้ใช้โปรแกรม Eclipse ในการเปิดโปรเจคจากนั้นคลิกที่เมนู Project เลือก Properties จะปรากฏหน้าต่างขึ้นมาที่เมนูด้านซ้ายมือเลือก Java Build Path จากนั้นคลิกที่แทบ Libraries กดปุ่ม Add JARs… ทำการเพิ่มคลังโปรแกรมทั้งสามตัวของเครื่องมือตัวอย่างลงในโปรเจคจากนั้นกดปุ่ม OK



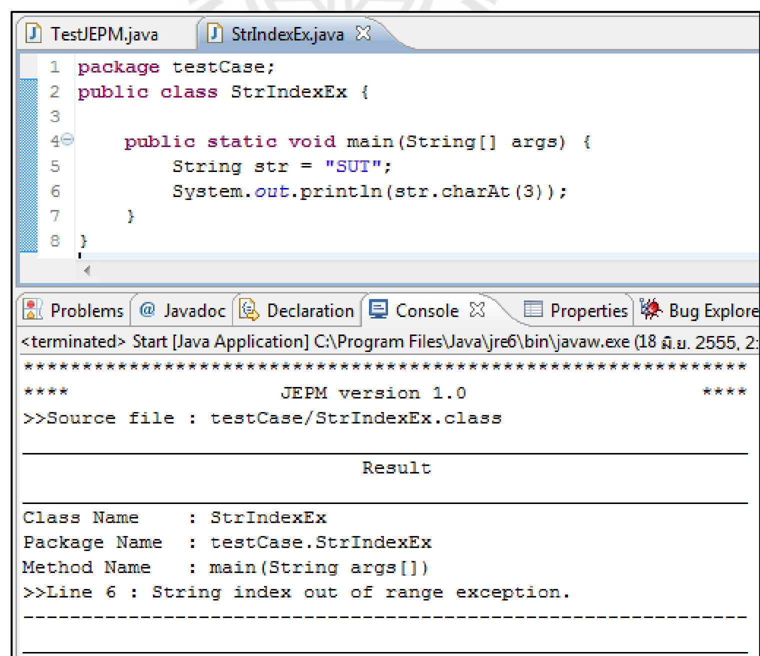
รูปที่ 3.36 การนำเข้าคลังโปรแกรมของเครื่องมือตัวอย่าง

3. สร้างคลาสไฟล์เพื่อใช้เป็นจุดเริ่มต้นในการทดสอบตัวอย่าง เช่นในตัวอย่างนี้จะทำการทดสอบคลาสชื่อ StrIndexEx (มีการเกิดความผิดพลาดประเภทการเรียกใช้สตริงเกินขอบเขต) จะต้องทำการเขียนซอสโค้ด (ดังที่แสดงในรูปที่ 3.37) โดยเริ่มจากการสร้างวัตถุของคลาสที่ต้องการทดสอบในที่นี้คือคลาส StrIndexEx (บรรทัดที่ 6) จากนั้นทำการสร้างวัตถุของเครื่องมือทดสอบพร้อมทั้งใส่พารามิเตอร์เป็นตัวแปรที่เก็บวัตถุของคลาสที่ต้องการทดสอบเอาไว้ (บรรทัดที่ 7) จากนั้นเรียกใช้เมท็อด test เพื่อเริ่มต้นการทดสอบ (บรรทัดที่ 8)



รูปที่ 3.37 การเขียนซอสโค้ดในเพื่อเรียกใช้เครื่องมือตัวอย่าง

4. จากนั้นทำการรันจะปรากฏผลลัพธ์ในหน้าต่าง Console ซึ่งในตัวอย่างนี้มีการแจ้งเตือนว่าบรรทัดที่ 6 เกิดความผิดพลาดประเภทที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด (String index out of range) ดังที่แสดงในรูปที่ 3.38



รูปที่ 3.38 ผลลัพธ์จากการทดสอบเครื่องมือในลักษณะคลังโปรแกรม

บทที่ 4
ผลการวิเคราะห์ข้อมูลและอภิปรายผล

จุดประสงค์ของการทดลองในการวิจัยครั้งนี้ คือการทดสอบและวัดความถูกต้องของวิธีการที่นำมาสร้างเป็นเครื่องมือตัวอย่าง โดยหยิบยกเอาเพียงตัวอย่างความผิดพลาดที่พบเห็นได้บ่อยในการเขียนโปรแกรมภาษาจาวาบางชนิดมาแสดงเท่านั้น ซึ่งตัวอย่างที่หยิบยกมานี้เป็นเพียงบางส่วนไม่ได้ครอบคลุมถึงความผิดพลาดทุกอย่างที่เกิดขึ้นในแต่ละชนิดของความผิดพลาด

การทดลองในการวิจัยครั้งนี้ได้แบ่งเป็นสองขั้นตอนคือขั้นตอน การเก็บรวบตัวอย่างความผิดพลาดประเภทที่สนใจเพื่อใช้ในการทดสอบเครื่องมือตัวอย่างที่สร้างขึ้น และขั้นตอนการทดสอบเครื่องมือตัวอย่างจริงโดยในที่นี้ได้เลือกประเภทของความผิดพลาดที่อาจเกิดขึ้นได้ในภาษาจาวามา 5 ประเภท คือ ความผิดพลาดที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง (Class cast exception) ความผิดพลาดประเภทการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด (Array index out of bound exception) ความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ (Arithmetic : Divided by zero exception) ความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง (Number format exception) และความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด (String index out of range)

4.1 อุปกรณ์ในการทดสอบ

- เครื่องคอมพิวเตอร์แบบพกพา HP Pavilion dv4 หน่วยประมวลผล Intel Core 2 Centrino 2.13 GHz หน่วยความจำหลัก 4 GB ฮาร์ดดิสต์วามจุ 300 GB ระบบปฏิบัติการ Windows 7 Service pack 1 ระบบประมวลผลแบบ 64-bit
- Java Development Kit (JDK) เวอร์ชัน 1.6
- โปรแกรม Eclipse SDK เวอร์ชัน 3.6.1
- โปรแกรมเสริม FindBugs Plug-in เวอร์ชัน 1.3.9

4.2 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง (Class cast exception)

ในหัวข้อนี้จะแสดงตัวอย่างความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง โดยรายละเอียดของการเกิดความผิดพลาดที่นำมาทดสอบมีดังนี้

1. การทดสอบด้วยตัวแปรระดับคลาส (Class variable)

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main**

```
public class testCase1_inMain {
      public static Object x = new Integer(3);
      public static void main(String args[]){
            String c = (String)x;
      }
}
```

รูปที่ 4.1 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อด main

- **เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main**

```
public class testCase1_inMethod {
      public static Object x =new Integer(3);
      public static void main(String args[]){
            testMethod();
      }
      public static void testMethod(){
            String c = (String)x;
      }
}
```

รูปที่ 4.2 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main

มหาวิทยาลัยเทคโนโลยีสุรนารี

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase1_1subMethod {
	public static Object x =new Integer(3);
	public static void main(String args[]){
		testMethod();
	}
	public static void testMethod(){
		subMethod1();
	}
	public static void subMethod1(){
		String c = (String)x;
	}
}
```

รูปที่ 4.3 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase1_2subMethod {
	public static Object x =new Integer(3);
	public static void main(String args[]){
		testMethod();
	}
	public static void testMethod(){
		subMethod1();
	}
	public static void subMethod1(){
		subMethod2();
	}
	public static void subMethod2(){
		String c = (String)x;
	}
}
```

รูปที่ 4.4 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

2. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main**

```
public class testCase2_inMain {

      public static void main(String args[]){
          Object x =new Integer(3);
            String c = (String)x;
      }
}
```

รูปที่ 4.5 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อด main

- **เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main**

```
public class testCase2_inMethod {

      public static void main(String args[]){
          testMethod();
      }
      public static void testMethod(){
            Object x =new Integer(3);
            String c = (String)x;
      }
}
```

รูปที่ 4.6 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase2_1subMethod {

      public static void main(String args[]){
          testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            Object x =new Integer(3);
            String c = (String)x;
      }
}
```

รูปที่ 4.7 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase2_2subMethod {

      public static void main(String args[]){
          testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            subMethod2();
      }
      public static void subMethod2(){
            Object x =new Integer(3);
            String c = (String)x;
      }
}
```

รูปที่ 4.8 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

3. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร

- **เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main**

```
public class testCase3_inMethodByval {

      public static void main(String args[]){
            Object x =new Integer(3);
          testMethod(x);
      }
      public static void testMethod(Object x){
            String c = (String)x;
      }
}
```

รูปที่ 4.9 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main

- ○ **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase3_1subMethodByval {

      public static void main(String args[]){
            Object x =new Integer(3);
          testMethod(x);
      }
      public static void testMethod(Object x){
            subMethod1(x);
      }
      public static void subMethod1(Object x){
            String c = (String)x;
      }
}
```

รูปที่ 4.10 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase3_2subMethodByval {

      public static void main(String args[]){
            Object x =new Integer(3);
          testMethod(x);
      }
      public static void testMethod(Object x){
            subMethod1(x);
      }
      public static void subMethod1(Object x){
            subMethod2(x);
      }
      public static void subMethod2(Object x){
            String c = (String)x;
      }
}
```

รูปที่ 4.11 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

ผลการทดสอบการค้นหาความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง

ในหัวข้อนี้จะเป็นการแสดงผลลัพธ์การค้นหาความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง โดยแสดงผลลัพธ์แบ่งตามกรณีทดสอบในหัวข้อที่ 4.2 ซึ่งกรณีทดสอบทั้งหมดที่นำมาทดสอบผู้วิจัยทราบก่อนล่วงหน้าว่าจะเกิดความผิดพลาดขึ้นจำนวนกี่จุด และจากรูปตัวอย่างกรณีทดสอบที่แสดงในหัวข้อที่ 4.2 ถูกนำเสนอในลักษณะซอสโค้ด แต่เมื่อจะนำไปทดสอบกับเครื่องมือตัวอย่างซอสโค้ดเหล่านี้จะถูกแปลงให้อยู่ในรูปแบบของไบต์โค้ดเสียก่อน เพื่อให้สามารถใช้เป็นข้อมูลนำเข้าของเครื่องมือตัวอย่างได้ โดยผลลัพธ์การทดสอบสามารถแสดงได้ดังตารางต่อไปนี้

ตารางที่ 4.1 ตารางแสดงผลการทดสอบการค้นหาความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง

กรณีทดสอบที่เกิดความผิดพลาดขึ้น	*A	* B	* C	* D
1. การทดสอบด้วยตัวแปรระดับคลาส (Class variable)				
ภายในเมท็อด main	1	1	0	1
ภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
2. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร				
ภายในเมท็อด main	1	1	0	1
ภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
3. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร				
ภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main	1	0	1	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	0	1	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	0	1	1
รวม	**11**	**8**	**3**	**11**

มหาวิทยาลัยเทคโนโลยีสุรนารี

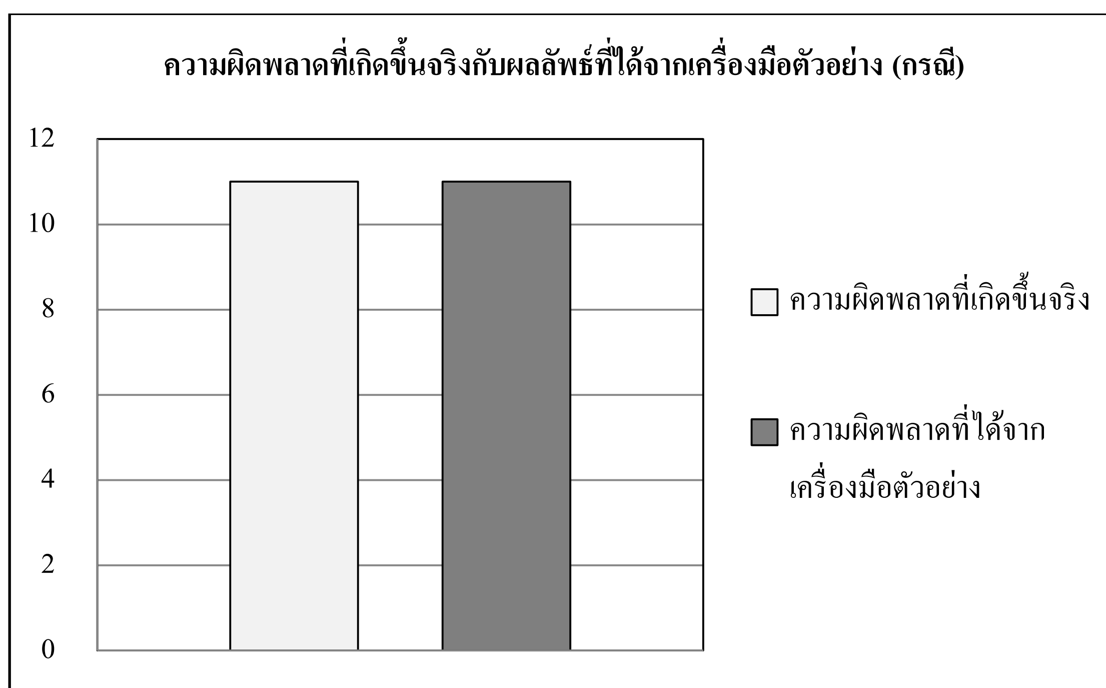
* A คือ จำนวนผลลัพธ์ที่ได้จากเครื่องมือตัวอย่าง

* B คือ จำนวนผลลัพธ์ที่ถูกต้อง

* C คือ จำนวนผลลัพธ์ที่ไม่ถูกต้อง

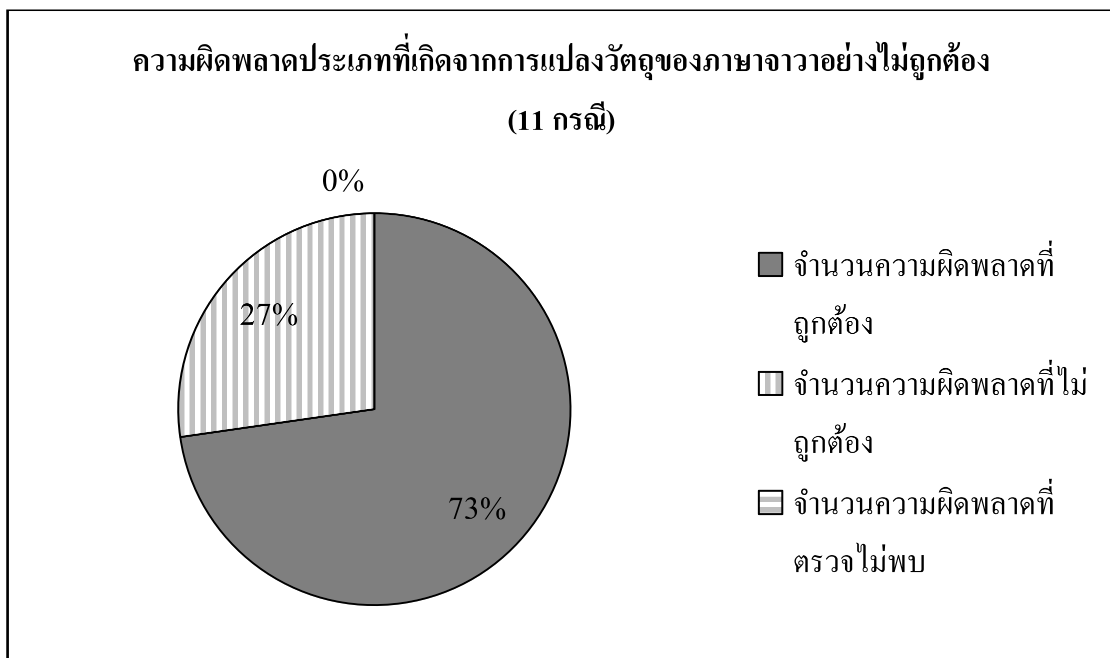
* D คือ จำนวนความผิดพลาดที่เกิดขึ้นจริง

การทดสอบความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง ในหัวข้อนี้ ผู้วิจัยได้นำเอากรณีทดสอบจำนวน 11 กรณีมาใช้เป็นข้อมูลนำเข้าให้แก่เครื่องมือตัวอย่างที่พัฒนาขึ้น ซึ่งผลการทดสอบสามารถสรุปได้ดังแผนภูมิในรูปที่ 4.12 และ 4.13



รูปที่ 4.12 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง

แผนภูมิที่แสดงในรูปที่ 4.12 อธิบายถึงจำนวนความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องที่เกิดขึ้นจริงในกรณีทดสอบ กับการแสดงผลลัพธ์ที่ได้จากจากเครื่องมือตัวอย่าง ซึ่งในที่นี้มีความผิดพลาดที่เกิดขึ้นจริงในกรณีทดสอบจำนวน 11 กรณี และเมื่อทดสอบกรณีทดสอบกับเครื่องมือตัวอย่าง เครื่องมือได้แสดงผลรายงานความผิดพลาดออกมาจำนวน 11 กรณี โดยในผลลัพธ์ที่แสดงสามารถแบ่งผลลัพธ์ออกเป็น 3 ประเภทคือ คือ รายงานผลลัพธ์ที่ถูกต้อง รายงานผลลัพธ์ที่ไม่ถูกต้อง และความผิดพลาดที่ตรวจไม่พบ ซึ่งสัดส่วนของผลลัพธ์ทั้ง 3 ประเภทสามารถแสดงได้ดังแผนภูมิวงกลมในรูปที่ 4.13



รูปที่ 4.13 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องด้วยเครื่องมือตัวอย่าง

