โปรแกรมสำหรับสังเคราะห์ Design Pattern จาก source code

A Design Pattern Recognition Program

นายปุณณวัฒน์ ธาดาภาคย์ รหัส 46360046

นายพรพงษ์ ร้องเกาะเกิด รหัส 46361994

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมไฟฟ้าและคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร

ปีการศึกษา 2549

ใบรับรองโครงงานวิศวกรรม

หัวข้อโครงงาน	โปรแกรมสำหรับสังเคราะห์ Design Pattern จาก source code
ผู้ดำเนินโครงงาน	นายปุณณวัฒน์ ธาดาภาคย์ รหัส 46360046
	นายพรพงษ์ ร้องเกาะเกิด รหัส 46361994
อาจารย์ที่ปรึกษา	ดร. สุรเดช จิตประไพกุลศาล
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
ภาควิชา	วิศวกรรมไฟฟ้าและคอมพิวเตอร์
ปีการศึกษา	2549

.....

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร อนุมัติให้โครงงานฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิศวกรรมศาสตรบัณฑิต สาขาวิศวกรรมคอมพิวเตอร์

คณะกรรมการสอบโครงงานวิศวกรรม

.....ประธานกรรมการ

(ดร.สุรเดช จิตประไพกุลศาล)

.....กรรมการ

(ดร.พนมขวัญ ริยะมงคล)

.....กรรมการ

(อาจารย์จิราพร พุกสุข)

หัวข้อโครงงาน	โปรแกรมสำหรับสังเคราะห์ Design Pattern จาก source code
ผู้ดำเนินโครงงาน	นายปุณณวัฒน์ ธาดาภาคย์ รหัส 46360046
	นายพรพงษ์ ร้องเกาะเกิด รหัส 46361994
อาจารย์ที่ปรึกษา	ดร. สุรเดช จิตประไพกุลศาล
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
ภาควิชา	วิศวกรรมไฟฟ้าและคอมพิวเตอร์
ปีการศึกษา	2549

.....

บทคัดย่อ

โครงงานนี้ได้พัฒนาโปรแกรม Design Pattern Recognition สำหรับตรวจหา Design Pattern จาก source code ภาษา Java โปรแกรมนี้ในขั้นต้นจะใช้โปรแกรม JavaCC ในการวิเคราะห์ source code ให้เป็นโครงสร้างในรูปแบบของ RSF (Rigi Standard Format) ซึ่งจะถูกนำไปให้โปรแกรม Crocopat ตรวจเปรียบเทียบกับ pattern ที่มีอยู่ในฐานข้อมูล จากการทดลองกับ source code ตัวอย่างในหนังสือทางด้าน Design Pattern 2 เล่ม คือ Design Pattern Java Companion และ Patterns in Java Volume 1 พบว่า โปรแกรมสามารถตรวจหา Design Pattern ได้ครบทุก Pattern ที่มีอยู่ในฐานข้อมูล

Project Title	A Design Pattern Recognition Program
Name	Mr. Punnawat Tadapak ID. 46360046
	Mr. Pornpong Rongkaokerd ID. 46361994
Project Advisor	Dr. Suradet Jitprapaikulsarn
Major	Computer Engineering
Department	Electrical and Computer Engineering
Academic Year	2006

.....

ABSTRACT

This project develops a Design Pattern Recognition program for detecting the Design Patterns in Java source codes. This program uses the JavaCC to help convert the source codes to structures in RSF format and uses Crocopat to compare the resulting RSF structures to the patterns in the database. Based on our experiments on the source codes from two book: Design Pattern Java Companion and Patterns in Java Volume 1, the program is able to detect all the patterns available in the Design Pattern database.

กิตติกรรมประกาศ

ขอขอบพระคุณ ดร.สุรเดช จิตประไพกุลศาล อาจารย์ที่ปรึกษาโครงงานนี้ ที่คอยให้คำปรึกษา ความช่วยเหลือตลอดจนคำแนะนำและแนวทางต่างๆ ในการทำโครงงานนี้ และสุดท้ายขอขอบพระคุณอาจารย์ทุกท่านและเพื่อนๆ ทุกคนที่ยังไม่ได้เอ่ยนามที่คอยให้การสนับสนุนผู้ดำเนินโครงงาน ให้สามารถทำโครงงานนี้จนสำเร็จลุล่วงไปได้ด้วยดี

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

ตารางที่ หน้า

สารบัญรูปภาพ

รูปที่ หน้า

สารบัญรูปภาพ (ต่อ)

สารบัญรูปภาพ (ต่อ)

สารบัญรูปภาพ (ต่อ)

บทที่ 1
บทนำ

1.1 ที่มาและความสำคัญของโครงงาน

การออกแบบซอฟต์แวร์เปรียบเสมือนกับการออกแบบบ้านของสถาปนิก ซึ่งจะนำรูปแบบแต่ละส่วนของบ้านที่มีโครงสร้างแตกต่างกัน แต่มีความสัมพันธ์กัน สามารถนำมารวมเป็นบ้านที่สมบูรณ์ได้ แต่บ้านที่ไม่ได้มีการออกแบบจะทำให้การสร้างบ้านเกิดความเสียหายได้ในภายหลัง ก็เหมือนกับการออกแบบซอฟต์แวร์ ถ้าขาดการออกแบบที่ดีแล้ว ผลที่ตามมาคือ ซอฟต์แวร์อาจเกิดข้อผิดพลาดได้

คำว่า "Design Pattern" ที่ใช้คำว่า Design เนื่องจากจะเน้นถึงเทคนิคการออกแบบโปรแกรมในการแก้ปัญหาแต่ละด้าน (Problem in context) ส่วน Pattern เป็นการนำเทคนิคการออกแบบที่มีอยู่หลายรูปแบบ นำมารวมกันให้สามารถนำกลับมาใช้ใหม่ได้ (Reusable)

ปัจจุบันการพัฒนาซอฟต์แวร์ในประเทศไทยกำลังมีการแข่งขันกันอย่างมาก เนื่องจากต้องการผลิตซอฟต์แวร์ออกมาเพื่อตอบสนองกับความต้องการของผู้ใช้ให้มากที่สุด ซึ่งผู้ผลิตซอฟต์แวร์ส่วนใหญ่จะคำนึงถึงแต่เพียงว่า จะพัฒนาซอฟต์แวร์อย่างไรให้สามารถตอบสนองกับความต้องการของผู้ใช้ได้อย่างรวดเร็วที่สุดโดยไม่คำนึงถึงเรื่องของคุณภาพในการพัฒนาซอฟต์แวร์ ซึ่งวิธีการควบคุมคุณภาพของซอฟต์แวร์นั้น ก็มีอยู่หลากหลาย แต่ถ้าผู้พัฒนาซอฟต์แวร์ได้คำนึงถึงคุณภาพของซอฟต์แวร์ตั้งแต่ขั้นตอนการออกแบบซอฟต์แวร์ ก็จะช่วยลดต้นทุนในการแก้ปัญหากับข้อผิดพลาดที่อาจจะเกิดขึ้นในภายหลังได้ ซึ่งวิธีการที่จะช่วยลดข้อผิดพลาดที่อาจจะเกิดขึ้น นั่นคือ หลักการของ Design Pattern