จากแผนภูมิวงกลมในรูปที่ 4.13 อธิบายถึงผลลัพธ์การทดสอบที่เกิดขึ้นกับทุกกรณีการทดสอบ โดยสามารถแบ่งผลลัพธ์ได้เป็นการรายงานผลความผิดพลาดที่ถูกต้อง 8 กรณี การรายงานผลที่ไม่ถูกต้องจำนวน 3 กรณี และในการทดสอบนี้เครื่องมือตัวอย่างได้แสดงรายงานผลความผิดพลาดในทุกกรณีทดสอบ (ไม่มีกรณีทดสอบที่ไม่ค้นพบความผิดพลาด) เพียงแต่ในผลลัพธ์ที่รายงานมีผลลัพธ์ที่ไม่ถูกต้องประกอบอยู่ด้วย 3 กรณี

4.3 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดประเภทการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด (Array index out of bound exception)

ในหัวข้อนี้จะแสดงตัวอย่างความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด โดยรายละเอียดของการเกิดความผิดพลาดที่นำมาทดสอบมีดังนี้

1. การทดสอบด้วยตัวแปรระดับคลาส (Class variable)

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยการเรียกใช้อาร์เรย์อย่างง่าย**

```
public class testCase1_SampleinMain {
      public static int[] var = new int[5];
      public static void main(String[] args) {
            System.out.println(var[5]);
      }

}
```

รูปที่ 4.14 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อด main โดยใช้ตัวแปรระดับคลาส

- **เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main**

```
public class testCase1_SampleinMethod {
      public static int[] var = new int[5];
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            System.out.println(var[5]);
      }

}
```

รูปที่ 4.15 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase1_Sample1subMethod {
      public static int[] var = new int[5];
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            System.out.println(var[5]);
      }
}
```

รูปที่ 4.16 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase1_Sample2subMethod {
      public static int[] var = new int[5];
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            subMethod2();
      }
      public static void subMethod2(){
            System.out.println(var[5]);
      }
}
```

รูปที่ 4.17 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

2. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยการเรียกใช้อาร์เรย์อย่างง่าย**

```
public class testCase2_SampleinMain {
	public static void main(String args[]) {
		int[] arry = new int[5];
		System.out.println(arry[5]);
	}
}
```

รูปที่ 4.18 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อด main โดยใช้ตัวแปรภายใน

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน**

```
public class testCase2_inMain_for {
	public static void main(String args[]) {
		int[] arry = new int[5];
		for (int i = 0; i <= 5; i++) {
			System.out.println(arry[i]);
		}
	}
}
```

รูปที่ 4.19 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อด main โดยใช้ตัวแปรภายในลูป for

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายในเมท็อด**

```
public class testCase2_inMain_for {
      public static void main(String args[]) {
            int[] arry = new int[5];
            for (int i = 0; i <= 5; i++) {
                  System.out.println(arry[i]);
            }
      }
}
```

รูปที่ 4.20 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อด main โดยใช้ตัวแปรภายในเมท็อด

- **เกิดความผิดพลาดขึ้นในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main**

```
public class testCase2_SampleinMethod {
      public static void main(String args[]) {
            testMethod();
      }
      public static void testMethod(){
            int[] arry = new int[5];
            System.out.println(arry[5]);
      }
}
```

รูปที่ 4.21 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อดอื่นที่มีการเรียกใช้โดยเมท็อด main

- **เกิดความผิดพลาดขึ้นในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน**

```
public class testCase2_inMethod_for {
      public static void main(String args[]) {
            testMethod();
      }
      public static void testMethod(){
            int[] arry = new int[5];
            for (int i = 0; i <= 5; i++) {
                  System.out.println(arry[i]);
            }
      }
}
```

รูปที่ 4.22 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อดอื่นที่มีการเรียกใช้โดยเมท็อด main โดยใช้ตัวแปรภายในลูป for อย่างง่าย

- **เกิดความผิดพลาดขึ้นในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรในเมท็อด**

```
public class testCase2_inMethodByCon_for {
      public static void main(String args[]) {
            testMethod();
      }
      public static void testMethod(){
            int round = 5;
            int[] arry = new int[5];
            for (int i = 0; i <= round; i++) {
                  System.out.println(arry[i]);
            }
      }}
```

รูปที่ 4.23 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาในเมท็อดอื่นที่มีการเรียกใช้โดยเมท็อด main โดยใช้ตัวแปรภายในเมท็อด

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase2_Sample1subMethod {
      public static void main(String args[]) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            int[] arry = new int[5];
            System.out.println(arry[5]);
      }
}
```

รูปที่ 4.24 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase2_Sample2subMethod {
      public static void main(String args[]) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            subMethod2();
      }
      public static void subMethod2(){
            int[] arry = new int[5];
            System.out.println(arry[5]);
      }
}
```

รูปที่ 4.25 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียกเมท็อดย่อยจำนวน 2 ครั้ง

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน**

```
public class testCase2_1subMethod_for {
      public static void main(String args[]) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            int[] arry = new int[5];
            for (int i = 0; i <= 5; i++) {
                  System.out.println(arry[i]);
            }}}
```

รูปที่ 4.26 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของภาษาจาวาจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง ซึ่งเกิดความผิดพลาดภายในลูปที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน

- **เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน**

```
public class testCase2_2subMethod_for {
      public static void main(String args[]) {
            testMethod();}
      public static void testMethod(){
            subMethod1();}
      public static void subMethod1(){
            subMethod2();}
      public static void subMethod2(){
            int[] arry = new int[5];
            for (int i = 0; i <= 5; i++) {
                  System.out.println(arry[i]);
            }}}
```

รูปที่ 4.27 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง ซึ่ง เกิดความผิดพลาดภายในลูปที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน

- **เกิดความผิดพลาดขึ้นจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายใน**

```
public class testCase2_1subMethodByCon_for {
      public static void main(String args[]) {
            testMethod();}
      public static void testMethod(){
            subMethod1();}
      public static void subMethod1(){
            int round = 5;
            int[] arry = new int[5];
            for (int i = 0; i <= round; i++) {
                  System.out.println(arry[i]);
            }}}
```

รูปที่ 4.28 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง ซึ่งเกิดความผิดพลาดภายในลูปที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายใน

- **เกิดความผิดพลาดขึ้นจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายใน**

```
public class testCase2_2subMethodByCon_for {
      public static void main(String args[]) {
            testMethod();}
      public static void testMethod(){
            subMethod1();}
      public static void subMethod1(){
            subMethod2();}
      public static void subMethod2(){
            int round = 5;
            int[] arry = new int[5];
            for (int i = 0; i <= round; i++) {
                  System.out.println(arry[i]);
            }}}
```

รูปที่ 4.29 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดของจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง ซึ่งเกิดความผิดพลาดภายในลูปที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายใน

3. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร

- **เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main**

```
public class testCase3_SampleinMethod {
      public static void main(String args[]) {
            int[] var = new int[5];
            testMethod(var);
      }
      public static void testMethod(int[] var){
            System.out.println(var[5]);
      }
}
```

รูปที่ 4.30 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main

- **เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase3_Sample1subMethod {
      public static void main(String args[]) {
            int[] var = new int[5];
            testMethod(var);
      }
      public static void testMethod(int[] var){
            subMethod1(var);
      }
      public static void subMethod1(int[] var){
            System.out.println(var[5]);
      }
}
```

รูปที่ 4.31 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase3_Sample2subMethod {
	public static void main(String args[]) {
		int[] var = new int[5];
		testMethod(var);
	}
	public static void testMethod(int[] var){
		subMethod1(var);
	}
	public static void subMethod1(int[] var){
		subMethod2(var);
	}
	public static void subMethod2(int[] var){
		System.out.println(var[5]);
	}
}
```

รูปที่ 4.32 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้องภายในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

ผลการทดสอบการค้นหาความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวา

ในหัวข้อนี้จะเป็นการแสดงผลลัพธ์การค้นหาความผิดพลาดประเภทที่เกิดจากการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนด โดยแสดงผลลัพธ์แบ่งตามกรณีทดสอบในหัวข้อที่ 4.3 ซึ่งกรณีทดสอบทั้งหมดที่นำมาทดสอบผู้วิจัยทราบก่อนล่วงหน้าว่าจะเกิดความผิดพลาดขึ้นจำนวนกี่จุด และจากรูปตัวอย่างกรณีทดสอบที่แสดงในหัวข้อที่ 4.3 ถูกนำเสนอในลักษณะซอสโค้ด แต่เมื่อจะนำไปทดสอบกับเครื่องมือตัวอย่างซอสโค้ดเหล่านี้จะถูกแปลงให้อยู่ในรูปแบบของไบต์โค้ดเสียก่อน เพื่อให้สามารถใช้เป็นข้อมูลนำเข้าของเครื่องมือตัวอย่างได้ โดยผลลัพธ์การทดสอบสามารถแสดงได้ดังตารางต่อไปนี้

ตารางที่ 4.2 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวา

กรณีทดสอบที่เกิดความผิดพลาดขึ้น	*A	* B	* C	* D
1. การทดสอบด้วยตัวแปรระดับคลาส (Class variable)				
ภายในเมท็อด main	1	1	0	1
ภายในเมท็อด main โดยการเรียกใช้อาร์เรย์อย่างง่าย	1	1	0	1
ภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
2. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร				
ภายในเมท็อด main โดยการเรียกใช้อาร์เรย์อย่างง่าย	1	1	0	1
ภายในเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน	1	1	0	1
ภายในเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายในเมท็อด	1	1	0	1
ภายเกิดความผิดพลาดขึ้นในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main	1	1	0	1
เมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรในลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน	1	1	0	1
เมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main โดยการใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดรอบทำงานจากตัวแปรในเมท็อด	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน	1	1	0	1
เมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดรอบการทำงานจากตัวแปรภายใน	1	1	0	1

ตารางที่ 4.2 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวา (ต่อ)

กรณีทดสอบที่เกิดความผิดพลาดขึ้น	***A**	*** B**	*** C**	*** D**
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดรอบการทำงานจากตัวแปรภายใน	1	1	0	1
3. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร				
ความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main	0	0	0	1
ความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	0	0	0	1
ความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	0	0	0	1
รวม	**16**	**16**	**0**	**19**

* A คือ จำนวนผลลัพธ์ที่ได้จากเครื่องมือตัวอย่าง

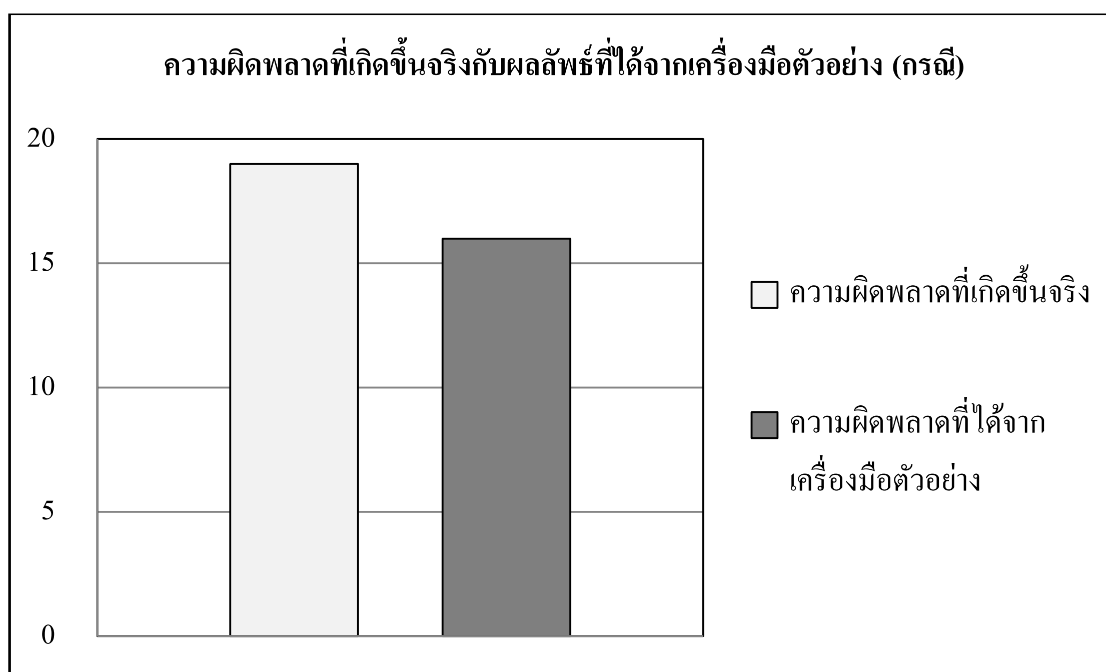
* B คือ จำนวนผลลัพธ์ที่ถูกต้อง

* C คือ จำนวนผลลัพธ์ที่ไม่ถูกต้อง

* D คือ จำนวนความผิดพลาดที่เกิดขึ้นจริง

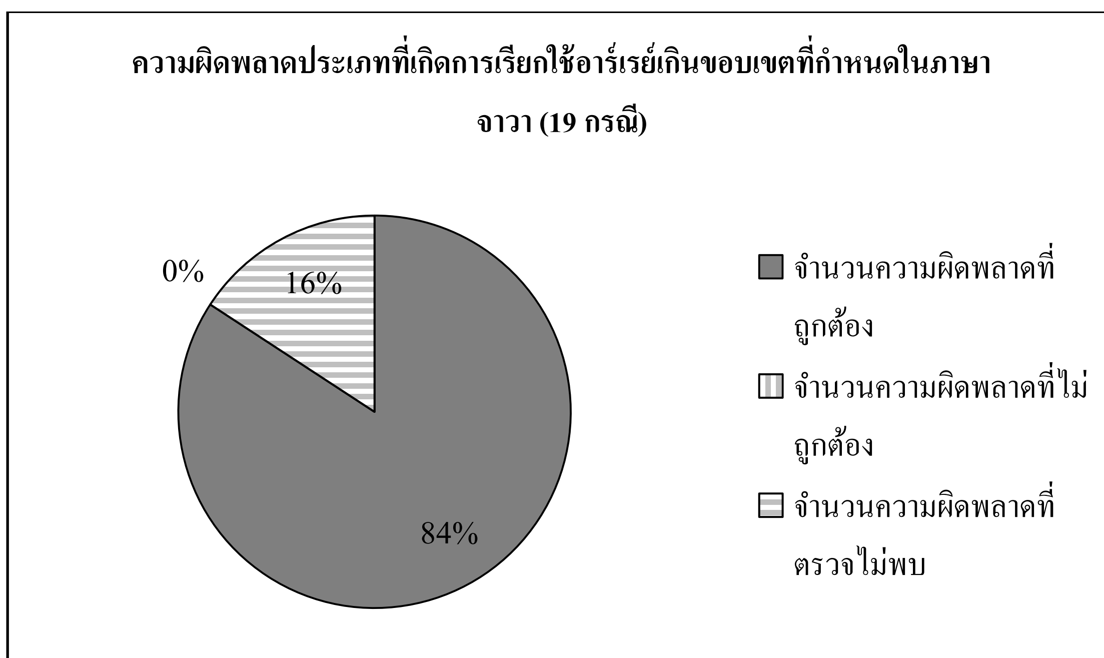
มหาวิทยาลัยเทคโนโลยีสุรนารี

การทดสอบความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวาในหัวข้อนี้ ผู้วิจัยได้นำเอากรณีทดสอบจำนวน 19 กรณีมาใช้เป็นข้อมูลนำเข้าให้แก่เครื่องมือตัวอย่างที่พัฒนาขึ้น ซึ่งผลการทดสอบสามารถสรุปได้ดังแผนภูมิในรูปที่ 4.33 และ 4.34



รูปที่ 4.33 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวา

แผนภูมิที่แสดงในรูปที่ 4.33 อธิบายถึงจำนวนความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวาที่เกิดขึ้นจริงในกรณีทดสอบ กับการแสดงผลลัพธ์ที่ได้จากจากเครื่องมือตัวอย่าง ซึ่งในที่นี้มีความผิดพลาดที่เกิดขึ้นจริงในกรณีทดสอบจำนวน 19 กรณี และเมื่อทดสอบกรณีทดสอบกับเครื่องมือตัวอย่าง เครื่องมือได้แสดงผลรายงานความผิดพลาดออกมาจำนวน 16 กรณี โดยในผลลัพธ์ที่แสดงสามารถแบ่งผลลัพธ์ออกเป็น 3 ประเภทคือ คือ รายงานผลลัพธ์ที่ถูกต้อง รายงานผลลัพธ์ที่ไม่ถูกต้อง และความผิดพลาดที่ตรวจไม่พบ ซึ่งสัดส่วนของผลลัพธ์ทั้ง 3 ประเภทสามารถแสดงได้ดังแผนภูมิวงกลมในรูปที่ 4.34



รูปที่ 4.34 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวาด้วยเครื่องมือตัวอย่าง

จากแผนภูมิวงกลมในรูปที่ 4.34 อธิบายถึงผลลัพธ์การทดสอบที่เกิดขึ้นกับทุกกรณีการทดสอบ ซึ่งสามารถแบ่งผลลัพธ์ได้เป็นการรายงานผลความผิดพลาดที่ถูกต้อง 16 กรณี โดยในการทดสอบครั้งนี้มีกรณีมีกรณีทดสอบจำนวน 3 กรณีที่เครื่องมือตัวอย่างไม่สามารถตรวจหาความผิดพลาดพบ และในผลลัพธ์ที่เครื่องมือตัวอย่างรายงานผลไม่มีผลลัพธ์ที่รายงานผลไม่ถูกต้อง

4.4 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ (Arithmetic : Divided by zero exception)

ในหัวข้อนี้จะแสดงตัวอย่างความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ โดยรายละเอียดของการเกิดความผิดพลาดที่นำมาทดสอบมีดังนี้

1. การทดสอบด้วยตัวแปรระดับคลาส (Class variable)

- **เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main**

```
public class testCase1_SampleinMain {
      public static int var = 5;
      public static void main(String[] args) {
            System.out.println(var/0);
      }

}
```

รูปที่ 4.35 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยความผิดพลาดที่เกิดขึ้นสามารถสังเกตได้ง่าย

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main เนื่องจากตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์**

```
public class testCase1_SampleinMain2 {
      public static int var = 5;
      public static int num = 0;
      public static void main(String[] args) {
            System.out.println(var/num);
      }

}
```

รูปที่ 4.36 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีค่าเป็นศูนย์

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ**

```
public class testCase1_inMainFor {
      public static int var = 5;
      public static void main(String[] args) {
            for(int i=0;i<5;i++){
                  System.out.println(var/i);
            }
      }
}
```

รูปที่ 4.37 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีการเพิ่มค่าภายในลูป for

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ**

```
public class testCase1_inMainForDec {
      public static int var = 5;
      public static void main(String[] args) {
            for(int i=5;i>=0;i--){
                  System.out.println(var/i);
            }
      }
}
```

รูปที่ 4.38 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีการลดค่าภายในลูป for

- **เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main**

```
public class testCase1_SampleinMethod {
      public static int var = 5;
      public static int num = 0;
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            System.out.println(var/num);
      }
}
```

รูปที่ 4.39 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์

- **เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ**

```
public class testCase1_inMethodFor {
      public static int var = 5;
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            for(int i=0;i<5;i++){
                  System.out.println(var/i);
            }
      }
}
```

รูปที่ 4.40 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าในลูป for

- **เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ**

```
public class testCase1_inMethodForDec {
      public static int var = 5;
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            for(int i=5;i>=0;i--){
                  System.out.println(var/i);
            }
      }
}
```

รูปที่ 4.41 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าในลูป for

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase1_1SubMethod {
      public static int var = 5;
      public static int num = 0;
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            System.out.println(var/num);
      }

}
```

รูปที่ 4.42 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ**

```
public class testCase1_1SubMethodFor {
	public static int var = 5;
	public static void main(String[] args) {
		testMethod();
	}
	public static void testMethod(){
		subMethod1();
	}
	public static void subMethod1(){
		for(int i=0;i<5;i++){
			System.out.println(var/i);
		}
	}}
```

รูปที่ 4.43 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปรที่เป็นตัวหารมีการเพิ่มค่าภายในลูป for

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ**

มหาวิทยาลัยเทคโนโลยีสุรนารี

```
public class testCase1_1SubMethodForDec {
	public static int var = 5;
	public static void main(String[] args) {
		testMethod();}
	public static void testMethod(){
		subMethod1();}
	public static void subMethod1(){
		for(int i=5;i>=0;i--){
			System.out.println(var/i);
		}}}
```

รูปที่ 4.44 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปรที่เป็นตัวหารมีการลดค่าภายในลูป for

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase1_2SubMethod {
      public static int var = 5;
      public static int num = 0;
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            subMethod2();
      }
      public static void subMethod2(){
            System.out.println(var/num);
      }

}
```

รูปที่ 4.45 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 2 ครั้ง

2. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร

- **เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main**

```
public class testCase2_SampleinMain {
      public static void main(String[] args) {
            int var = 5;
            System.out.println(var/0);
      }
}
```

รูปที่ 4.46 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยความผิดพลาดที่เกิดขึ้นสามารถสังเกตได้ง่าย

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main เนื่องจากตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์**

```
public class testCase2_SampleinMain2 {
      public static void main(String[] args) {
            int var = 5;
            int num = 0;
            System.out.println(var/num);
      }
}
```

รูปที่ 4.47 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีค่าเป็นศูนย์

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ**

```
public class testCase2_inMainFor {
      public static void main(String[] args) {
            int var = 5;
            for(int i=0;i<5;i++){
                  System.out.println(var/i);
            }
      }
}
```

รูปที่ 4.48 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีการเพิ่มค่าภายในลูป for

- **เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ**

```
public class testCase2_inMainForDec {
	public static void main(String[] args) {
		for(int i=5;i>=0;i--){
			int var = 5;
			System.out.println(var/i);
		}
	}
}
```

รูปที่ 4.49 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อด main โดยตัวแปรที่นำมาเป็นตัวหารมีการลดค่าภายในลูป for

- **เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main**

```
public class testCase2_SampleinMethod {

	public static void main(String[] args) {
		testMethod();
	}
	public static void testMethod(){
		int var = 5;
		int num = 0;
		System.out.println(var/num);
	}

}
```

รูปที่ 4.50 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์

- **เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ**

```
public class testCase2_inMethodFor {
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            int var = 5;
            for(int i=0;i<5;i++){
                  System.out.println(var/i);
            }
      }
}
```

รูปที่ 4.51 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าในลูป for

- **เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ**

```
public class testCase2_inMethodForDec {
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            int var = 5;
            for(int i=5;i>=0;i--){
                  System.out.println(var/i);
            }
      }
}
```

รูปที่ 4.52 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าในลูป for

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase2_1SubMethod {
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            int var = 5;
            int num = 0;
            System.out.println(var/num);
      }

}
```

รูปที่ 4.53 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ**

```
public class testCase2_1SubMethodFor {
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            int var = 5;
            for(int i=0;i<5;i++){
                  System.out.println(var/i);
            }}}
```

รูปที่ 4.54 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปรที่เป็นตัวหารมีการเพิ่มค่าภายในลูป for

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ**

```
public class testCase2_1SubMethodForDec {
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            int var = 5;
            for(int i=5;i>=0;i--){
                  System.out.println(var/i);
            }}}
```

รูปที่ 4.55 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปรที่เป็นตัวหารมีการลดค่าภายในลูป for

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase2_2SubMethod {
      public static void main(String[] args) {
            testMethod();
      }
      public static void testMethod(){
            subMethod1();
      }
      public static void subMethod1(){
            subMethod2();
      }
      public static void subMethod2(){
            int var = 5;
            int num = 0;
            System.out.println(var/num);
      }

}
```

รูปที่ 4.56 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 2 ครั้ง

3. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร

- **เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main**

```
public class testCase3_SampleinMethod {

      public static void main(String[] args) {
            int var1 = 5;
            int var2 = 0;
            testMethod(var1,var2);
      }
      public static void testMethod(int var,int num){
            System.out.println(var/num);
      }

}
```

รูปที่ 4.57 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์

- **เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ**

```
public class testCase3_inMethodFor {
      public static void main(String[] args) {
            int var = 5;
            testMethod(var);
      }
      public static void testMethod(int var){
            for(int i=0;i<5;i++){
                  System.out.println(var/i);
            }
      }
}
```

รูปที่ 4.58 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าในลูป for

- **เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ**

```
public class testCase3_inMethodForDec {
	public static void main(String[] args) {
		int var = 5;
		testMethod(var);
	}
	public static void testMethod(int var){
		for(int i=5;i>=0;i--){
			System.out.println(var/i);
		}
	}
}
```

รูปที่ 4.59 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดที่ถูกเรียกใช้จากเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าในลูป for

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง**

```
public class testCase3_1SubMethod {
	public static void main(String[] args) {
		int var1 = 5;
		int var2 = 0;
		testMethod(var1,var2);
	}
	public static void testMethod(int var1,int var2){
		subMethod1(var1,var2);
	}
	public static void subMethod1(int var,int num){
		System.out.println(var/num);
	}
}
```

รูปที่ 4.60 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้ง

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ**

```
public class testCase3_1SubMethodFor {
      public static void main(String[] args) {
            int var = 5;
            testMethod(var);
      }
      public static void testMethod(int var){
            subMethod1(var);
      }
      public static void subMethod1(int var){
            for(int i=0;i<5;i++){
                  System.out.println(var/i);
            }}}
```

รูปที่ 4.61 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปรที่เป็นตัวหารมีการเพิ่มค่าภายในลูป for

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ**

```
public class testCase3_1SubMethodForDec {
      public static void main(String[] args) {
            int var = 5;
            testMethod(var);}
      public static void testMethod(int var){
            subMethod1(var);}
      public static void subMethod1(int var){
            for(int i=5;i>=0;i--){
                  System.out.println(var/i);
            }}}
```

รูปที่ 4.62 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 1 ครั้งโดยที่ตัวแปรที่เป็นตัวหารมีการลดค่าภายในลูป for

มหาวิทยาลัยเทคโนโลยีสุรนารี

- **เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง**

```
public class testCase3_2SubMethod {
      public static void main(String[] args) {
            int var = 5;
            int num = 0;
            testMethod(var,num);
      }
      public static void testMethod(int var,int num){
            subMethod1(var,num);
      }
      public static void subMethod1(int var,int num){
            subMethod2(var,num);
      }
      public static void subMethod2(int var,int num){
            System.out.println(var/num);
      }

}
```

รูปที่ 4.63 ตัวอย่างซอสโค้ดที่ทำให้เกิดความผิดพลาดประเภทเกิดจากการหารด้วยศูนย์ในเมท็อดย่อยที่ถูกเรียกใช้จากเมท็อด main ซ้อนจำนวน 2 ครั้ง

ผลการทดสอบการค้นหาความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์

ในหัวข้อนี้จะเป็นการแสดงผลลัพธ์การค้นหาความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ โดยแสดงผลลัพธ์แบ่งตามกรณีทดสอบในหัวข้อที่ 4.4 ซึ่งกรณีทดสอบทั้งหมดที่นำมาทดสอบผู้วิจัยทราบก่อนล่วงหน้าว่าจะเกิดความผิดพลาดขึ้นจำนวนกี่จุด และจากรูปตัวอย่างกรณีทดสอบที่แสดงในหัวข้อที่ 4.4 ถูกนำเสนอในลักษณะซอสโค้ด แต่เมื่อจะนำไปทดสอบกับเครื่องมือตัวอย่างซอสโค้ดเหล่านี้จะถูกแปลงให้อยู่ในรูปแบบของไบต์โค้ดเสียก่อน เพื่อให้สามารถใช้เป็นข้อมูลนำเข้าของเครื่องมือตัวอย่างได้ โดยผลลัพธ์การทดสอบสามารถแสดงได้ดังตารางที่ 4.3

ตารางที่ 4.3 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์

กรณีทดสอบที่เกิดความผิดพลาดขึ้น	*A	* B	* C	* D
1. การทดสอบด้วยตัวแปรระดับคลาส (Class variable)				
ความผิดพลาดอย่างง่ายที่เกิดขึ้นภายในเมท็อด main	1	1	0	1
ภายในเมท็อด main เนื่องจากตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์	1	1	0	1
ภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากลูป for	1	1	0	1
ภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าจากลูป for	0	0	0	1
ความผิดพลาดอย่างง่ายภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main	1	1	0	1
ภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากลูป for	1	1	0	1
ภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากลูป for	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
2. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร				
ความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main	1	1	0	1
ภายในเมท็อด main เนื่องจากตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์	1	1	0	1
ภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากลูป for	1	1	0	1
เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ	0	0	0	1
ความผิดพลาดอย่างง่ายภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main	1	1	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากลูป for	1	1	0	1

ตารางที่ 4.3 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ (ต่อ)

กรณีทดสอบที่เกิดความผิดพลาดขึ้น	*A	* B	* C	* D
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากลูป for	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากลูป for	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และ ตัวแปรที่เป็นตัวหารมีการลดค่าจากลูป for	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
3. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร				
ความผิดพลาดอย่างง่ายภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main	0	0	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากลูป for	0	0	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากลูป for	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	0	0	0	1
รวม	**16**	**16**	**0**	**29**

* A คือ จำนวนผลลัพธ์ที่ได้จากเครื่องมือตัวอย่าง

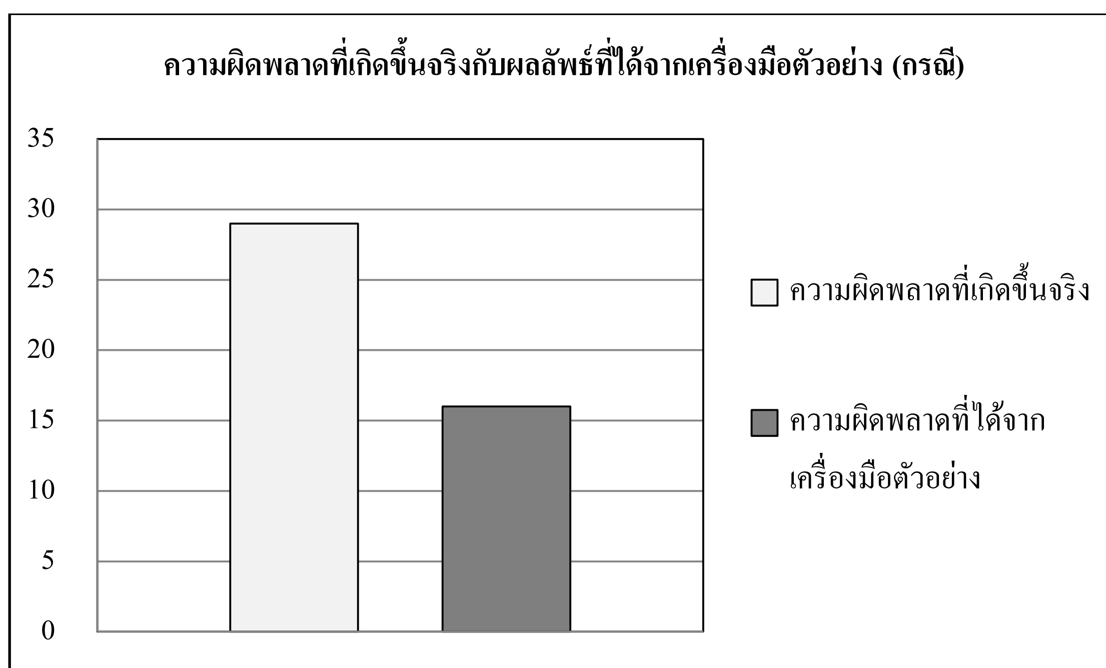
* B คือ จำนวนผลลัพธ์ที่ถูกต้อง

* C คือ จำนวนผลลัพธ์ที่ไม่ถูกต้อง

* D คือ จำนวนความผิดพลาดที่เกิดขึ้นจริง

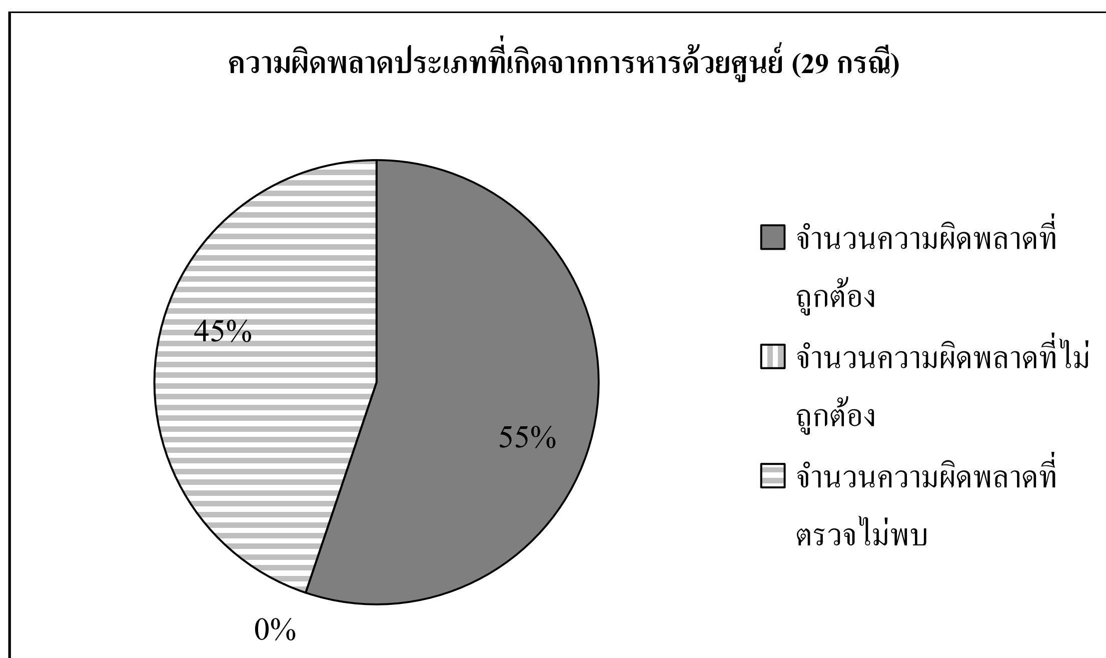
มหาวิทยาลัยเทคโนโลยีสุรนารี

การทดสอบความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์จาวาในหัวข้อนี้ ผู้วิจัยได้นำเอากรณีทดสอบจำนวน 29 กรณีมาใช้เป็นข้อมูลนำเข้าให้แก่เครื่องมือตัวอย่างที่พัฒนาขึ้น ซึ่งผลการทดสอบสามารถสรุปได้ดังแผนภูมิในรูปที่ 4.64 และ 4.65