หลักการของ Design Pattern ได้ผ่านการใช้งานจริงจากผู้เชี่ยวชาญที่ได้ใช้กับงานจริงอย่างมากมาย ซึ่งซอฟต์แวร์ที่ใช้ Design Pattern จะมีความยืดหยุ่นสูง, ง่ายต่อการพัฒนา, เปลี่ยนแปลง, แก้ไขและทดสอบ ตลอดจนสามารถตรวจสอบได้ง่าย

นอกจากนี้การออกแบบซอฟต์แวร์ โดยใช้หลักการของ Design Pattern ทำให้ผู้ที่มาพัฒนาซอฟต์แวร์รุ่นต่อมา สามารถทำความเข้าใจกับ Source code ได้อย่างรวดเร็ว เนื่องจากการออกแบบโดยใช้ Design Pattern นั้น ไม่มีความซับซ้อน และจำนวนของคำสั่งที่ใช้ในแต่ละส่วนยังกระชับอีกด้วย ทำให้โปรแกรมที่พัฒนาขึ้นโดยใช้ Design Pattern นั้นมีประสิทธิภาพการทำงานที่ดีกว่าโปรแกรมที่พัฒนาโดยใช้วิธีการอื่น

1.2 วัตถุประสงค์ของโครงงาน

1. เพื่อให้เล็งเห็นถึงความสำคัญของการใช้ Design Pattern ช่วยในการออกแบบและพัฒนาซอฟต์แวร์ให้มีประสิทธิภาพมากขึ้น

2. สามารถตรวจหาได้ว่า Source code ที่นำมาตรวจหานั้น มี Design Pattern แบบใดบ้าง

1.3 ขอบข่ายของโครงงาน

1. พัฒนาโปรแกรมให้สามารถวิเคราะห์โครงสร้างจาก source code ภาษาจาวาให้เป็นโครงสร้างในรูปแบบของ RSF

2. พัฒนาโปรแกรมให้สามารถเปรียบเทียบ pattern ที่ได้จากโครงสร้างในข้อ 1 กับ Design Pattern ที่จัดเก็บในฐานข้อมูล

1.4 ขั้นตอนการดำเนินงาน

1. ศึกษาทฤษฎีและหลักการที่เกี่ยวข้องกับโครงงาน
   - รวบรวมข้อมูลและบทความที่เกี่ยวข้องกับโครงงาน
   - เลือก Tool เพื่อนำมาใช้พัฒนาโปรแกรม และศึกษาวิธีการใช้งาน
   - ศึกษารูปแบบการเขียนโปรแกรมภาษาจาวา
   - ศึกษารูปแบบและวิธีการเขียนรายงานที่ถูกต้อง
2. ออกแบบโปรแกรม
3. เขียนโปรแกรม
4. ทดสอบและปรับปรุงโปรแกรมให้สมบูรณ์และเหมาะสม
5. เขียนรายงานและจัดทำรูปเล่ม

1.5 แผนการดำเนินงาน

กิจกรรม	ปี 2548	ปี 2549									
	ธ.ค.	ม.ค.	ก.พ.	มี.ค.	เม.ย.	พ.ค.	มิ.ย.	ก.ค.	ส.ค.	ก.ย.	ต.ค.
1. ศึกษาทฤษฎีและหลักการที่เกี่ยวข้องกับโครงงาน	←	—	→								
2. ออกแบบโปรแกรม		←	—	→							
3. เขียนโปรแกรม				←	—	—	→				
4. ทดสอบและปรับปรุงโปรแกรมให้สมบูรณ์และเหมาะสม					←	—	—	—	→		
5. เขียนรายงาน และจัดทำรูปเล่ม									←	—	→

1.6 ผลที่คาดว่าจะได้รับ

1. เข้าใจวิธีการออกแบบซอฟต์แวร์โดยใช้ Design Pattern และได้เรียนรู้การพัฒนาซอฟต์แวร์อย่างมืออาชีพ

2. ได้เห็นถึงประสิทธิภาพของโปรแกรมที่ถูกสร้างและพัฒนาขึ้นโดยใช้ Design Pattern

3. สามารถนำ Design Pattern ไปประยุกต์ใช้กับการพัฒนาซอฟต์แวร์ในด้านต่างๆ ได้อย่างเหมาะสม

4. ได้ซอฟต์แวร์ที่สามารถแปลง Source code ที่เป็นภาษาจาวา ให้เป็นภาษา RSF ที่เป็น Pattern Language ได้ และสามารถตรวจหาได้ว่าพบ Design Pattern แบบใดบ้าง ใน source code

1.7 งบประมาณที่ใช้

1. ค่าวัสดุสำนักงาน	เป็นเงิน	400	บาท
2. ค่าวัสดุคอมพิวเตอร์	เป็นเงิน	100	บาท
3. ค่าหนังสือ	เป็นเงิน	800	บาท
4. ค่าถ่ายเอกสาร	เป็นเงิน	500	บาท
5. ค่าวัสดุอื่นๆ	เป็นเงิน	200	บาท
	รวมเป็นเงิน	2,000	บาท (สองพันบาทถ้วน)

บทที่ 2
หลักการและทฤษฎีที่เกี่ยวข้อง

ในบทนี้จะกล่าวถึงหลักการและทฤษฎีต่างๆ ที่นำมาประยุกต์ใช้เพื่อทำโครงงานนี้ ซึ่งทฤษฎีที่ได้ใช้ ได้แก่ ทฤษฎีและขั้นตอนการทำงานของตัวแปลภาษา หรือ Compiler เพื่อให้โปรแกรมสามารถตรวจสอบหาความสัมพันธ์ต่างๆ ได้ และหลักการทำงานของ Thread เพื่อควบคุมให้โปรแกรมมีการหยุดรอการทำงานที่ยังค้างอยู่ให้เสร็จก่อนที่จะทำงานอื่นต่อไป

2.1 ตัวแปลภาษา (Language Translator)

ตัวแปรภาษาแบ่งออกเป็น 3 ประเภท

Assemblers

Assembler คือ ตัวแปลภาษาสำหรับภาษา Assembly ซึ่งมีรูปแบบเป็นเครื่องหมายหรือสัญลักษณ์ (Symbolic Form) แทน Machine Language

Interpreters

Interpreter คือ โปรแกรมตัวแปลภาษา โดยจะอ่าน source code ทีละบรรทัด จากนั้นแปลโปรแกรมบรรทัดนั้นให้เป็น executable code และทำงานตามคำสั่งนั้น ตัวอย่างภาษาคอมพิวเตอร์ที่ใช้ตัวแปลภาษาแบบ Interpreter ได้แก่ ภาษา Basic, ภาษา Prolog เป็นต้น

Compilers

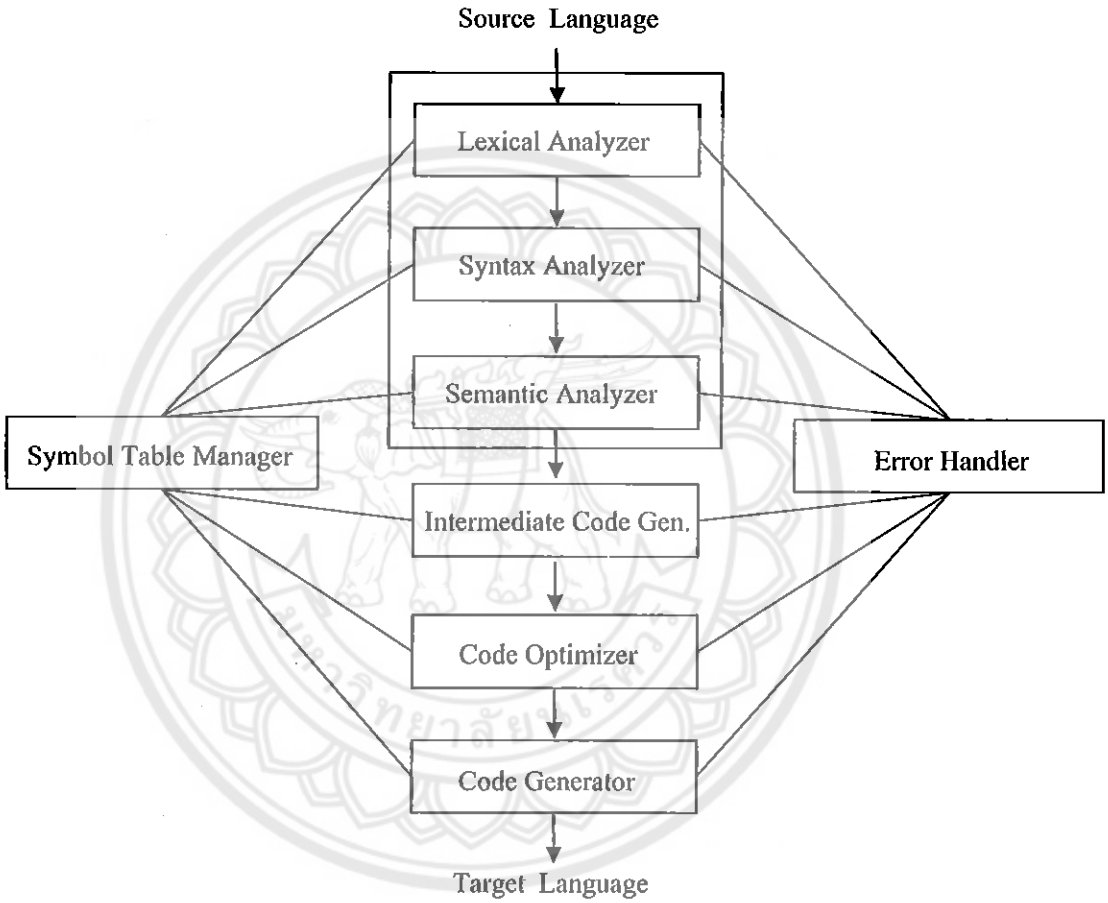
Compiler คือ โปรแกรมคอมพิวเตอร์ที่อ่านข้อมูล หรือโปรแกรมที่ถูกสร้างขึ้นจากภาษาหนึ่ง โดยจะเรียกว่า Source Language แล้วแปลงไปสู่โปรแกรมอีกภาษาหนึ่ง ที่มีการทำงานเหมือนกัน เรียกว่า Target Language โดยทั่วไป Source Language จะเป็นภาษาระดับสูง (High-Level Language) เช่น ภาษา C หรือ ภาษา COBOL ส่วน Target Language ก็คือ Machine Code

รูปที่ 2.1 ขั้นตอนการทำงานของ Compiler

โครงงานนี้ใช้หลักการของ Compiler โดยกำหนดให้ Source Language เป็นภาษา JAVA และ Target Language คือภาษา RSF

2.2 ขั้นตอนการทำงานของ Compiler

ก่อนที่จะได้เป็น Target Language ได้นั้น ตัว Compiler ต้องผ่านกระบวนการและขั้นตอนต่างๆ มากมาย โดยแต่ละขั้นตอนจะแปลง Source Language จากเครื่องหมายหรือสัญลักษณ์หนึ่งไปเป็นสัญลักษณ์อีกรูปแบบหนึ่ง ดังรูปที่ 2.2 แต่ในทางปฏิบัติขั้นตอนบางขั้นตอนนั้นสามารถที่จะรวมกันได้



รูปที่ 2.2 ขั้นตอนการทำงานของ Compiler [2]

จากรูปที่ 2.2 ขั้นตอน Lexical Analysis, Syntax Analysis และ Semantic Analysis เป็นกระบวนการที่สำคัญสำหรับการวิเคราะห์ใน Compiler (Analysis Phases) โดยมีส่วนสนับสนุนการทำงานอีก 2 ส่วน คือ Symbol Table Manager และ Error Handler

2.2.1 Lexical Analysis

เป็นขั้นตอนการอ่านอักขระจาก source language แล้วทำการจัดเป็น token หรือกลุ่มของตัวอักขระที่เป็นชนิดเดียวกันและอยู่ติดกันไว้ในกลุ่มอักขระเดียวกัน

2.2.2 Syntax Analysis

เป็นขั้นตอนการวิเคราะห์กลุ่มคำกับไวยกรณ์ (parsing) เพื่อกำหนดชนิดของ token แต่ละกัน

2.2.3 Semantic Analysis

เป็นขั้นตอนการตรวจสอบองค์ประกอบของโปรแกรมและข้อผิดพลาดเกี่ยวกับความหมายหรือชนิดของ token

2.2.4 Intermediate Code Generator Phase

เป็นขั้นตอนการสร้าง Intermediate form หรือ รูปแบบที่เป็นกลาง เพื่อให้ง่ายต่อการแปลงไปเป็น target language