รูปที่ 4.64 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์

แผนภูมิที่แสดงในรูปที่ 4.64 อธิบายถึงจำนวนความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ที่เกิดขึ้นจริงในกรณีทดสอบ กับการแสดงผลลัพธ์ที่ได้จากจากเครื่องมือตัวอย่าง ซึ่งในที่นี้มีความผิดพลาดที่เกิดขึ้นจริงในกรณีทดสอบจำนวน 29 กรณี และเมื่อทดสอบกรณีทดสอบกับเครื่องมือตัวอย่าง เครื่องมือได้แสดงผลรายงานความผิดพลาดออกมาจำนวน 16 กรณี โดยในผลลัพธ์ที่แสดงสามารถแบ่งผลลัพธ์ออกเป็น 3 ประเภทคือ คือ รายงานผลลัพธ์ที่ถูกต้อง รายงานผลลัพธ์ที่ไม่ถูกต้อง และความผิดพลาดที่ตรวจไม่พบ ซึ่งสัดส่วนของผลลัพธ์ทั้ง 3 ประเภทสามารถแสดงได้ดังแผนภูมิวงกลมในรูปที่ 4.65



รูปที่ 4.65 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์ด้วยเครื่องมือตัวอย่าง

จากแผนภูมิวงกลมในรูปที่ 4.65 อธิบายถึงผลลัพธ์การทดสอบที่เกิดขึ้นกับทุกกรณีการทดสอบ ซึ่งสามารถแบ่งผลลัพธ์ได้เป็นการรายงานผลความผิดพลาดที่ถูกต้อง 16 กรณี โดยในการทดสอบครั้งนี้มีกรณีมีกรณีทดสอบจำนวน 13 กรณีที่เครื่องมือตัวอย่างไม่สามารถตรวจหาความผิดพลาดพบ และในผลลัพธ์ที่เครื่องมือตัวอย่างรายงานผลไม่มีผลลัพธ์ที่รายงานผลไม่ถูกต้อง

4.5 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง (Number format exception)

ตารางที่ 4.4 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง

กรณีทดสอบที่เกิดความผิดพลาดขึ้น	*A	* B	* C	* D
1. การทดสอบด้วยตัวแปรระดับคลาส (Class variable)				
ความผิดพลาดอย่างง่ายภายในเมท็อด main	1	1	0	1
ภายในเมท็อด main เนื่องจากตัวแปรที่นำมาจัดรูปแบบมีค่าไม่ถูกต้อง	1	1	0	1
ภายในเมท็อด main โดยตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง	1	1	0	1
ความผิดพลาดอย่างง่ายภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main	1	1	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main และ ตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และ ตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
2. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร				
ความผิดพลาดอย่างง่ายภายในเมท็อด main	1	1	0	1
ภายในเมท็อด main เนื่องจากตัวแปรที่นำมาจัดรูปแบบมีค่าไม่ถูกต้อง	1	1	0	1
ภายในเมท็อด main โดยตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง	1	1	0	1
ความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main	1	1	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1

ตารางที่ 4.4 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง (ต่อ)

กรณีทดสอบที่เกิดความผิดพลาดขึ้น	***A**	*** B**	*** C**	*** D**
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
3. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร				
ความผิดพลาดอย่างง่ายภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main	0	0	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	0	0	0	1
รวม	**16**	**16**	**0**	**21**

* A คือ จำนวนผลลัพธ์ที่ได้จากเครื่องมือตัวอย่าง

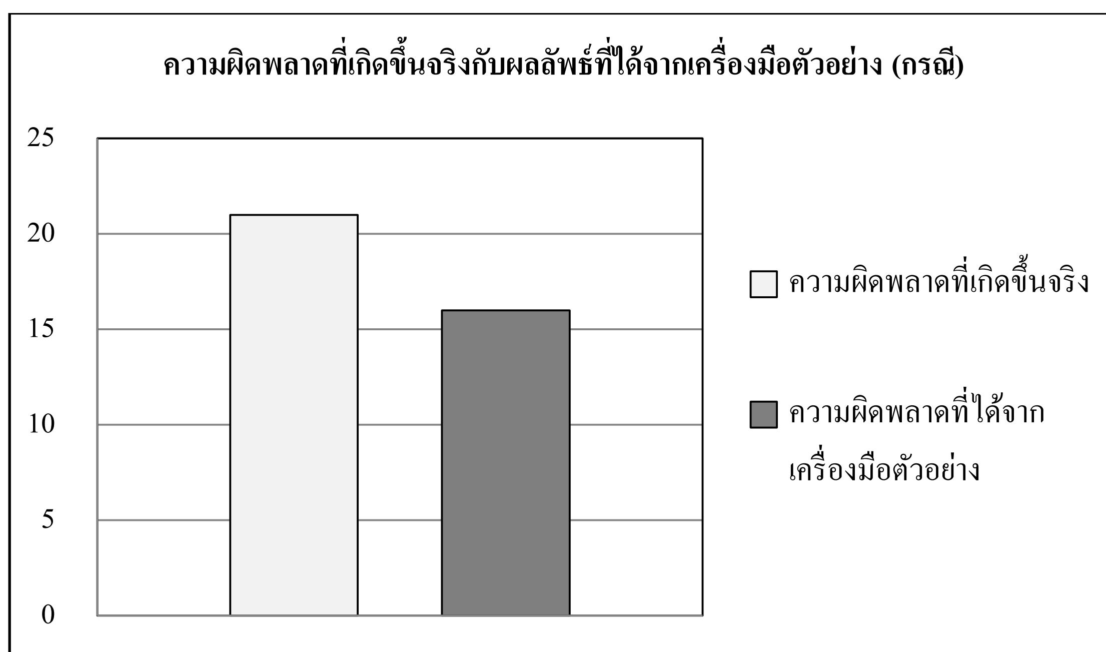
* B คือ จำนวนผลลัพธ์ที่ถูกต้อง

* C คือ จำนวนผลลัพธ์ที่ไม่ถูกต้อง

* D คือ จำนวนความผิดพลาดที่เกิดขึ้นจริง

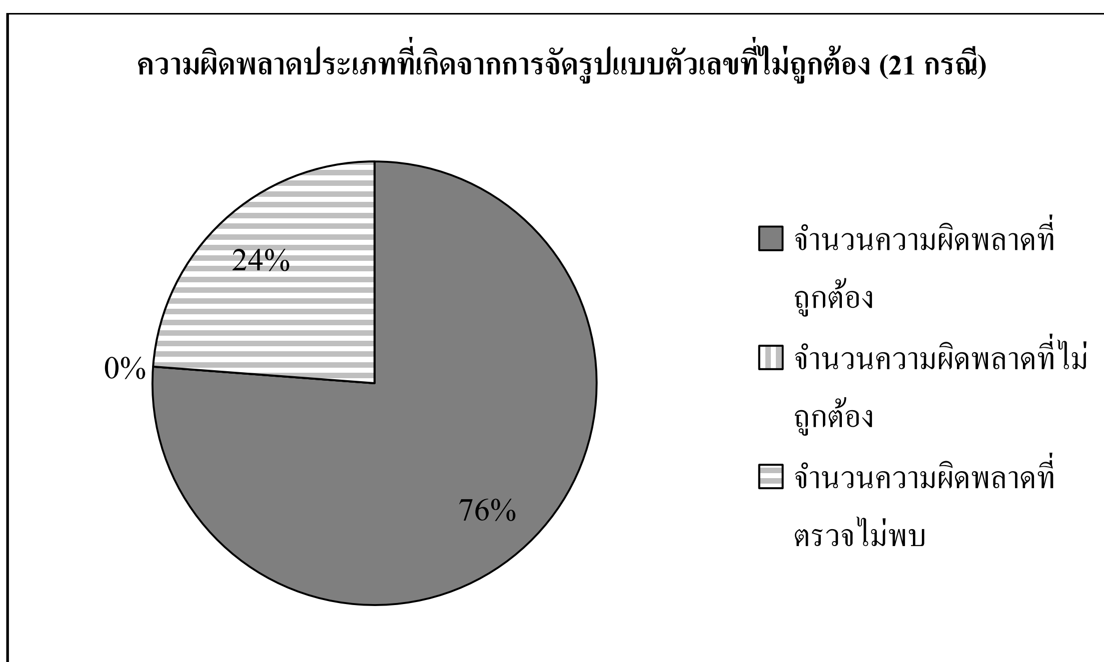
มหาวิทยาลัยเทคโนโลยีสุรนารี

การทดสอบความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้องจาวาในหัวข้อนี้ ผู้วิจัยได้นำเอากรณีทดสอบจำนวน 21 กรณีมาใช้เป็นข้อมูลนำเข้าให้แก่เครื่องมือตัวอย่างที่พัฒนาขึ้น ซึ่งผลการทดสอบสามารถสรุปได้ดังแผนภูมิในรูปที่ 4.66 และ 4.7



รูปที่ 4.66 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง

แผนภูมิที่แสดงในรูปที่ 4.66 อธิบายถึงจำนวนความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้องที่เกิดขึ้นจริงในกรณีทดสอบ กับการแสดงผลลัพธ์ที่ได้จากจากเครื่องมือตัวอย่าง ซึ่งในที่นี้มีความผิดพลาดที่เกิดขึ้นจริงในกรณีทดสอบจำนวน 21 กรณี และเมื่อทดสอบกรณีทดสอบกับเครื่องมือตัวอย่าง เครื่องมือได้แสดงผลรายงานความผิดพลาดออกมาจำนวน 16 กรณี โดยในผลลัพธ์ที่แสดงสามารถแบ่งผลลัพธ์ออกเป็น 3 ประเภทคือ คือ รายงานผลลัพธ์ที่ถูกต้อง รายงานผลลัพธ์ที่ไม่ถูกต้อง และความผิดพลาดที่ตรวจไม่พบ ซึ่งสัดส่วนของผลลัพธ์ทั้ง 3 ประเภทสามารถแสดงได้ดังแผนภูมิวงกลมในรูปที่ 4.67



รูปที่ 4.67 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้องด้วยเครื่องมือตัวอย่าง

จากแผนภูมิวงกลมในรูปที่ 4.67 อธิบายถึงผลลัพธ์การทดสอบที่เกิดขึ้นกับทุกกรณีการทดสอบ ซึ่งสามารถแบ่งผลลัพธ์ได้เป็นการรายงานผลความผิดพลาดที่ถูกต้อง 16 กรณี โดยในการทดสอบครั้งนี้มีกรณีมีกรณีทดสอบจำนวน 5 กรณีที่เครื่องมือตัวอย่างไม่สามารถตรวจหาความผิดพลาดพบ และในผลลัพธ์ที่เครื่องมือตัวอย่างรายงานผล ไม่มีผลลัพธ์ที่รายงานผลไม่ถูกต้อง

4.6 ตัวอย่างการทดสอบเครื่องมือด้วยข้อมูลความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด (String index out of range)

ตารางที่ 4.5 ตารางแสดงผลการทดสอบค้นหาความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด

กรณีทดสอบที่เกิดความผิดพลาดขึ้น	*A	* B	* C	* D
1. การทดสอบด้วยตัวแปรระดับคลาส (Class variable)				
ความผิดพลาดอย่างง่ายภายในเมท็อด main	1	1	0	1
ความผิดพลาดภายในเมท็อด main เนื่องจากการตัดสตริงไม่ถูกต้อง	1	1	0	1
ภายในเมท็อด main โดยเกิดความผิดพลาดขึ้นภายในลูป for	1	1	0	1
ภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main	1	1	0	1
ภายในเมท็อดที่ถูกเรียกโดยเมท็อด main และ ตัดสตริงภายในลูป for	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
2. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร				
ความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main	1	1	0	1
ภายในเมท็อด main เนื่องจากการตัดสตริงไม่ถูกต้อง	1	1	0	1
ภายในเมท็อด main จากการตัดสตริงภายในลูป for	1	1	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main	1	1	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main จากการตัดสตริงภายในลูป for	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	1	1	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	1	1	0	1
3. การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร				
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main	0	0	0	1
เมท็อดที่ถูกเรียกใช้โดยเมท็อด main จากการตัดสตริงภายในลูป for	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง	0	0	0	1
เมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง	0	0	0	1
รวม	**14**	**14**	**0**	**18**

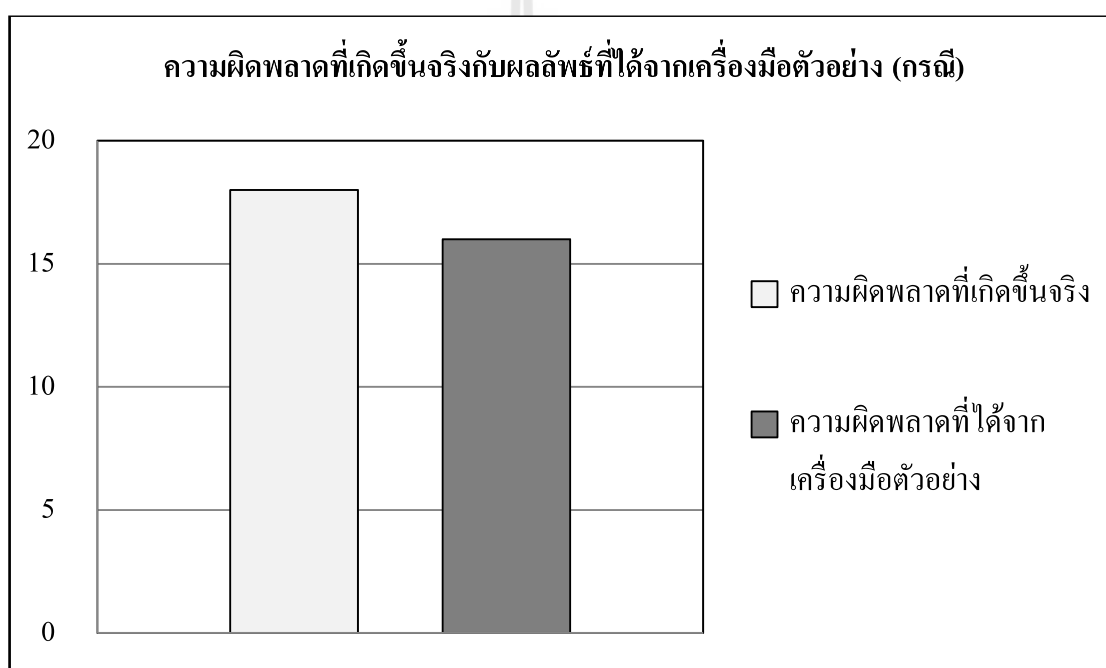
* A คือ จำนวนผลลัพธ์ที่ได้จากเครื่องมือตัวอย่าง

* B คือ จำนวนผลลัพธ์ที่ถูกต้อง

* C คือ จำนวนผลลัพธ์ที่ไม่ถูกต้อง

* D คือ จำนวนความผิดพลาดที่เกิดขึ้นจริง

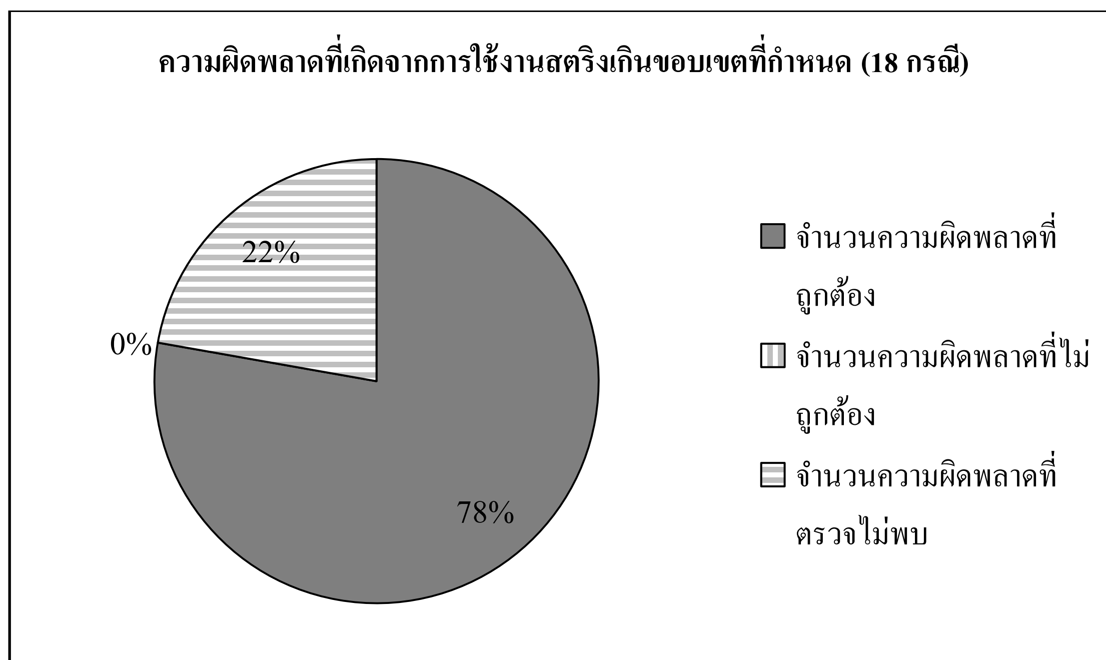
การทดสอบความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนดจาวาในหัวข้อนี้ ผู้วิจัยได้นำเอากรณีทดสอบจำนวน 18 กรณีมาใช้เป็นข้อมูลนำเข้าให้แก่เครื่องมือตัวอย่างที่พัฒนาขึ้น ซึ่งผลการทดสอบสามารถสรุปได้ดังแผนภูมิในรูปที่ 4.68 และ 4.9



รูปที่ 4.68 แผนภูมิแท่งแสดงจำนวนของผลลัพธ์จากการทดสอบความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด

แผนภูมิที่แสดงในรูปที่ 4.68 อธิบายถึงจำนวนความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนดที่เกิดขึ้นจริงในกรณีทดสอบ กับการแสดงผลลัพธ์ที่ได้จากจากเครื่องมือตัวอย่าง ซึ่งในที่นี้มีความผิดพลาดที่เกิดขึ้นจริงในกรณีทดสอบจำนวน 18 กรณี และเมื่อทดสอบกรณีทดสอบกับเครื่องมือตัวอย่าง เครื่องมือได้แสดงผลรายงานความผิดพลาดออกมาจำนวน 14 กรณี โดยในผลลัพธ์ที่แสดงสามารถแบ่งผลลัพธ์ออกเป็น 3 ประเภทคือ คือ รายงานผลลัพธ์ที่ถูกต้อง

รายงานผลลัพธ์ที่ไม่ถูกต้อง และความผิดพลาดที่ตรวจไม่พบ ซึ่งสัดส่วนของผลลัพธ์ทั้ง 3 ประเภทสามารถแสดงได้ดังแผนภูมิวงกลมในรูปที่ 4.69



รูปที่ 4.69 แผนภูมิวงกลมแสดงสัดส่วนของผลลัพธ์จากการทดสอบความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนดด้วยเครื่องมือตัวอย่าง

จากแผนภูมิวงกลมในรูปที่ 4.69 อธิบายถึงผลลัพธ์การทดสอบที่เกิดขึ้นกับทุกกรณีการทดสอบ ซึ่งสามารถแบ่งผลลัพธ์ได้เป็นการรายงานผลความผิดพลาดที่ถูกต้อง 14 กรณี โดยในการทดสอบครั้งนี้มีกรณีมีกรณีทดสอบจำนวน 4 กรณีที่เครื่องมือตัวอย่างไม่สามารถตรวจหาความผิดพลาดพบ และในผลลัพธ์ที่เครื่องมือตัวอย่างรายงานผลไม่มีผลลัพธ์ที่รายงานผลไม่ถูกต้อง

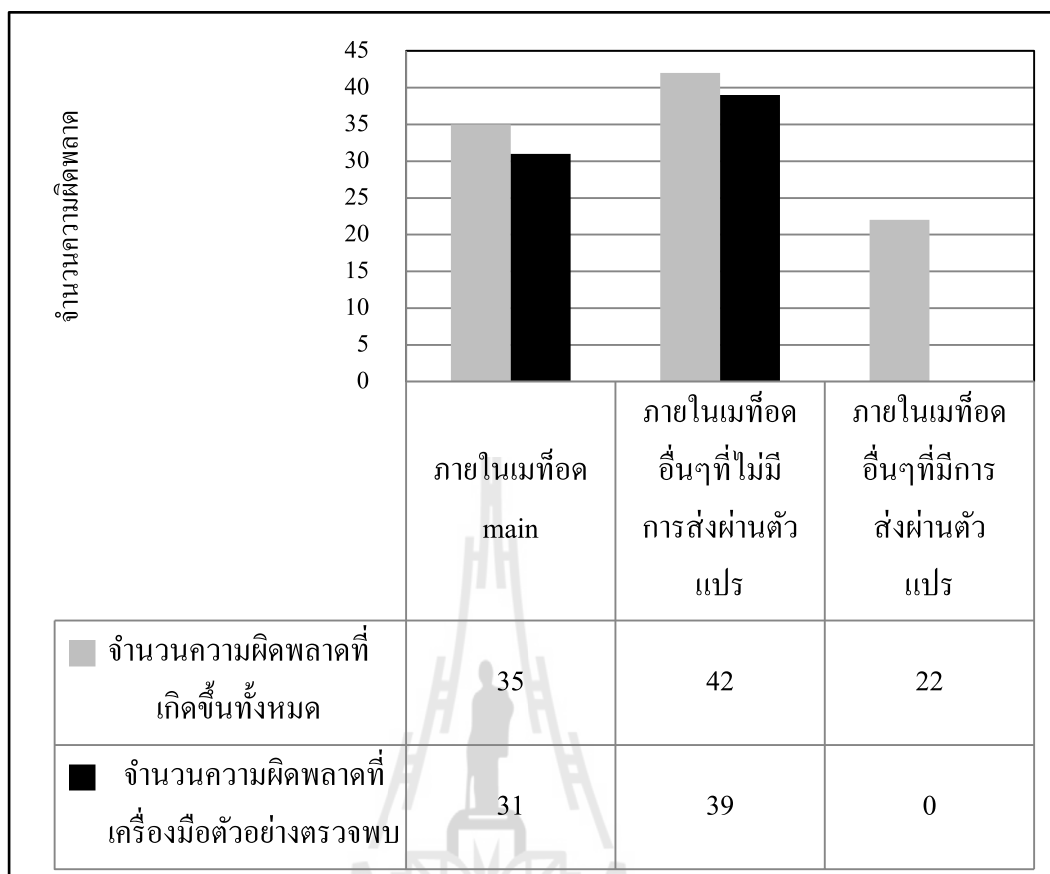
4.7 สรุปผลการทดสอบเครื่องมือตัวอย่างด้วยกรณีทดสอบ (Test case)

ผลการทดสอบเครื่องมือตัวอย่างกับความผิดพลาดทั้ง 5 ประเภทมีผลลัพธ์ดังต่อไปนี้

ตารางที่ 4.6 สรุปผลลัพธ์การทดสอบเครื่องมือตัวอย่างด้วยความผิดพลาดทั้ง 5 ประเภท

ประเภทความผิดพลาด	**จำนวนความผิดพลาดที่เกิดขึ้นจริง**	**จำนวนที่เครื่องมือตรวจพบ**	**ร้อยละความถูกต้อง**
Class cast exception	11	8	73
Array index out of bound exception	19	16	84
Arithmetic : Divided by zero exception	29	16	55
Number format exception	21	16	76
String index out of range	18	14	78

จากตารางที่ 4.6 จะเห็นว่าการทดสอบเครื่องมือตัวอย่างกับความผิดพลาดแต่ละประเภทจะมีความถูกต้องที่แตกต่างกัน โดยหากคิดรวมโดยไม่สนใจประเภทของความผิดพลาดแล้วเครื่องมือตัวอย่างนี้จะมีประสิทธิภาพความถูกต้องอยู่ที่ประมาณร้อยละ 73 แต่เนื่องจากความแตกต่างกันในความผิดพลาดแต่ละประเภทผู้วิจัยเห็นว่าการพิจารณาความถูกต้องของเครื่องมือนั้นควรแยกคิดตามประเภทของความผิดพลาดจะเหมาะสมกว่า โดยหากแยกดูร้อยละของความถูกต้องแล้วจะทำให้ทราบได้ว่าวิธีการค้นหาความผิดพลาดที่ได้นำเสนอไม่เหมาะสมกับความผิดพลาดบางประเภท ที่เห็นได้ชัดจากตารางที่ 4.6 คือความผิดพลาดประเภทที่เกิดจากการหารด้วย 0 (Arithmetic : Divided by zero exception) ซึ่งมีความถูกต้องอยู่ที่ประมาณร้อยละ 55 เท่านั้น อีกทั้งเมื่อได้ศึกษาการทำงานของไบต์โค้ดและจากผลลัพธ์การทดลองทำให้ผู้วิจัยทราบว่าเครื่องมือตัวอย่างที่สร้างขึ้นไม่สามารถตรวจสอบตัวแปรที่มีการเปลี่ยนแปลงค่าขณะโปรแกรมประมวลผลได้เพียงแต่สามารถตรวจสอบค่าคงที่ที่มีการกำหนดไว้ตอนเริ่มทำงานเท่านั้น นอกจากนี้การค้นหาความผิดพลาดในภาษาจาวาโดยใช้ไบต์โค้ดนั้นมีข้อจำกัดในเรื่องของการที่ไม่สามารถตรวจสอบตัวแปรที่มีการส่งผ่านค่าระหว่างเมท็อดได้ ดังนั้นเมื่อทำการนำเอาผลลัพธ์จำนวนความผิดพลาดมาจำแนกออกตามตำแหน่งที่เกิดความผิดพลาดแล้วข้อมูลที่ได้จะมีลักษณะดังแผนภูมิต่อไปนี้



รูปที่ 4.70 แผนภูมิแท่งแสดงจำนวนความผิดพลาดจำแนกตามตำแหน่งที่เกิดความผิดพลาด

จากแผนภูมิแท่ง ในรูปที่ 4.70 สังเกตว่าจำนวนความผิดพลาดที่เครื่องมือตัวอย่างตรวจพบจากความผิดพลาดประเภทที่เกิดขึ้นภายในเมท็อดอื่นๆที่มีการส่งผ่านตัวแปร มีค่าเท่ากับ 0 นั่นหมายถึงวิธีการตรวจหาความผิดพลาดในภาษาจาวาโดยใช้วิธีการเปรียบเทียบรูปแบบไม่สามารถค้นหาความผิดพลาดได้ในกรณีที่ความผิดพลาดเกิดขึ้นภายในเมท็อดที่มีการส่งผ่านค่าตัวแปรและตัวแปรนั้นทำให้เกิดความผิดพลาด ซึ่งนับว่าเป็นจุดอ่อนของวิธีการนี้

บทที่ 5
สรุปผลการวิจัยและข้อเสนอแนะ

การใช้ไบต์โค้ดเพื่อค้นหาความผิดพลาด (Exception) ของภาษาจาวาทำให้วิธีการทดสอบซอฟต์แวร์ในภาษาจาวาสะดวกขึ้นเป็นอย่างมาก เนื่องจากไม่จำเป็นที่ต้องใช้ซอสโค้ดในการทดสอบซึ่งในบางกรณีซอฟต์แวร์ที่ต้องการทดสอบอาจไม่เปิดเผยซอสโค้ด และถ้าหากใช้วิธีการทดสอบซอฟต์แวร์แบบปรกติ (ใช้ซอสโค้ดเป็นข้อมูลนำเข้าในการทดสอบ) ก็จะไม่สามารถกระทำการทดสอบได้แต่ถ้าหากใช้ไบต์โค้ดเป็นข้อมูลนำเข้าแล้วก็จะช่วยแก้ไขปัญหานี้ได้ โดยในงานวิจัยครั้งนี้ได้พัฒนาเครื่องมือตัวอย่างที่ใช้ไบต์โค้ดในการตรวจหาความผิดพลาดและนำมาทดสอบกับกรณีทดทอดของความผิดพลาดทั้ง 5 ประเภท (ดังที่แสดงในบทที่ 4) ซึ่งเมื่อรวมแล้วจะได้การทดสอบทั้งหมด 98 กรณีทดสอบ โดยเมื่อนำผลลัพธ์การทดสอบมาสรุปผลเพื่อหาข้อดี-ข้อเสีย และความถูกต้องแล้ว จะสามารถสรุปได้ดังหัวข้อที่ 5.1

5.1 สรุปผลการวิจัย

5.1.1 ข้อจำกัดที่พบหลังการทดสอบเครื่องมือตัวอย่าง

หลังจากทดสอบเครื่องมือตัวอย่างที่พัฒนาขึ้น ผู้วิจัยได้พบข้อจำกัดของเครื่องมือตัวอย่างซึ่งมีดังต่อไปนี้

1. เครื่องมือตัวอย่างที่พัฒนาขึ้นสามารถตรวจสอบค่าของตัวแปรที่มีลักษณะเป็นค่าคงที่ ที่ได้มีการกำหนดค่าแน่นอนไว้ในตอนเริ่มต้นเท่านั้น หากความผิดพลาดเกิดขึ้นหลังจากที่โปรแกรมเริ่มประมวลผลแล้วมีการเปลี่ยนค่าตัวแปรไปเครื่องมือตัวอย่างจะไม่สามารถตรวจพบได้ เนื่องจากเครื่องมือใช้เทคนิคการวิเคราะห์แบบคงที่ (Static analysis) ซึ่งใช้การเปรียบเทียบรูปแบบในลักษณะของข้อความที่เป็นค่าคงที่ จึงทำให้ไม่สามารถตรวจหาข้อผิดพลาดที่เกิดจากการเปลี่ยนแปลงค่าระหว่างโปรแกรมประมวลผลได้

2. เครื่องมือตัวอย่างที่สร้างขึ้นไม่สามารถตรวจหาความผิดพลาดในกรณีที่ความผิดพลาดเกิดจากตัวแปรที่มีการส่งค่าระหว่างเมทอด เนื่องจากในไบต์โค้ดเมื่อมีการส่งผ่านค่าไปยังเมทอดอื่นแล้วตำแหน่งของหน่วยจัดเก็บข้อมูลจำลองจะเปลี่ยนไป ซึ่งหากใช้วิธีการเปรียบเทียบรูปแบบแล้วจะทำให้เกิดความไม่ถูกต้องของค่าที่อ้างอิงไปยังหน่วยจัดเก็บข้อมูลจำลอง และเกิดความไม่

มหาวิทยาลัยเทคโนโลยีสุรนารี

ถูกต้องหากนำมาเปรียบเทียบกัน ดังเช่นตัวอย่างในรูปที่ 5.1 ในบรรทัดที่ 6-7 เป็นการเรียกใช้เมท็อด print โดยส่งค่าที่เก็บในหน่วยจัดเก็บข้อมูลจำลองตำแหน่งที่ 1 เป็นพารามิเตอร์ของเมท็อด แต่ในเมื่อมีการโหลดค่าเพื่อแสดงผลในเมทอด print บรรทัดที่ 11-12 จะมีการอ้างอิงถึงหน่วยเก็บข้อมูลจำลองตำแหน่งที่ 0 ซึ่งทำให้การอ้างอิงภายในเครื่องมือตัวอย่างไม่สามารถทำได้ถูกต้อง

```
public class printEx {

      public static void main(String[] args) {
            String str = "Suranaree";
            print(str);
      }
      static void print(String str){
            System.out.println(str);
      }
}
```

```
1:  public class printEx {
2:
3:  public static main([Ljava/lang/String;)V
4:     LDC "Suranaree"
5:     ASTORE 1
6:     ALOAD 1
7:     INVOKESTATIC printEx.print(Ljava/lang/String;)V
8:     RETURN
9:  static print(Ljava/lang/String;)V
10:    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
11:    ALOAD 0
12:    INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang/String;)V
13:    RETURN
14: }
```

รูปที่ 5.1 ตัวอย่างของไบต์โค้ดที่มีการส่งผ่านพารามิเตอร์ในการเรียกใช้เมท็อด

5.1.2 ข้อดี-ข้อเสียของวิธีการค้นหาความผิดพลาดด้วยการเปรียบเทียบรูปแบบ

ข้อดี

1. ใช้ไบต์โค้ดเป็นข้อมูลนำเข้าในการทดสอบโปรแกรม จึงทำให้สามารถทดสอบโปรแกรมได้ถึงแม้ไม่มีซอสโค้ดของโปรแกรม

2. เป็นเครื่องมือที่เปิดเผยซอสโค้ด จึงทำให้ผู้ที่นำไปใช้สามารถแก้ไขปรับปรุงเครื่องมือให้เหมาะสมกับงานที่จะนำไปใช้ หรือนำไปประยุกต์ใช้กับงานด้านอื่นๆได้

3. เป็นเครื่องมือที่พัฒนาด้วยภาษาจาวาทั้งหมด ทำให้ผู้ที่ศึกษาภาษาจาวาสามารถทำความเข้าใจได้ง่าย

4. สนับสนุนการทำงานทั้งบนชุดเครื่องมือพัฒนาโปรแกรม (Software development kit) Eclipse และ Netbeans เนื่องจากมีลักษณะการทำงานแบบคลังโปรแกรม (Library) ซึ่งต่างจากเครื่องมือบางตัวที่อาจสนับสนุนการทำงานบนชุดเครื่องมือพัฒนาโปรแกรมเพียงบางตัว อาทิเช่น FindBugs ที่สนับสนุนการทำงานเฉพาะบนชุดเครื่องมือพัฒนาโปรแกรม Eclipse เท่านั้น

5. เปิดเผยลักษณะของรูปแบบความผิดพลาด (Pattern) ที่ใช้ในการค้นหาความผิดพลาดและอนุญาตให้ผู้ใช้แก้ไขหรือเพิ่มรูปแบบได้ ซึ่งส่วนมากเครื่องมือที่ใช้ในการทดสอบความผิดพลาดจะไม่อนุญาตให้ผู้ใช้แก้ไขส่วนนี้ อาทิเช่น FindBugs ที่ไม่มีการเปิดเผยรูปแบบการค้นหาความผิดพลาดแก่ผู้ใช้ โดยหากผู้ใช้ต้องการเพิ่มความผิดพลาดใหม่ที่ค้นพบจะต้องทำการแจ้งไปยังเว็บไซต์ของ FindBugs เพื่อให้ทีมพัฒนาทำการเพิ่มความผิดพลาดลงในเครื่องมือ

ข้อเสีย

1. ไม่สามารถค้นหาความผิดพลาดครอบคลุมทุกกรณีทดสอบที่นำมาทดสอบ เนื่องจากยังมีข้อจำกัดของเครื่องมือตัวอย่างที่พัฒนาขึ้นซึ่งข้อจำกัดที่พบได้อธิบายไว้ในหัวข้อที่ 5.1.2

2. ไม่มีส่วนต่อประสานกับผู้ใช้ (User interface) เนื่องจากมีลักษณะเป็นคลังโปรแกรมจึงทำให้เมื่อผู้ใช้ต้องการใช้งานจะต้องเขียนซอสโค้ดอย่างน้อย 4 บรรทัดเพื่อเรียกใช้งานเครื่องมือ

3. เนื่องจากมีการเปิดเผยรูปแบบความผิดพลาดและอนุญาตให้ผู้ใช้แก้ไขหรือเพิ่มรูปแบบได้เอง จึงเกิดความเสี่ยงที่ผู้ใช้จะเพิ่มรูปแบบที่ไม่ถูกต้องเข้าไปจนทำให้เครื่องมือทำงานผิดพลาด