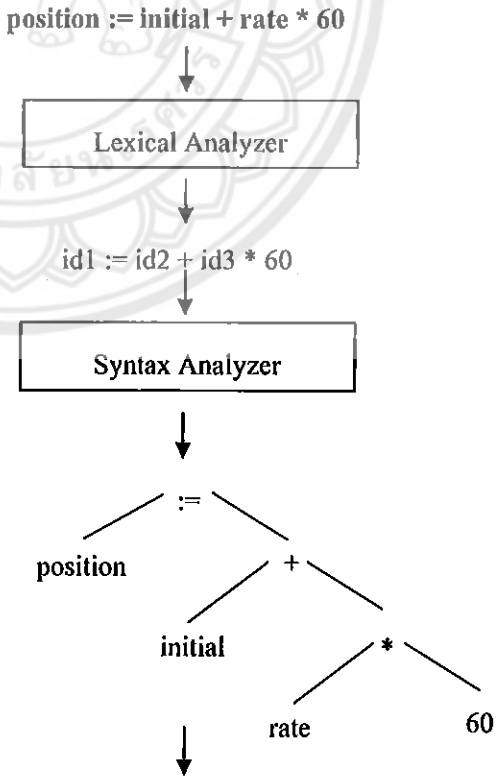
2.2.5 Code Optimizer Phase

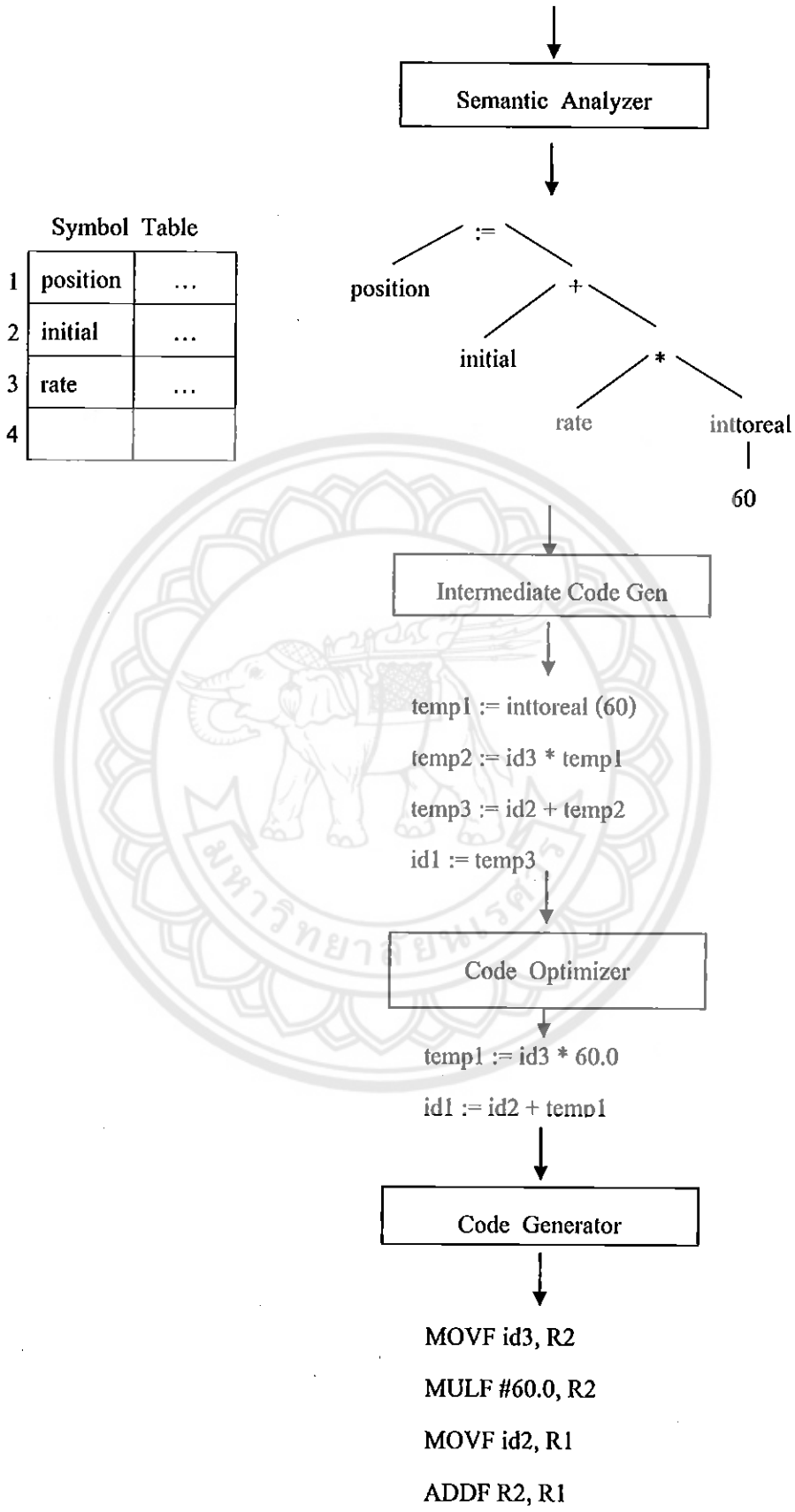
เป็นการปรับปรุง code เพื่อให้สามารถใช้ได้กับ machine code ได้อย่างเหมาะสมและรวดเร็วที่สุด

2.2.6 Code Generation Phase

เป็นขั้นตอนสุดท้ายของการคอมไพเลอร์ ซึ่งขั้นตอนนี้จะสร้าง target language, จัดการกับหน่วยความจำ, การเลือกชุดคำสั่ง และการกำหนด register ให้กับตัวแปรแต่ละตัว

ตัวอย่างการทำงานของคอมไพเลอร์ในแต่ละขั้นตอน แสดงดังรูปที่ 2.3

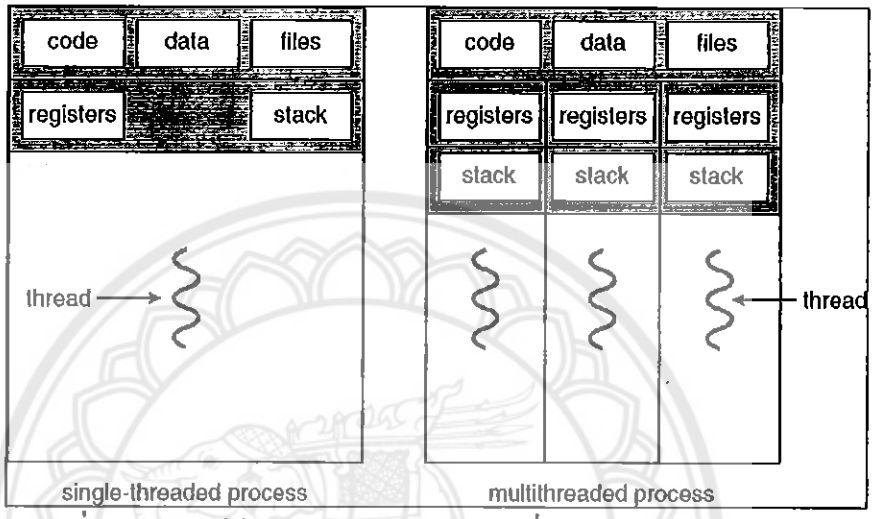




รูปที่ 2.3 ตัวอย่างการทำงานของคอมไพเลอร์ [2]

2.3 Thread

Thread เป็นส่วนประกอบของโปรเซส ถ้าในหนึ่งโปรเซสมี thread ควบคุมเพียง thread เดียว จะเรียกว่า Single Thread แต่ถ้าในหนึ่งโปรเซสประกอบด้วย thread หลายๆ ตัว จะเรียกว่า Multithreaded ซึ่ง thread ที่อยู่ในโปรเซสเดียวกันจะสามารถใช้โค้ด, ข้อมูล และทรัพยากรของระบบร่วมกันได้



รูปที่ 2.4 การใช้ข้อมูลร่วมกันของ thread ที่อยู่ในโปรเซสเดียวกัน [3]

ภายใน Thread แต่ละตัวจะประกอบไปด้วย

1. Thread ID เป็นหมายเลขของ Thread ในโปรเซส
2. Program Counter ใช้นับจำนวนคำสั่ง เพื่อระบุตำแหน่งของคำสั่งที่กำลังจะประมวลผลในลำดับต่อไป
3. Register Set ใช้เก็บค่าที่ทำงานอยู่
4. Stack ใช้เก็บประวัติการประมวลผล

2.4 Thread ในภาษาจาวา

การสร้าง thread ในภาษาจาวามี 2 ลักษณะ คือ

2.4.1 การสร้างโดยใช้ Thread Class จะมี method สำหรับควบคุมการทำงานของ thread คือ start(), yield(), sleep(), suspend(), resume(), stop(), run() เป็นต้น เพื่อกำหนดสถานะและควบคุมการทำงานของ thread

2.4.2 การสร้างโดยใช้ Runnable Interface จะมีเพียง method สำหรับควบคุมการทำงานของ thread เพียง method เดียวเท่านั้น นั่นคือ run() และ class ที่ได้ implement Runnable

Interface ไปใช้งานนั้น จำเป็นต้องมี method run() เสมอ เพื่อระบุขั้นตอนการทำงานให้กับ thread เมื่ออยู่ในสถานะ running

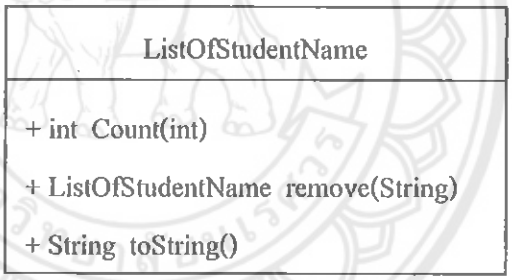
เนื่องจากภาษาจาวาไม่สามารถ extends class ได้มากกว่า 1 class จึงจำเป็นต้องมีการสร้าง thread ในลักษณะที่ 2 ขึ้น เพื่อให้ class ที่ไม่สามารถ extends class อื่นได้ สามารถเรียกใช้งาน thread ได้

2.5 ความสัมพันธ์ของระหว่าง class ในภาษาจาวา

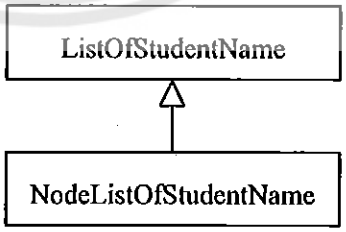
ความสัมพันธ์ของระหว่าง class (Class relationship) มีด้วยกัน 3 ลักษณะ คือ

2.5.1 ความสัมพันธ์แบบ Inherit

เป็นความสัมพันธ์ของการถ่ายทอดคุณสมบัติระหว่าง class ที่มีความสัมพันธ์กัน โดยคุณสมบัติที่ class ลูกจะได้รับ คือ การเรียกใช้งาน method ต่างๆ หรือการกำหนดขั้นตอนการทำงานให้กับ method ตัวอย่างเช่น ไฟล์ NodeListOfStudentName.java ได้ inherit จากไฟล์ ListOfStudentName.java ซึ่งไฟล์นี้มี method 3 method ได้แก่ method count, remove และ toString (ดังรูปที่ 2.5 และรูปที่ 2.6)



รูปที่ 2.5 class diagram ของไฟล์ ListOfStudentName.java



รูปที่ 2.6 UML ของความสัมพันธ์แบบ Inherit

ดังนั้นไฟล์ NodeListOfStudentName.java สามารถกำหนดขั้นตอนการทำงานให้กับ method count และ method remove ได้ตามต้องการ เพราะ method ทั้งสองอันได้ประกาศไว้เป็นแบบ abstract และนอกจากนี้ไฟล์ NodeList-OfStudentName.java สามารถเรียกให้ method toString ทำงานได้โดยไม่ต้องกำหนดขั้นตอนการทำงาน เนื่องจากไฟล์ ListOfStudentName.java

ได้กำหนดขั้นตอนการทำงานของ method นี้แล้ว ซึ่งไฟล์ NodeListOfStudentName.java และไฟล์ ListOfStudentName.java สามารถดู source code ได้จากภาคผนวก ก.

2.5.2 ความสัมพันธ์แบบ Contain

เป็นความสัมพันธ์ที่ attribute สามารถใช้ชนิดของข้อมูลที่เป็น Object อื่นได้อย่างหลากหลาย ไม่จำกัดเพียงแค่ชนิดข้อมูลขั้นพื้นฐาน ตัวอย่างเช่น ไฟล์ Employee.java ในภาคผนวก ก. มี attribute ดังนี้

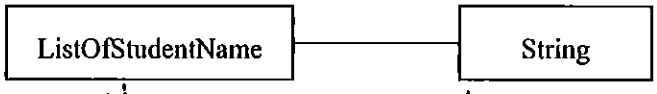
- String name;
- float salary;
- Vector subordinates;
- boolean isLeaf;
- Employee parent = null;

ดังนั้นไฟล์ Employee.java มีความสัมพันธ์แบบ contain กับ Object String, Vector และ Employee (ดังรูปที่ 2.7) เพราะชนิดข้อมูลของ attribute name, subordinates และ parent ไม่ได้เป็นชนิดข้อมูลแบบพื้นฐาน ซึ่งชนิดข้อมูลแบบพื้นฐานของภาษาจาวา ได้แก่ int, float, double, boolean, char, byte, short และ long

รูปที่ 2.7 UML ของความสัมพันธ์แบบ Contain

2.5.3 ความสัมพันธ์แบบ Use

เป็นความสัมพันธ์อีกลักษณะหนึ่ง ซึ่งมีลักษณะคล้ายกับความสัมพันธ์แบบ Contain แต่แตกต่างกันตรงที่ความสัมพันธ์แบบ Use เป็นการกำหนดชนิดข้อมูลของตัวแปรเท่านั้น ตัวอย่างเช่น ไฟล์ ListOfStudentName.java ในภาคผนวก ก. มีการประกาศใช้ตัวแปร 2 ตัว คือ nameToCount และตัวแปร nameToRemove ซึ่งตัวแปรทั้งสองตัวนี้มีชนิดตัวแปรแบบ String ดังนั้น ไฟล์ ListOfStudentName.java มีความสัมพันธ์แบบ Use กับ String ดังรูปที่ 2.8



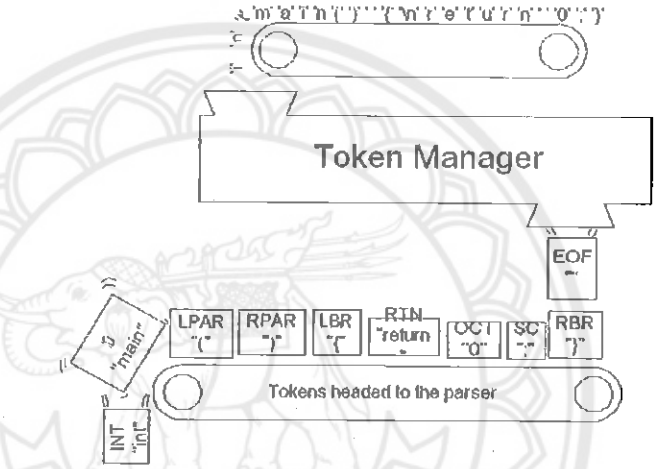
รูปที่ 2.8 UML ของความสัมพันธ์แบบ Use

2.6 โปรแกรม JavaCC

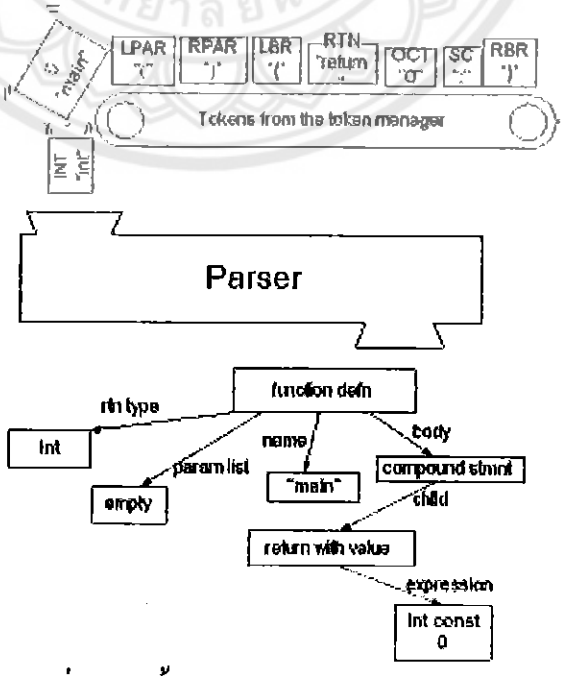
JavaCC หรือ Java Compiler Compiler เป็นทั้ง parser generator และ lexical analyzer generator กล่าวคือ JavaCC เป็นโปรแกรมที่ช่วยสร้างคอมไพเลอร์ โดยจะอ่านคำอธิบายหลักไวยากรณ์ของภาษา จากนั้นจะสร้างคอมไพเลอร์ ซึ่งภายในเขียนขึ้นด้วยภาษาจาวา