5.2 ข้อเสนอแนะ

งานวิจัยนี้ได้นำเสนอวิธีการตรวจหาความผิดพลาดในภาษาจาวาด้วยการใช้วิธีการเปรียบเทียบรูปแบบซึ่งสามารถนำเอาไปใช้ในการตรวจสอบความผิดพลาดในภาษาจาวาได้ แต่ในขณะนี้วิธีการยังคงมีข้อบกพร่องอยู่ ซึ่งหากผู้วิจัยท่านอื่นสามารถนำเอาไปพัฒนาแก้ไขจุดบกพร่องนี้ได้ และนอกจากนี้แนวคิดที่ผู้วิจัยได้นำเสนอนี้ยังสามารถนำเอาไปประยุกต์ใช้ซึ่งเป็นประโยชน์แก่นักวิจัยท่านอื่นที่ต้องการศึกษาเกี่ยวกับการค้นหาความผิดพลาดของภาษาจาวา อีกทั้ง

ผู้วิจัยยังมองเห็นถึงแนวทางการประยุกต์ใช้หรือวิจัยโดยใช้แนวคิดการใช้ไบต์โค้ดกับการเปรียบเทียบรูปแบบ ดังนี้

- **การประยุกต์ใช้กับไบนารีโค้ด** ในแพลตฟอร์มวินโดวส์โปรแกรมที่จะนำรันบนระบบปฏิบัติการนี้ส่วนใหญ่จะถูกคอมไพล์ให้อยู่ในไฟล์นามสกุล exe โดยภายในบรรจุ ไบนารีโค้ดเอาไว้ซึ่งไบนารีโค้ดมีคุณลักษณะที่คล้ายคลึงกับไบต์โค้ดดังนั้นจึงสามารถนำเอาวิธีการนี้ไปประยุกต์ใช้กับไบนารีโค้ดได้
- **การตรวจหาการทำงานที่ไม่พึงประสงค์ของซอฟต์แวร์** ในบางกรณีซอฟต์แวร์ภาษาจาวาที่นำมาใช้มีการประมวลผลที่อาจก่อให้เกิดความไม่ปลอดภัย หรือละเมิดความเป็นส่วนตัวของผู้ใช้ได้ ซึ่งสามารถตรวจสอบชุดคำสั่งเหล่านี้ได้โดยการประยุกต์ใช้วิธีการเปรียบเทียบรูปแบบที่ผู้วิจัยได้นำเสนอ ตัวอย่างเช่น มีโปรแกรมไม่พึงประสงค์ซึ่งทำหน้าที่เข้าถึง Registry ของเครื่องคอมพิวเตอร์เพื่อสั่งให้คอมพิวเตอร์รันโปรกรมไม่พึงประสงค์ทุกครั้งที่เปิดเครื่อง ดังซอสโค้ดในรูปที่ 5.2 โดยเมื่อทำการแปลงให้อยู่ในรูปแบบของชุดคำสั่งไบต์โค้ดแล้ว สามารถที่จะระบุถึงคำสั่งที่ไม่พึงประสงค์ได้ ดังรูปที่ 5.3 ซึ่งในชุดคำสั่งหมายเลข 9 เป็นชุดคำสั่งที่เรียกใช้คลังโปรแกรมที่ใช้จัดการการเข้าถึง Registry และ ชุดคำสั่งหมายเลข 9 เป็นตัวแปรที่ใช้ในการเข้าถึงซึ่งเป็นตำแหน่งของ Registry ที่ทำให้เกิดการรันโปรแกรมทุกครั้งเมื่อเปิดเครื่องซึ่งสามารถนำเอาไบต์โค้ดสองส่วนนี้มาใช้เป็นไบต์โค้ดต้นแบบในการค้นหาไบต์โค้ดโดยประยุกต์เครื่องมือ JEPM ซึ่งผลลัพธ์ที่ได้ แสดงดังรูปที่ 5.4
- **การตรวจหาซอฟต์แวร์ที่เป็นอันตราย** ในที่นี้หมายถึงไวรัสคอมพิวเตอร์ซึ่งในปัจจุบันได้พัฒนาให้สามารถตรวจสอบได้ยากขึ้น หากนำเอารูปแบบการทำงานที่จำเพาะบางส่วนของไวรัสมาเป็นข้อมูลต้นแบบและใช้วิธีการตรวจสอบรูปแบบที่ผู้วิจัยได้นำเสนอไปแล้วนั้นมาประยุกต์ใช้งานก็จะช่วยให้สามารถระบุซอฟต์แวร์ที่เป็นอันตรายได้

```
1  import javaQuery.core.RKey;
2  import javaQuery.core.jqReg;
3  import javaQuery.core.keyType;
4
5  public class mulwEx {
6      public static void main(String args[]) {
7          jqReg _jr = new jqReg();
8          String path = "\\Microsoft\\Windows\\CurrentVersion\\Run";
9          String res = _jr.jqReg(RKey.HKEY_LOCAL_MACHINE_SOFTWARE, path,
10                 keyType.String_Value, "runMe",
11                 "C:\\Windows\\System32\\system.exe");
12     }
13 }
```

รูปที่ 5.2 ตัวอย่างของซอสโค้ดที่มีคำสั่งไม่พึงประสงค์

ID	CODE

5	aload_0
	invokespecial #8; //Method java/lang/Object."<init>":()V
	return

7	new #16; //class javaQuery/core/jqReg
	dup
	invokespecial #18; //Method javaQuery/core/jqReg."<init>":()V
	astore_1

8	ldc #19; //String \Microsoft\Windows\CurrentVersion\Run
	astore_2

9	invokevirtual #29; //Method
	javaQuery/core/jqReg.jqReg:(Ljava/lang/String;Ljava/lang/String;Ljava/lang/
	String;Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
	astore_3

9	aload_1
	ldc #21; //String HKEY_LOCAL_MACHINE\SOFTWARE\
	aload_2

10	ldc #23; //String String
	ldc #25; //String runMe

11	ldc #27; //String C:\Windows\System32\system.exe
12	return

รูปที่ 5.3 ตัวอย่างของไบต์โค้ดที่มีคำสั่งไม่พึงประสงค์

```
*TestJEPM.java
1 import testCase.*;
3
4 public class TestJEPM {
5     public static void main(String args[]) {
6         JEPM x = new JEPM(new mulwEx());
7         x.test();
8     }
9 }
```

```
<terminated> TestJEPM [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (20 ส.ค. 2555, 4:37:5
Package Name     : mulwEx
Method Name      : main(java.lang.String[]);
>>Line 9 : "Mulware : Registry autorun.".
```

รูปที่ 5.4 ตัวอย่างการประยุกต์เครื่องมือ JEPM กับโปรแกรมไม่พึงประสงค์

รายการอ้างอิง

Gupta, R. (2006). **Java Virtual Machine** [On-line]. Available : http://www.boloji.com /index .cfm?md=Content&sd=Articles&ArticleID=549

Haggar, P. (2006). Java bytecode : **Understanding bytecode makes you a better programmer** [On-line]. Available : http://www.ibm.com/developerworks/ibm/library/it-haggar_byte code/

Harrison, T. (2006). **Java bytecode instruction listings** [On-line]. Available : http://en.wikipedia .org/wiki/Java_bytecode_instruction_listings

Hovemeyer, D., B. Pugh, et al. (2012**). FindBugs™ - Find Bugs in Java Programs** [On-line]. Available : http://findbugs.sourceforge.net/

Humbad, S. N. (2004). **Perl Regular Expressions by Example** [On-line]. Available : http://www .somacon.com/p127.php

Lievens, W. (2006). **Java bytecode** [On-line]. Available : http://en.wikipedia.org/wiki/Java_byte code

Muller, T. (2006). **H2 Database Engine** [On-line]. Available : http://www.h2database.com/ h2.pdf

Muller, T. (2006). **H2 Database Engine Features** [On-line]. Available : http://www.h2database .com/html/features.html

Oracle Inc. (2004). **javap - The Java Class File Disassembler** [On-line]. Available : http://docs .oracle.com/javase/1.5.0/docs/tooldocs/windows/javap.html.

Schwartz, R., T. Christiansen, et al. (1997). **Learning Perl** [On-line]. Available : http://docstore .mik.ua/orelly/perl/learn/ch07_03.htm.

Sun Microsystems, Inc. (2005). **Understanding Checked and Unchecked Exceptions in Java** [On-line]. Available : http://www.jforeach.com/understanding-checked-and-unchecked-exceptions-in-java/59

Sun Microsystems, Inc. (2006). **The Java Programming Language** [On-line]. Available : https: //www.cs.auckland.ac.nz/references/java/java1.5/tutorial/getStarted/intro/definition.html

Albert, E., M. Gómez-Zamalloa, et al. (2007). **Verification of Java Bytecode Using Analysis and Transformation of Logic Programs Practical Aspects of Declarative Languages**. Springer Berlin / Heidelberg: 124-139.

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman. **Compilers: Principles, Techniques and Tools**, Massachusetts: Addison-Wesley, March 1986.

Cabral, B. and P. Marques (2008). A Case for Automatic Exception Handling. **Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering**. IEEE Computer Society: 403-406.

Foster, J. S., M. W. Hicks, et al. (2007). Improving software quality with static analysis. **Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering**. San Diego, California, USA, ACM: 83-84.

Hamid, N. A. (2009). Pattern matching on objects in Java. **J. Comput. Small Coll.** 25(1): 51-57.

Hovemeyer, D. and W. Pugh (2004). Finding bugs is easy. **Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.** Vancouver, BC, CANADA, ACM: 132-136.

Kiczales, G. and M. Mezini (2004). Aspect-Oriented Programming and Modular Reasoning. **27th international conference on Software engineering**. New York, USA, ACM: 49-58.

Lance, D., R. H. Untch, et al. (1999). Bytecode-based Java program analysis. In **Proceedings of the 37th annual Southeast regional conference**. ACM: 14.

Lippert, M. and C. V. Lopes (2000). A study on exception detection and handling using aspect-oriented programming. In **Proceedings of the 22nd international conference on Software engineering**. Limerick, Ireland, ACM: 418-427.

Murphy, C., E. Kim, et al. (2008). Backstop : A Tool for Debugging Runtime Errors. **SIGCSE'8**. Columbia University: 173-177.

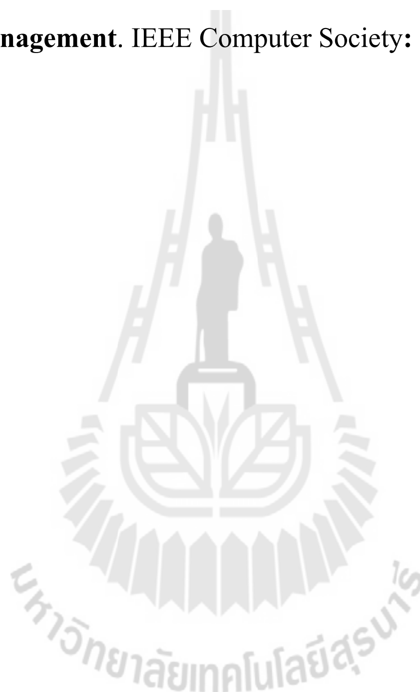
Pugh, W. (2007). Improving Software Quality with Static Analysis. In **2007 JavaOneSM Conference**. University of Maryland.

Robillard, M. P. and G. C. Murphy (2003). Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems. **ACM Transactions on Software Engineering and Methodology**. University of British Columbia. Vol. 12: 191-211.

Rutar, N., C. B. Almazan, et al. (2004). A Comparison of Bug Finding Tools for Java. In **Proceedings of the 15th International Symposium on Software Reliability Engineering**. IEEE Computer Society: 245-256.

Sinha, S., H. Shah, et al. (2009). Fault localization and repair for Java runtime exceptions. In **Proceedings of the eighteenth international symposium on Software testing and analysis**. Chicago, IL, USA, ACM: 153-164.

Zhao, G., H. Chen, et al. (2008). Data-Flow Based Analysis of Java Bytecode Vulnerability. In **Proceedings of the 2008 The Ninth International Conference on Web-Age Information Management**. IEEE Computer Society: 647-653.



ภาคผนวก ก

ตารางคำสั่งไบต์โค้ดในภาษาจาวา

Java Bytecode instruction table (Harrison, T., 2006)

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
		A		
aaload	32		arrayref, index → value	loads onto the stack a reference from an array
aastore	53		arrayref, index, value →	stores into a reference to an array
aconst_null	01		→ null	pushes a *null* reference onto the stack
aload	19	index	→ objectref	loads a reference onto the stack from a local variable *#index*
aload_0	2a		→ objectref	loads a reference onto the stack from local variable 0
aload_1	2b		→ objectref	loads a reference onto the stack from local variable 1
aload_2	2c		→ objectref	loads a reference onto the stack from local variable 2
aload_3	2d		→ objectref	loads a reference onto the stack from local variable 3
anewarray	bd	indexbyte1, indexbyte2	count → arrayref	creates a new array of references of length *count* and component type identified by the class reference *index* (*indexbyte1 << 8 + indexbyte2*) in the constant pool

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** [before]→[after]	**Description**
areturn	b0		objectref → [empty]	returns a reference from a method
arraylength	be		arrayref → length	gets the length of an array
astore	3a	index	objectref →	stores a reference into a local variable *#index*
astore_0	4b		objectref →	stores a reference into local variable 0
astore_1	4c		objectref →	stores a reference into local variable 1
astore_2	4d		objectref →	stores a reference into local variable 2
astore_3	4e		objectref →	stores a reference into local variable 3
athrow	bf		objectref → [empty], objectref	throws an error or exception

Mnemonic	**Opcode** ***(in hex)***	**Other bytes**	**Stack** **[before]→[after]**	**Description**
		B		
baload	33		arrayref, index → value	loads a byte or Boolean value from an array
bastore	54		arrayref, index, value →	stores a byte or Boolean value into an array
bipush	10	byte	→ value	pushes a *byte* onto the stack as an integer *value*
		C		
caload	34		arrayref, index → value	loads a char from an array
castore	55		arrayref, index, value →	stores a char into an array
checkcast	c0	indexbyte1, indexbyte2	objectref → objectref	checks whether an *objectref* is of a certain type, the class reference of which is in the constant pool at *index* (*indexbyte1 << 8 + indexbyte2*)

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
		D		
d2f	90		value → result	converts a double to a float
d2i	8e		value → result	converts a double to an int
d2l	8f		value → result	converts a double to a long
dadd	63		value1, value2 → result	adds two doubles
daload	31		arrayref, index → value	loads a double from an array
dastore	52		arrayref, index, value →	stores a double into an array
dcmpg	98		value1, value2 → result	compares two doubles
dcmpl	97		value1, value2 → result	compares two doubles
dconst_0	0e		→ 0.0	pushes the constant *0.0* onto the stack
dconst_1	0f		→ 1.0	pushes the constant *1.0* onto the stack
ddiv	6f		value1, value2 → result	divides two doubles
dload	18	index	→ value	loads a double *value* from a local variable *#index*
dload_0	26		→ value	loads a double from local variable 0
dload_1	27		→ value	loads a double from local variable 1

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** [before]→[after]	**Description**
dload_2	28		→ value	loads a double from local variable 2
dload_3	29		→ value	loads a double from local variable 3
dmul	6b		value1, value2 → result	multiplies two doubles
dneg	77		value → result	negates a double
drem	73		value1, value2 → result	gets the remainder from a division between two doubles
dreturn	af		value → [empty]	returns a double from a method
dstore	39	index	value →	stores a double *value* into a local variable *#index*
dstore_0	47		value →	stores a double into local variable 0
dstore_1	48		value →	stores a double into local variable 1
dstore_2	49		value →	stores a double into local variable 2
dstore_3	4a		value →	stores a double into local variable 3
dsub	67		value1, value2 → result	subtracts a double from another
dup	59		value → value, value	duplicates the value on top of the stack

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** **[before]⟶[after]**	**Description**
dup_x1	5a		value2, value1 ⟶ value1, value2, value1	inserts a copy of the top value into the stack two values from the top
dup_x2	5b		value3, value2, value1 ⟶ value1, value3, value2, value1	inserts a copy of the top value into the stack two (if value2 is double or long it takes up the entry of value3, too) or three values (if value2 is neither double nor long) from the top
dup2	5c		{value2, value1} ⟶ {value2, value1}, {value2, value1}	duplicate top two stack words (two values, if value1 is not double nor long; a single value, if value1 is double or long)
dup2_x1	5d		value3, {value2, value1} ⟶ {value2, value1}, value3, {value2, value1}	duplicate two words and insert beneath third word (see explanation above)

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]⟶[after]	Description
dup2_x2	5e		{value4, value3}, {value2, value1} ⟶ {value2, value1}, {value4, value3}, {value2, value1}	duplicate two words and insert beneath fourth word
		F		
f2d	8d		value ⟶ result	converts a float to a double
f2i	8b		value ⟶ result	converts a float to an int
f2l	8c		value ⟶ result	converts a float to a long
fadd	62		value1, value2 ⟶ result	adds two floats
faload	30		arrayref, index ⟶ value	loads a float from an array
fastore	51		arreyref, index, value ⟶	stores a float in an array
fcmpg	96		value1, value2 ⟶ result	compares two floats
fcmpl	95		value1, value2 ⟶ result	compares two floats
fconst_0	0b		⟶ 0.0f	pushes *0.0f* on the stack
fconst_1	0c		⟶ 1.0f	pushes *1.0f* on the stack