2.6.1 ขั้นตอนการทำงานของคอมไพเลอร์ที่สร้างด้วย JavaCC

คอมไพเลอร์ที่ถูกสร้างด้วยโปรแกรม JavaCC จะมี Token Manager สำหรับอ่านตัวอักขระ และจัดกลุ่มของตัวอักขระที่เป็นชนิดเดียวกันและอยู่ติดกันไว้ด้วยกัน (Token) ซึ่งหลักการแบ่งชนิดของกลุ่มคำจะขึ้นอยู่กับ Regular Expression ที่กำหนดไว้



รูปที่ 2.9 ขั้นตอนการทำงานของ Token Manager [6]



รูปที่ 2.10 ขั้นตอนการทำงานของ Parser [6]

จากรูปที่ 2.9 หลังจากตัวอักษรทั้งหมดได้ผ่านขั้นตอน Token Manager แล้ว จะได้เป็นกลุ่มคำที่เรียกว่า Token หลังจากนั้นจะนำ Token ไปขั้นตอน parser เพื่อกำหนดชนิดของ Token แต่ละอัน ดังรูปที่ 2.10

2.6.2 Regular Expression

ขั้นตอนการทำงานของ Token Manger จำเป็นต้องใช้หลักการเรื่อง Regular Expression เพื่อช่วยในการจัดกลุ่มของ Token ต่างๆ โดย Regular Expression ที่นำมาใช้ในโปรแกรม JavcCC นั้น มีการกำหนดสัญลักษณ์เพื่อใช้แทนความหมายต่างๆ ดังนี้

1. (...)+ หมายถึง จะต้องมี ... อย่างน้อย 1 ตัว
2. (...)* หมายถึง จะมี ... หรือไม่มีก็ได้ ถ้ามีจะมี ... ได้ตั้งแต่ 1 ตัวขึ้นไป
3. [...] หมายถึง การแทนช่วงของตัวอักษร เช่น ["A" – "Z"] จะหมายถึงตัว "A" ถึงตัว "Z" ตัวพิมพ์ใหญ่ทั้งหมด
4. ~ หมายถึง นิเสธ เช่น ~[] แทนทุกตัวอักขระ

2.7 โปรแกรม Crocopat

Crocopat เป็นโปรแกรมสำหรับวิเคราะห์ และจัดการกับความสัมพันธ์ในรูปแบบต่างๆ รวมทั้งรูปแบบกราฟที่มีทิศทางด้วย (Directed Graph) และโปรแกรม Crocopat มีประสิทธิภาพในด้านการใช้เวลาประมวลผลและการใช้หน่วยความจำให้มีประสิทธิภาพ เนื่องจากโปรแกรมนี้จัดเก็บความสัมพันธ์ต่างๆ โดยใช้โครงสร้างของข้อมูลแบบ Binary Decision Diagram (BDD) นอกจากนี้โปรแกรม Crocopat สามารถใช้งานได้ง่ายและสามารถนำไปประยุกต์ใช้กับโปรแกรมอื่นที่ต้องการวิเคราะห์หาความสัมพันธ์ในลักษณะที่ต้องการได้

การใช้งานโปรแกรม Crocopat เพื่อวิเคราะห์หาความสัมพันธ์นั้น จะต้องประกอบไปด้วย

- โปรแกรมที่เขียนด้วยภาษา RML (Relation Manipulation Language) ซึ่งเป็นโปรแกรมที่ใช้หาความสัมพันธ์ตามที่ต้องการ ซึ่งจะเป็นไฟล์ข้อมูลนำเข้าที่จำเป็นต้องมีเมื่อใช้งานโปรแกรม Crocopat โดยตัวอย่างโปรแกรมภาษา RML ดังรูปที่ 2.11
- ไฟล์ข้อมูลที่ใช้รูปแบบการเก็บข้อมูลแบบ RSF (Rigi Standard Format) ซึ่งไฟล์ RSF สามารถเป็นได้ทั้งไฟล์ข้อมูลนำเข้า ในกรณีที่ในไฟล์มีความสัมพันธ์ของข้อมูล และไฟล์ข้อมูลนำออก ในกรณีที่โปรแกรม RML สั่งให้แสดงผลโดยบันทึกลงไฟล์

รูปแบบการเก็บข้อมูลของไฟล์ RSF ประกอบด้วยหลายๆ triples มารวมกัน ซึ่งแต่ละบรรทัดจะมีเพียง triple เดียวเท่านั้น โดยแต่ละ triple จะประกอบด้วย identifier จำนวน 3 ตัว และแต่ละ identifier จะถูกแบ่งด้วยช่องว่าง ส่วนการเขียน comment จะต้องเริ่มต้นบรรทัดด้วย # เท่านั้น โดยรูปแบบทั่วไปของภาษา RSF และตัวอย่างไฟล์ข้อมูล RSF ดังรูปที่ 2.12 และรูปที่ 2.13 ตามลำดับ

```
CompPat(component, composite, leaf) :=
        Inherit(composite, component)
        & Inherit(leaf, component)
        & Contain(composite, component)
        & ! Contain(leaf, component);
PRINT CompPat(component, composite, leaf);
```

รูปที่ 2.11 ตัวอย่างโปรแกรมภาษา RML

```
Verb        Subject        Subject
```

รูปที่ 2.12 รูปแบบทั่วไปของการเก็บข้อมูลในรูปแบบ RSF

```
Inherit   A   C
Inherit   B   C
Contain   A   C
```

รูปที่ 2.13 ตัวอย่างไฟล์ข้อมูล RSF

บทที่ 3
วิธีการดำเนินงาน

ในบทนี้จะกล่าวถึงขั้นตอนการทำงาน เพื่อให้ได้มาซึ่งโปรแกรมที่สามารถค้นหา Design Pattern จาก source code ภาษาจาวาได้ ซึ่งมีขั้นตอนการทำงานหลักๆ ได้แก่ ขั้นตอนการศึกษาและรวบรวมข้อมูล, ขั้นตอนการทดลองใช้งานโปรแกรม Crocopat และ JavaCC, ขั้นตอนการออกแบบและเขียนโปรแกรม Design Pattern Recognition

3.1 การประยุกต์ใช้ JavaCC

การสร้างคอมไพเลอร์โดยใช้โปรแกรม JavaCC จำเป็นจะต้องสร้างไฟล์ที่มีนามสกุล .jj ซึ่งภายในไฟล์จะเป็นคำอธิบายหลักไวยากรณ์ของภาษา รวมถึงสิ่งที่ต้องการให้โปรแกรมนั้นทำงานควบคู่ไปกับการทำงานของคอมไพเลอร์

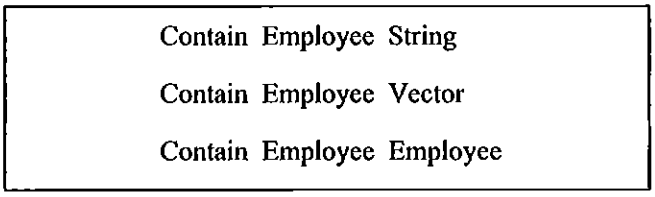
คำอธิบายหลักไวยากรณ์ของภาษาจาวาที่นำมาใช้ในโครงงานนี้ เขียนขึ้นโดย Sreenivasa Viswanadha จากนั้นทางผู้ดำเนินโครงงานก็ได้นำมาปรับปรุงให้สามารถหาความสัมพันธ์ระหว่าง class ซึ่งได้แก่ ความสัมพันธ์แบบ Inherit, ความสัมพันธ์แบบ Contain และความสัมพันธ์แบบ Use โดยมีขั้นตอนดังนี้

1. การหาความสัมพันธ์ของ class แบบ Inherit สามารถหาได้จาก class นั้นมีการ extends หรือ implements Object อื่น ตอนประกาศชื่อ class ตัวอย่างเช่น ไฟล์ NodeListOf-StudentName.java และไฟล์ ListOfStudentName.java ในภาคผนวก ก. จะได้รูปแบบของภาษา RSF ดังรูปที่ 3.1 และแผนภาพ UML ดังรูปที่ 2.6

```
Inherit NodeListOfStringName ListOfStudentName
```

รูปที่ 3.1 ความสัมพันธ์แบบ Inherit ในรูปแบบภาษา RSF

2. การหาความสัมพันธ์ของ class แบบ Contain สามารถหาได้จาก attribute ที่อยู่ใน class นั้น ตัวอย่างเช่นไฟล์ Employee.java ในภาคผนวก ก. จะได้รูปแบบของภาษา RSF ดังรูปที่ 3.2 และแผนภาพ UML ดังรูปที่ 2.7



รูปที่ 3.2 ความสัมพันธ์แบบ Contain ในรูปแบบภาษา RSF

3. การหาความสัมพันธ์ของ class แบบ Use สามารถหาได้จาก class นั้นมีการประกาศใช้ตัวแปรที่มีลักษณะเป็น Object อื่นนอกเหนือจาก Object ที่มีความสัมพันธ์แบบ inherit หรือ contain ตัวอย่างเช่น ไฟล์ ListOfStudentName.java ในภาคผนวก ก. จะได้รูปแบบของภาษา RSF ดังรูปที่ 3.3 และแผนภาพ UML ดังรูปที่ 2.8

รูปที่ 3.3 ความสัมพันธ์แบบ Use ในรูปแบบภาษา RSF

เมื่อได้ตำแหน่งของความสัมพันธ์ทั้ง 3 รูปแบบแล้ว ถ้าต้องการให้คอมไพเลอร์ส่งค่า token ณ ตำแหน่งที่เป็นความสัมพันธ์ดังกล่าว เพื่อใช้ประมวลผลต่อไป จะต้องใช้ method getNextToken () ถ้าต้องการ token ในตำแหน่งถัดไปจากตำแหน่งที่คอมไพเลอร์กำลังพิจารณาอยู่ หรือ method getToken (int index) โดยที่ index คือตำแหน่งของ token ที่ต้องการ ซึ่งถ้า index มีค่าเป็นศูนย์ นั่นคือ token ณ ตำแหน่งที่คอมไพเลอร์กำลังพิจารณาอยู่

3.2 การออกแบบโปรแกรม

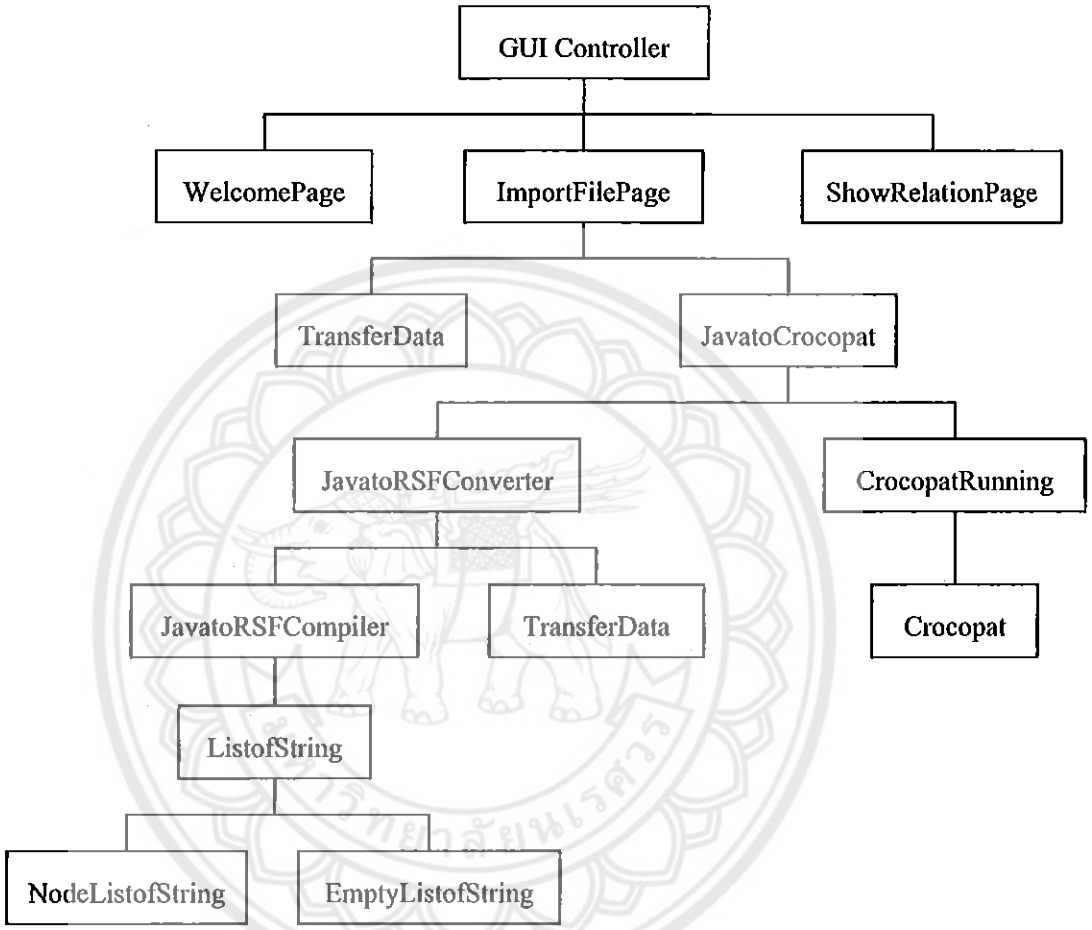
3.2.1 เลือกเทคนิคที่ใช้ในการออกแบบโปรแกรม