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack [before]⟶[after]**	**Description**
fconst_2	0d		⟶ 2.0f	pushes *2.0f* on the stack
fdiv	6e		value1, value2 ⟶ result	divides two floats
fload	17	index	⟶ value	loads a float *value* from a local variable *#index*
fload_0	22		⟶ value	loads a float *value* from local variable 0
fload_1	23		⟶ value	loads a float *value* from local variable 1
fload_2	24		⟶ value	loads a float *value* from local variable 2
fmul	6a		value1, value2 ⟶ result	multiplies two floats
fneg	76		value ⟶ result	negates a float
frem	72		value1, value2 ⟶ result	gets the remainder from a division between two floats
freturn	ae		value ⟶ [empty]	returns a float
fstore	38	index	value ⟶	stores a float *value* into a local variable *#index*
fstore_0	43		value ⟶	stores a float *value* into local variable 0
fstore_1	44		value ⟶	stores a float *value* into local variable 1
fstore_2	45		value ⟶	stores a float *value* into local variable 2
fsub	66		value1, value2 ⟶ result	subtracts two floats

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
		G		
getfield	b4	index1, index2	objectref → value	gets a field *value* of an object *objectref*, where the field is identified by field reference in the constant pool *index* (*index1 << 8 + index2*)
goto	a7	branchbyte1, branchbyte2	[no change]	goes to another instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
goto_w	c8	branchbyte1, branchbyte2, branchbyte3, branchbyte4	[no change]	goes to another instruction at *branchoffset* (signed int constructed from unsigned bytes *branchbyte1 << 24 +* branchbyte2 << 16 + *branchbyte3 << 8 + branchbyte4*)
		I		
i2b	91		value → result	converts an int into a byte
i2c	92		value → result	converts an int into a character
i2d	87		value → result	converts an int into a double
i2f	86		value → result	converts an int into a float
i2l	85		value → result	converts an int into a long
i2s	93		value → result	converts an int into a short
iadd	60		value1, value2 → result	adds two ints together

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** [before]⟶[after]	**Description**
iaload	2e		arrayref, index ⟶ value	loads an int from an array
iand	7e		value1, value2 ⟶ result	performs a bitwise and on two integers
iastore	4f		arrayref, index, value ⟶	stores an int into an array
iconst_m1	02		⟶ -1	loads the int value -1 onto the stack
iconst_0	03		⟶ 0	loads the int value 0 onto the stack
iconst_1	04		⟶ 1	loads the int value 1 onto the stack
iconst_2	05		⟶ 2	loads the int value 2 onto the stack
iconst_3	06		⟶ 3	loads the int value 3 onto the stack
iconst_4	07		⟶ 4	loads the int value 4 onto the stack
iconst_5	08		⟶ 5	loads the int value 5 onto the stack
idiv	6c		value1, value2 ⟶ result	divides two integers
if_acmpeq	a5	branchbyte1, branchbyte2	value1, value2 ⟶	if references are equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)

Mnemonic	**Opcode** ***(in hex)***	**Other bytes**	**Stack** **[before]→[after]**	**Description**
if_acmpne	a6	branchbyte1, branchbyte2	value1, value2 →	if references are not equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
if_icmpeq	9f	branchbyte1, branchbyte2	value1, value2 →	if ints are equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
if_icmpne	a0	branchbyte1, branchbyte2	value1, value2 →	if ints are not equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** [before]⟶[after]	**Description**
if_icmplt	a1	branchbyte1, branchbyte2	value1, value2 ⟶	if *value1* is less than *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)
if_icmpge	a2	branchbyte1, branchbyte2	value1, value2 ⟶	if *value1* is greater than or equal to *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)
if_icmpgt	a3	branchbyte1, branchbyte2	value1, value2 ⟶	if *value1* is greater than *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** [before]⟶[after]	**Description**
if_icmple	a4	branchbyte1, branchbyte2	value1, value2 ⟶	if *value1* is less than or equal to *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)
ifeq	99	branchbyte1, branchbyte2	value ⟶	if *value* is 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)
ifne	9a	branchbyte1, branchbyte2	value ⟶	if *value* is not 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)
iflt	9b	branchbyte1, branchbyte2	value ⟶	if *value* is less than 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)
ifge	9c	branchbyte1, branchbyte2	value ⟶	if *value* is greater than or equal to 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1* << *8* + *branchbyte2*)

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** [before]→[after]	**Description**
ifge	9c	branchbyte1, branchbyte2	value →	if *value* is greater than or equal to 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifgt	9d	branchbyte1, branchbyte2	value →	if *value* is greater than 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifle	9e	branchbyte1, branchbyte2	value →	if *value* is less than or equal to 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifnonnull	c7	branchbyte1, branchbyte2	value →	if *value* is not null, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)
ifnull	c6	branchbyte1, branchbyte2	value →	if *value* is null, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*)

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** ***(in hex)***	**Other bytes**	**Stack [before]→[after]**	**Description**
iinc	84	index, const	[No change]	increment local variable *#index* by signed byte *const*
iload	15	index	→ value	loads an int *value* from a variable *#index*
iload_0	1a		→ value	loads an int *value* from variable 0
iload_1	1b		→ value	loads an int *value* from variable 1
iload_2	1c		→ value	loads an int *value* from variable 2
iload_3	1d		→ value	loads an int *value* from variable 3
imul	68		value1, value2 → result	multiply two integers
ineg	74		value → result	negate int
instanceof	c1	indexbyte1, indexbyte2	objectref → result	determines if an object *objectref* is of a given type, identified by class reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)
invokeinterface	b9	indexbyte1, indexbyte2, count, 0	objectref, [arg1, arg2, ...] →	invokes an interface method on object *objectref*, where the interface method is identified by method reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode *(in hex)***	**Other bytes**	**Stack [before]⟶[after]**	**Description**
invokespecial	b7	indexbyte1, indexbyte2	objectref, [arg1, arg2, ...] ⟶	invoke instance method on object *objectref*, where the method is identified by method reference *index* in constant pool (*indexbyte1* << *8* + *indexbyte2*)
invokestatic	b8	indexbyte1, indexbyte2	[arg1, arg2, ...] ⟶	invoke a static method, where the method is identified by method reference *index* in constant pool (*indexbyte1* << *8* + *indexbyte2*)
invokevirtual	b6	indexbyte1, indexbyte2	objectref, [arg1, arg2, ...] ⟶	invoke virtual method on object *objectref*, where the method is identified by method reference *index* in constant pool (*indexbyte1* << *8* + *indexbyte2*)
ior	80		value1, value2 ⟶ result	bitwise int or
irem	70		value1, value2 ⟶ result	logical int remainder
ireturn	ac		value ⟶ [empty]	returns an integer from a method
ishl	78		value1, value2 ⟶ result	int shift left
ishr	7a		value1, value2 ⟶ result	int arithmetic shift right
istore	36	index	value ⟶	store int *value* into variable *#index*

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
istore_0	3b		value →	store int *value* into variable 0
istore_1	3c		value →	store int *value* into variable 1
istore_2	3d		value →	store int *value* into variable 2
istore_3	3e		value →	store int *value* into variable 3
isub	64		value1, value2 → result	int subtract
iushr	7c		value1, value2 → result	int logical shift right
ixor	82		value1, value2 → result	int xor
			J	
jsr	a8	branchbyte1, branchbyte2	→ address	jump to subroutine at *branchoffset* (signed short constructed from unsigned bytes *branchbyte1 << 8 + branchbyte2*) and place the return address on the stack
jsr_w	c9	branchbyte1, branchbyte2, branchbyte3, branchbyte4	→ address	jump to subroutine at *branchoffset* (signed int constructed from unsigned bytes *branchbyte1 << 24 + branchbyte2 << 16 + branchbyte3 << 8 + branchbyte4*) and place the return address on the stack

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
		L		
l2d	8a		value → result	converts a long to a double
l2f	89		value → result	converts a long to a float
l2i	88		value → result	converts a long to a int
ladd	61		value1, value2 → result	add two longs
laload	2f		arrayref, index → value	load a long from an array
land	7f		value1, value2 → result	bitwise and of two longs
lastore	50		arrayref, index, value →	store a long to an array
lcmp	94		value1, value2 → result	compares two longs values
lconst_0	09		→ 0L	pushes the long 0 onto the stack
lconst_1	0a		→ 1L	pushes the long 1 onto the stack
ldc	12	index	→ value	pushes a constant *#index* from a constant pool (String, int or float) onto the stack
ldc_w	13	indexbyte1, indexbyte2	→ value	pushes a constant *#index* from a constant pool (String, int or float) onto the stack (wide *index* is constructed as *indexbyte1 << 8 + indexbyte2*)

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** [before]→[after]	**Description**
ldc2_w	14	indexbyte1, indexbyte2	→ value	pushes a constant *#index* from a constant pool (double or long) onto the stack (wide *index* is constructed as *indexbyte1 << 8* + *indexbyte2*)
ldiv	6d		value1, value2 → result	divide two longs
lload	16	index	→ value	load a long value from a local variable *#index*
lload_0	1e		→ value	load a long value from a local variable 0
lload_1	1f		→ value	load a long value from a local variable 1
lload_2	20		→ value	load a long value from a local variable 2
lload_3	21		→ value	load a long value from a local variable 3
lmul	69		value1, value2 → result	multiplies two longs
lneg	75		value → result	negates a long

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack** [before]→[after]	**Description**
lookupswitch	ab	<0-3 bytes padding>, defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, npairs1, npairs2, npairs3, npairs4, match-offset pairs...	key →	a target address is looked up from a table using a key and execution continues from the instruction at that address
lor	81		value1, value2 → result	bitwise or of two longs
lrem	71		value1, value2 → result	remainder of division of two longs
lreturn	ad		value → [empty]	returns a long value
lshl	79		value1, value2 → result	bitwise shift left of a long *value1* by *value2* positions
lshr	7b		value1, value2 → result	bitwise shift right of a long *value1* by *value2* positions
lstore	37	index	value →	store a long *value* in a local variable *#index*
lstore_0	3f		value →	store a long *value* in a local variable 0
lstore_1	40		value →	store a long *value* in a local variable 1

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack [before]→[after]**	**Description**
lstore_2	41		value →	store a long *value* in a local variable 2
lstore_3	42		value →	store a long *value* in a local variable 3
lsub	65		value1, value2 → result	subtract two longs
lushr	7d		value1, value2 → result	bitwise shift right of a long *value1* by *value2* positions, unsigned
lxor	83		value1, value2 → result	bitwise exclusive or of two longs
			M	
monitorenter	c2		objectref →	enter monitor for object ("grab the lock" - start of synchronized() section)
monitorexit	c3		objectref →	exit monitor for object ("release the lock" - end of synchronized() section)
multianewarray	c5	indexbyte1, indexbyte2, dimensions	count1, [count2,...] → arrayref	create a new array of *dimensions* dimensions with elements of type identified by class reference in constant pool *index* (*indexbyte1* << *8* + *indexbyte2*); the sizes of each dimension is identified by *count1*, [*count2*, etc]

มหาวิทยาลัยเทคโนโลยีสุรนารี

Mnemonic	**Opcode** *(in hex)*	**Other bytes**	**Stack [before]⟶[after]**	**Description**
		N		
new	bb	indexbyte1, indexbyte2	⟶ objectref	creates new object of type identified by class reference in constant pool *index* (*indexbyte1 << 8 + indexbyte2*)
newarray	bc	atype	count ⟶ arrayref	creates new array with *count* elements of primitive type identified by *atype*
nop	00		[No change]	performs no operation
		P		
pop	57		value ⟶	discards the top value on the stack
pop2	58		{value2, value1} ⟶	discards the top two values on the stack (or one value, if it is a double or long)
putfield	b5	indexbyte1, indexbyte2	objectref, value ⟶	set field to *value* in an object *objectref*, where the field is identified by a field reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)
putstatic	b3	indexbyte1, indexbyte2	value ⟶	set static field to *value* in a class, where the field is identified by a field reference *index* in constant pool (*indexbyte1 << 8 + indexbyte2*)

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
		R		
ret	a9	index	[No change]	continue execution from address taken from a local variable *#index* (the asymmetry with jsr is intentional)
return	b1		→ [empty]	return void from method
		S		
saload	35		arrayref, index → value	load short from array
sastore	56		arrayref, index, value →	store short to array
sipush	11	byte1, byte2	→ value	pushes a signed integer (*byte1 << 8 + byte2*) onto the stack
swap	5f		value2, value1 → value1, value2	swaps two top words on the stack (note that value1 and value2 must not be double or long)

Mnemonic	**Opcode** ***(in hex)***	**Other bytes**	**Stack** **[before]→[after]**	**Description**
		T		
tableswitch	aa	[0-3 bytes padding], defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, lowbyte1, lowbyte2, lowbyte3, lowbyte4, highbyte1, highbyte2, highbyte3, highbyte4, jump offsets...	index →	continue execution from an address in the table at offset *index*
		W		
wide	c4	opcode, indexbyte1, indexbyte2 or iinc, indexbyte1, indexbyte2, countbyte1, countbyte2	[same as for corresponding instructions]	execute *opcode*, where *opcode* is either iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore, or ret, but assume the *index* is 16 bit; or execute iinc, where the *index* is 16 bits and the constant to increment by is a signed 16 bit short

Mnemonic	Opcode *(in hex)*	Other bytes	Stack [before]→[after]	Description
		Unused		
breakpoint	ca			reserved for breakpoints in Java debuggers; should not appear in any class file
impdep1	fe			reserved for implementation-dependent operations within debuggers; should not appear in any class file
impdep2	ff			reserved for implementation-dependent operations within debuggers; should not appear in any class file
(no name)	cb-fd			these values are currently unassigned for opcodes and are reserved for future use
xxxunusedxxx	ba			this opcode is reserved "for historical reasons"

มหาวิทยาลัยเทคโนโลยีสุรนารี

ภาคผนวก ข

สรุปรายการกรณีทดสอบ (Test case) ที่ใช้ทดสอบเครื่องมือตัวอย่าง JEPM 1.0

มหาวิทยาลัยเทคโนโลยีสุรนารี

สรุปรายการกรณีทดสอบ (Test case) ที่ใช้ทดสอบเครื่องมือตัวอย่าง JEPM 1.0

1. ความผิดพลาดประเภทที่เกิดจากการแปลงวัตถุของภาษาจาวาอย่างไม่ถูกต้อง

1.1 การทดสอบด้วยตัวแปรระดับคลาส (Class variable)

1.1.1 เกิดความผิดพลาดขึ้นภายในเมท็อด main

1.1.2 เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main

1.1.3 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

1.1.4 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

1.2 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร

1.2.1 เกิดความผิดพลาดขึ้นภายในเมท็อด main

1.2.2 เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main

1.2.3 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

1.2.4 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

1.3 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร

1.3.1 เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main

1.3.2 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

1.3.3 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

2. ความผิดพลาดประเภทที่เกิดการเรียกใช้อาร์เรย์เกินขอบเขตที่กำหนดในภาษาจาวา

2.1 การทดสอบด้วยตัวแปรระดับคลาส (Class variable)

2.1.1 เกิดความผิดพลาดขึ้นภายในเมท็อด main

2.1.2 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยการเรียกใช้อาร์เรย์อย่างง่าย

2.1.3 เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main

2.1.4 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

2.1.5 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

2.2 การทดสอบด้วยตัวแปรเฉพาะที่ (Local) ที่ไม่มีการส่งผ่านตัวแปรระหว่างเรียกใช้เมท็อด

2.2.1 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยการเรียกใช้อาร์เรย์อย่างง่ายเกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main

2.2.2 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน

2.2.3 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายในเมท็อด

2.2.4 เกิดความผิดพลาดขึ้นในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main

2.2.5 เกิดความผิดพลาดขึ้นในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน

2.2.6 เกิดความผิดพลาดขึ้นในเมท็อดอื่นที่ถูกเรียกใช้โดยเมท็อด main โดยการเรียกใช้อาร์เรย์ผ่านตัวแปรภายในลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายในเมท็อด

2.2.7 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

2.2.8 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

2.2.9 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน

2.2.10 เกิดความผิดพลาดขึ้นจากเมท็อด main มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานอย่างชัดเจน

2.2.11 เกิดความผิดพลาดขึ้นจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายใน

2.3 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร

2.3.1 เกิดความผิดพลาดขึ้นจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง โดยใช้ตัวแปรจากลูป for ที่มีการกำหนดจำนวนรอบการทำงานจากตัวแปรภายใน

2.3.2 เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main

2.3.3 เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

2.3.4 เกิดความผิดพลาดภายในเมท็อดอื่นที่เรียกใช้โดยเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

3. ความผิดพลาดประเภทที่เกิดจากการหารด้วยศูนย์

3.1 การทดสอบด้วยตัวแปรระดับคลาส (Class variable)

3.1.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main

3.1.2 เกิดความผิดพลาดขึ้นภายในเมท็อด main เนื่องจากตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์

3.1.3 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ

3.1.4 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ

3.1.5 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

3.1.6 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ

3.1.7 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ

3.1.8 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

3.1.9 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ

3.1.10 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ

3.1.11 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

มหาวิทยาลัยเทคโนโลยีสุรนารี

3.2 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร

3.2.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main

3.2.2 เกิดความผิดพลาดขึ้นภายในเมท็อด main เนื่องจากตัวแปรที่เป็นตัวหารมีค่าเป็นศูนย์

3.2.3 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ

3.2.4 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ

3.2.5 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

3.2.6 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ

3.2.7 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ

3.2.8 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

3.2.9 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ

3.2.10 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และ ตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ

3.2.11 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

3.3 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร

3.3.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

3.3.2 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจากการวนรอบ

3.3.3 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่เป็นตัวหารมีการลดค่าจากการวนรอบ

มหาวิทยาลัยเทคโนโลยีสุรนารี

3.3.4 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

3.3.5 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการเพิ่มค่าจาการวนรอบ

3.3.6 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่เป็นตัวหารมีการลดค่าจาการวนรอบ

3.3.7 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

4. ความผิดพลาดประเภทที่เกิดจากการจัดรูปแบบตัวเลขที่ไม่ถูกต้อง

4.1 การทดสอบด้วยตัวแปรระดับคลาส (Class variable)

4.1.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main

4.1.2 เกิดความผิดพลาดขึ้นภายในเมท็อด main เนื่องจากตัวแปรที่นำมาจัดรูปแบบมีค่าไม่ถูกต้อง

4.1.3 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง

4.1.4 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

4.1.5 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และ ตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง

4.1.6 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

4.1.7 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และ ตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง

4.1.8 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

4.2 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร

4.2.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main

4.2.2 เกิดความผิดพลาดขึ้นภายในเมท็อด main เนื่องจากตัวแปรที่นำมาจัดรูปแบบมีค่าไม่ถูกต้อง

4.2.3 เกิดความผิดพลาดขึ้นภายในเมท็อด main โดยตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง

4.2.4 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

4.2.5 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง

4.2.6 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

4.2.7 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง

4.2.8 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

4.3 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร

4.3.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

4.3.2 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main และตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง

4.3.3 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

4.3.4 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง และตัวแปรที่นำมาจัดรูปแบบเป็นผลลัพธ์จากการต่อสตริง

4.3.5 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

มหาวิทยาลัยเทคโนโลยีสุรนารี

5. **ความผิดพลาดที่เกิดจากการใช้งานสตริงเกินขอบเขตที่กำหนด**

5.1 การทดสอบด้วยตัวแปรระดับคลาส (Class variable)

5.1.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main

5.1.2 เกิดความผิดพลาดขึ้นภายในเมท็อด main เนื่องจากการตัดสตริงไม่ถูกต้อง

5.1.3 เกิดความผิดพลาดภายในเมท็อด main โดยเกิดความผิดพลาดขึ้นภายในลูป for

5.1.4 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

5.1.5 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main จากการตัดสตริงภายในลูป for

5.1.6 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

5.1.7 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

5.2 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่ไม่มีการส่งผ่านตัวแปร

5.2.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อด main

5.2.2 เกิดความผิดพลาดขึ้นภายในเมท็อด main เนื่องจากการตัดสตริงไม่ถูกต้อง

5.2.3 เกิดความผิดพลาดขึ้นภายในเมท็อด main จากการตัดสตริงภายในลูป for

5.2.4 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

5.2.5 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main จากการตัดสตริงภายในลูป for

5.2.6 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

5.2.7 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

5.3 การทดสอบด้วยตัวแปรเฉพาะที่ (Local variable) ที่มีการส่งผ่านตัวแปร

5.3.1 เกิดความผิดพลาดอย่างง่ายขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main

5.3.2 เกิดความผิดพลาดขึ้นภายในเมท็อดที่ถูกเรียกใช้โดยเมท็อด main จากการตัดสตริงภายในลูป for

5.3.3 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 1 ครั้ง

5.3.4 เกิดความผิดพลาดจากเมท็อด main ที่มีการเรียกเมท็อดย่อยซ้อนจำนวน 2 ครั้ง

มหาวิทยาลัยเทคโนโลยีสุรนารี

ภาคผนวก ค

บทความทางวิชาการที่ได้รับการตีพิมพ์เผยแพร่ในระหว่างศึกษา

มหาวิทยาลัยเทคโนโลยีสุรนารี

รายชื่อบทความที่ได้รับการตีพิมพ์เผยแพร่ในระหว่างศึกษา

S. Sonnum and P. Mahatthanapiwat (2012), **A Structured Analysis Approach for Java Programming Using SQL,** The Fourth International Conference on Science and Technology for Sustainable Development of the Greater Mekong Sub-region. Khon Kaen, Thailand. January 23-24, 2012. 6 pp.

A Structured Analysis Approach for Java Program Using SQL

Sarawuth Sonnum, Pichayotai Mahatthanapiwat

(School of Computer Engineering, Suranaree University of Technology)

Abstract

Program analysis is the process for analyzing a program's behavior. Many techniques have been developed over the years to analyze programs. Program analysis techniques are usually complex undertaking or use complicated techniques. Motivated by this problem, this paper proposes a non-complicated technique for Java program analysis. Furthermore, it develops the simple approach that can be adapted by using a Java Bytecode and SQL together. This paper focuses on Java Bytecode structure analysis with partition the Java Bytecode component to be accessible and searchable by SQL. Eventually, when the structure of a Java program can be accessed via SQL, It will be useful and provide expediently for the field of Java program analysis.

KEYWORDS: Java, Bytecode, analysis, SQL

Introduction

The process of program analysis can be divided into 2 major fields, dynamic program analysis and static program analysis. The dynamic program analysis is the process for analyzing the behavior of programs running. The static program analysis uses source code to determine the characteristic of the program. This paper deals with static program analysis because this approach is less complicated than dynamic program analysis. Therefore, the static program analysis can be applied for this research and explained to people easily.

Often, a program analysis tool can be used to determine the abnormal of program by program testers. In addition, the benefits of program analysis tool include time saving, cost saving and analyzing faster than humans. In present, source code is used in mostly static program analysis tool and applying many techniques for analyze program. Unfortunately, most of static program analysis tool operate by source codes that are complicated for undertaking and difficult to understand. Motivated by these problems, researchers investigate new method for program analysis.

This research attempted to use a Java Bytecode instead of source code. Most of Java programs could be transformed into Bytecode form but can't be transformed to source code directly. Sometimes when a program cannot be open source code that makes analysis by source code is difficult or impossible. So program analysis by Bytecode has more benefits.

School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000

Sarawuth Sonnum, E-mail: M5340316@g.sut.ac.th

Java Bytecode

Java Bytecode is the form of instructions that the Java virtual machine executes. It is created from the Java compile process. The most common form of output from a Java compiler is Java class files containing platform-neutral Java Bytecode. The amounts of Bytecode have effect to program size, used memory, and performance directly. Therefore, if a Java class file contained with larger amount of Bytecode, it must use most of memory for processing. Furthermore, the performance of program is reduced too. Figure 1 is the sample of java source code and next figure 2 is Bytecode compiled from figure 1.

```
public static int sumArray(int[] theArray){
        int sum = 0;
        for(int k=0;k<theArray.length;k++)
                sum += theArray[k];
        return sum;
}
```

Figure 1. The Source code of method "sumArray"

```
public static int sumArray(int[]);
 Code:
0:  iconst_0          //Load constant 0 onto stack
1:  istore_1          //Store 0 into var #1 (sum)
2:  iconst_0          //Store constant 0 on stack
3:  istore_2          //Store 0 into var #2 (k)
4:  iload_2           //Load var #2 (k) onto stack
5:  aload_0           //Load var #0 (theArray) onto stack
6:  arraylength       //Load length of "theArray" onto stack
7:  if_icmpge  22     //Branch to addr. 22 if k<length
10: iload_1           //Load var #1 (sum) onto stack
11: aload_0           //Load var #0 (theArray) onto stack
12: iload_2           //Load var #2 (k) onto stack
13: iaload            //Resolve "theArray[k]", load on stack
14: iadd              //Add "theArray[k]" to "sum"
15: istore_1          // Store 0 into var #1 (sum)
16: iinc 2, 1         //Increment var #2( k) by 1
19: goto 4            //Branch to addr. 4
22: iload_1           //Load var #1 (sum) onto stack
23: ireturn           //Return top item on stack (sum)
```

Figure 2. The Java Bytecode that compile from figure1.

Experimental and Methods

Transform a Java program to Java Bytecode.

```
public class Hello {
    public static void main(String args[]){
        int x = 11;
        int y = 12;
        System.out.println("Sum of x and y is : ");
        System.out.println(x+y);
    }
}
```

Figure 3. Source code of Hello.java

Figure 3 is an example of Java program name Hello. Then compile this program by "javac Hello.java" command that it transformed into Hello.class file. A file in form of .class is unreadable because it is Java machine language that called Bytecode. It must use "javap –c Hello" command to format this Bytecode and the result of formatted Bytecode is shown in figure 4. Both of "javac" and "javap" are built-in commands of JVM so they can be used in any computer that installed JDK.

```
Compiled from "Hello.java"                                    (1)
public class Hello extends java.lang.Object{                  (2)
public Hello();                                               (3)
  Code:
(4) 0:   aload_0
    1:   invokespecial   #8; //Method java/lang/Object."<init>":()V
    4:   return                                               (5)

public static void main(java.lang.String[]);                  (3)
  Code:
(4) 0:   bipush  11
    2:   istore_1
    3:   bipush  12
    5:   istore_2
    6:   getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
    9:   ldc     #22; //String Sum of x and y is :
    11:  invokevirtual   #24; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
    14:  getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
    17:  iload_1
    18:  iload_2
    19:  iadd
    20:  invokevirtual   #30; //Method java/io/PrintStream.println:(I)V
    23:  return                                               (5)

}
```

Figure 4. The component of Java Bytecode in Hello.class file.

The Java ByteCode component partitioning

Figure 4. is an example of Bytecode from Hello.class file. Its' format is easy to read and interpret by human. This topic demonstrates how each part of this Bytecode works and its benefits. In this figure, each part of Bytecode represented by following numbers.

1. The source of this class file can be specified by text "Complied from". In this figure, source file is Hello.java.
2. A Java class name can be specified by word "class". Furthermore, we can guess class name from source file name because one rule in java programming said "Source file name must have the same name as the class". In this figure class name is Hello.

3. A Method can be noticed from symbol "()" and ";". If any of line contains with both symbol it is the method name. In this figure , there are 2 methods, i.e. Hello (default constructor) and main.
4. A Bytecode instruction in each method.
5. Return command represents to instruction of return a value or end of method.

The Java Bytecode command partitioning

In Java compile process, compiling 1 line of source code can be one or more Instruction in Bytecode (call Instruction set). Previous topic cannot determine the relation between source code and Bytecode. How can we know how one line of source code reference to any Bytecode line? We found that command "javap –l Hello" can solve this doubt. The result of this command (shown in figure 5) specifies each source code line and its corresponding Bytecode line.

```
public static void main(java.lang.String[]);
  LineNumberTable:
   line 4: 0
   line 5: 3
   line 6: 6
   line 7: 14
   line 8: 23
```

Figure 5. The result of command "javap –l Hello"

From figure 5, we found that text "line" and symbol ":" can specify the relation between source code and Bytecode. For example, command "line 4: 0" mean the source code at line number 3 reference to Bytecode line number 0 to 2 (because next command start at line 3). Next, we will partition the Bytecode by using this method and the result is shown in figure 6.

```
Compiled from "Hello.java"
public class Hello extends java.lang.Object{
public Hello();
  Code:
   0:   aload_0
   1:   invokespecial   #8; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   bipush  11
   2:   istore_1
   3:   bipush  12
   5:   istore_2
   6:   getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   9:   ldc     #22; //String Sum of x and y is :
   11:  invokevirtual   #24; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   14:  getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   17:  iload_1
   18:  iload_2
   19:  iadd
   20:  invokevirtual   #30; //Method java/io/PrintStream.println:(I)V
   23:  return

}
```

Figure 6. The result of Java Bytecode partitioned.

We store the partitioned Bytecode into H2 database using one Instruction set per one record in the database. We can access Bytecode by SQL that can be applied in several aspects and the examples will be demonstrated in the next topic.

Results and Discussion

An Example of Implementation

The structure of a Java program can be accessed via SQL. Thus 'it can be used in several aspects. A Bytecode that stored into database is easier to handle than normal class file. We will show some examples of the approach implementation from the research as follows:

- The Java Bytecode specific access: Example from figure 6, if you want to know what is the memory address of value 11(value of x). You can use "SELECT * FROM Hello WHERE code LIKE '*bipush_11*istore_*'" SQL command to query data. Finally, you will get the result is line 0 and 2 command. With this command set, you can specify the address of value 11 which is address 1.
- The Java program debugging: Example from figure 6, if you want to know what the variable x (stored in address 1) be used by any Bytecode line. You can use "SELECT * FROM Hello WHERE code LIKE '*istore_1*' OR code LIKE '*iload_1*'" SQL command to query data.
- Making control flow graphs (CFG): A Java Bytecode interpretation is a sequential process. Thus 'it can demonstrate the cursory operation. With each record of Bytecode, it can represent to each node on CFG. Nevertheless, the CFG from source and Bytecode may be different that is a weak point of this method.
- Making Flow charts: By using the sequential process of Bytecode and understanding the meaning of the Bytecode commands. Thus, we can draw the flow charts.
- Finding a danger operation: For example, virus, Trojan, and danger command.
- Finding an Exception: The tool "FindBugs" is one from many sample of Java exception analysis tool that used Bytecode.

The future works

In previous phase, demonstrate the structure of a Java program can be accessed via SQL and sample of use in several aspects. In the future, we will take this approach to develop the Java Exception finding tool. That tool will use this approach collaborate with other material such as pattern matching theory and H2 database. Figure 7 is cursory processes of the Java Exception finding tool that develop in the future.

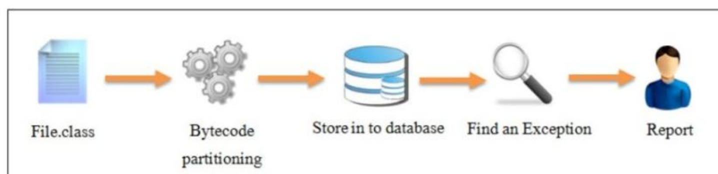


Figure 7. Operation of the Java Exception finding tool.

Conclusions

This paper has demonstrated that the structure of a Java program in the form of Bytecode can be accessible by the SQL. Thus ‘it can be used in several aspects, such as Java Bytecode debugging, making control flow graphs and creating flow charts. Furthermore, we have found that working with an in-memory SQL is convenient and faster than storing data in normal text files. In the future, we will use this approach to create a Java exception checking tool. Finally, we hope that this approach will provide a convenient method for people who are interested in Java Bytecode analysis.

References

Lance, D., R. H. Untch, et al. (1999). Bytecode-based Java program analysis. Proceedings of the 37th annual Southeast regional conference (CD-ROM), ACM: 14.

Hovemeyer, D. and W. Pugh (2004). "Finding bugs is easy." SIGPLAN Not. 39(12): 92-106.

Lievens, W. (2006, 3 November 2011). "Java bytecode." Java bytecode. Retrieved 28 December, 2010, from http://en.wikipedia.org/wiki/Java_bytecode.

Cabral, B. and P. Marques (2008). A Case for Automatic Exception Handling. Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society: 403-406.

Zhao, G., H. Chen, et al. (2008). Data-Flow Based Analysis of Java Bytecode Vulnerability. Proceedings of the 2008 The Ninth International Conference on Web-Age Information Management, IEEE Computer Society: 647-653.

Hamid, N. A. (2009). "Pattern matching on objects in Java." J. Comput. Small Coll. 25(1): 51-57.

มหาวิทยาลัยเทคโนโลยีสุรนารี

ประวัติผู้เขียน

นายศราวุธ สอนนำ เกิดเมื่อวันที่ 23 เดือนกุมภาพันธ์ พ.ศ. 2531 ณ จังหวัดชัยภูมิ สำเร็จการศึกษาระดับชั้นมัธยมศึกษาจากโรงเรียนแก้งคร้อวิทยา อำเภอแก้งคร้อ จังหวัดชัยภูมิ และสำเร็จการศึกษาระดับปริญญาตรีจากคณะวิศวกรรมศาสตร์ สาขาวิชาวิศวกรรมคอมพิวเตอร์ มหาวิทยาลัยเทคโนโลยีสุรนารี ในปีการศึกษา 2552 หลังจากสำเร็จการศึกษาได้เข้าทำงานใน บริษัทซอฟต์แวร์เฮาส์ ตำแหน่งโปรแกรมเมอร์เป็นเวลาเกือบ 1 ปี ในระหว่างนั้นได้มีโอกาสทำงานเกี่ยวกับการออกแบบซอฟต์แวร์ซึ่งมีความรู้สึกว่ามีความสนใจเป็นอย่างมาก จึงทำให้เกิดแรงจูงใจที่จะศึกษาต่อในระดับปริญญาโททางด้านวิศวกรรมซอฟต์แวร์ เพื่อเป็นการพัฒนาความรู้และความสามารถให้กับตนเองจึงได้เข้าศึกษาต่อในระดับปริญญาโท สาขาวิชาวิศวกรรมคอมพิวเตอร์ สำนักวิชาวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีสุรนารี ในปี พ.ศ.2553 ในขณะที่ศึกษาอยู่ได้มีโอกาสเป็นผู้ช่วยสอนในสาขาวิชาวิศวกรรมคอมพิวเตอร์มหาวิทยาลัยเทคโนโลยีสุรนารีจำนวน 2 รายวิชา คือ เทคโนโลยีเชิงวัตถุ (Object - Oriented Technology) และ การโปรแกรมโดยยึดเหตุการณ์(Event - Driven Programming) ซึ่งช่วยให้ผู้วิจัยได้นำประสบการณ์ และความรู้ที่ได้จาก การเป็นผู้ช่วยสอนมาประยุกต์ใช้กับงานวิจัยได้เป็นอย่างดี จากการทำวิจัยนี้ทำให้ผู้วิจัยมีความรู้และความเข้าใจทางด้านการออกแบบซอฟต์แวร์ด้วยภาษาจาวาและการทดสอบซอฟต์แวร์เป็นอย่างดี และมีผลงานตีพิมพ์เผยแพร่จำนวน 1 เรื่อง ดังที่แสดงในภาคผนวก ค.

มหาวิทยาลัยเทคโนโลยีสุรนารี