ผู้ดำเนินโครงงานได้นำเทคนิคการการออกแบบซอฟต์แวร์แบบ Modular Decomposition มาช่วยแยกหน้าที่และระบบต่างๆ ของโปรแกรมออกเป็นส่วนๆ เพื่ออำนวยความสะดวกในการเขียนและทดสอบโปรแกรม ซึ่งหลังจากการวิเคราะห์ความต้องการและขอบเขตของโครงงานแล้ว พบว่าโปรแกรมสามารถแยกหน้าที่การทำงานออกเป็นส่วนๆ ได้ดังนี้

- ส่วนควบคุมการทำงานของ GUI
- ส่วนอ่านข้อมูลจากไฟล์ข้อมูล
- ส่วนเขียนข้อมูลจากไฟล์ข้อมูล
- ส่วนควบคุมการทำงานโปรแกรม Crocopat
- ส่วนควบคุมการทำงานโปรแกรม JavaCC

3.2.2 วิเคราะห์โครงสร้างเพื่อเตรียมเขียนโปรแกรม

หลังจากการออกแบบซอฟต์แวร์โดยใช้เทคนิค Modular Decomposition แล้วจะได้ส่วนประกอบต่างๆ ของโปรแกรม จากนั้นนำส่วนต่างๆ มาวิเคราะห์และหาความสัมพันธ์ระหว่างส่วนต่างๆ จนได้เป็นโครงสร้างของโปรแกรมที่แสดงไว้ในรูปที่ 3.4

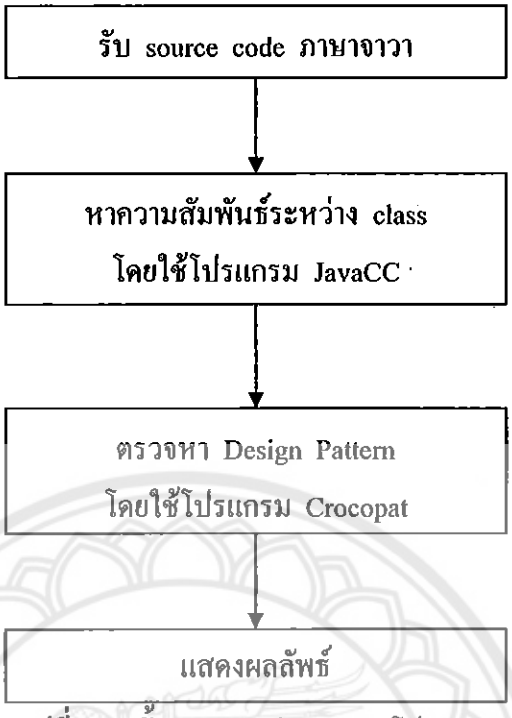


รูปที่ 3.4 โครงสร้างของโปรแกรม

จากรูปที่ 3.4 พบว่ามีส่วนประกอบบางส่วนมีการใช้งานหลายครั้ง ทำให้ทราบว่ามีส่วนใดบ้างที่มีการทำงานคล้ายกันและสามารถนำกลับมาใช้ใหม่ได้ โดยจะได้ไม่ต้องเสียเวลาในการเขียนโปรแกรมซ้ำเดิมอีกครั้ง

3.2.3 วิเคราะห์ลำดับการทำงาน

เมื่อได้โครงสร้างของโปรแกรมแล้ว จะต้องมาวิเคราะห์และจัดลำดับขั้นตอนการทำงานโปรแกรมในส่วนต่างๆ เพื่อลดการทำงานที่ซับซ้อน และเพิ่มความรวดเร็วในการทำงานของโปรแกรม ซึ่งจะได้ลำดับการทำงานของโปรแกรมดังรูปที่ 3.5



รูปที่ 3.5 ขั้นตอนการทำงานของโปรแกรม

3.3 การเขียนโปรแกรม

การเขียนโปรแกรมแบ่งออกได้เป็น 3 ส่วนหลักๆ คือ ส่วนควบคุมการทำงานของโปรแกรม Crocopat, ส่วนค้นหาความสัมพันธ์ของ source code โดยใช้โปรแกรม JavaCC ช่วยในการสร้าง Compiler เพื่อวิเคราะห์ไวยากรณ์ของภาษาจาวา และส่วนควบคุมการทำงานของ GUI (Graphic User Interface)

การเขียนโปรแกรมเพื่อตรวจหา Design Pattern ทั้ง 3 ส่วนเขียนโดยใช้ภาษาจาวา เนื่องจากภาษาจาวาเป็นภาษาหนึ่งที่ใช้ในการเขียนโปรแกรมทางคอมพิวเตอร์ ซึ่งถูกออกแบบมาให้เหมาะสมกับการเขียนโปรแกรมเชิงวัตถุ (Object Oriented Programming) โดยภาษาจาวานั้นจะมีลักษณะเด่น ดังต่อไปนี้

- เป็นภาษาที่สามารถทำความเข้าใจได้ง่าย สำหรับผู้ที่เคยศึกษาการเขียนโปรแกรมโดยใช้ภาษา C หรือ C++ มาแล้ว เนื่องจากหลักไวยากรณ์ของภาษาจาวาส่วนใหญ่ มีลักษณะคล้ายกับภาษา C และ C++ และนอกจากนี้ภาษาจาวายังได้ตัดกลไกที่ซับซ้อนของภาษา C และ C++ ออกไป เช่น การใช้งาน pointer, การสร้างโครงสร้างข้อมูล, การจัดการหน่วยความจำ และการใช้ header file เป็นต้น
- ภาษาจาวามี exception handling เพื่อใช้สำหรับจัดการกับข้อผิดพลาดของโปรแกรมที่ไม่ได้เกิดขึ้นจากการเขียนโปรแกรม เพื่อให้โปรแกรมสามารถทำงานต่อไปได้

- โปรแกรมที่เขียนด้วยภาษาจาวานั้น สามารถทำงานบนเครื่องคอมพิวเตอร์ที่มีระบบปฏิบัติการที่แตกต่างกันได้ (platform independent) โดยอาศัย java virtual machine
- ภาษาจาวามี garbage collection เพื่อช่วยจัดการกับหน่วยความจำที่ไม่ได้ใช้งานโดยอัตโนมัติ ทำให้ลดปัญหาการเกิด memory leak
- ภาษาจาวามีระบบรักษาความปลอดภัยที่เรียกว่า Sandbox Model เพื่อรักษาความปลอดภัยให้กับระบบที่มีการรันโปรแกรมที่ถูก download มาจากแหล่งที่ไม่น่าเชื่อถือผ่านทางระบบเครือข่าย โดยจะถูกนำไปไว้ใน sandbox เพื่อจำกัดสิทธิ์ในการใช้ทรัพยากรของระบบ

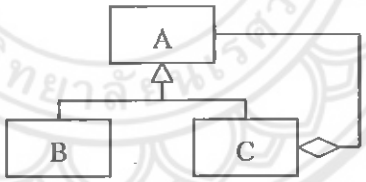
บทที่ 4

ผลการทดลอง

ในบทนี้จะกล่าวถึงการทดสอบโปรแกรมและแสดงผลของการทดสอบ โดยขั้นตอนการทดสอบโปรแกรมจะแบ่งออกเป็น 2 ส่วน เพื่อความสะดวกในการตรวจสอบหาข้อผิดพลาด โดยการทดสอบส่วนที่หนึ่งเป็น การทดสอบโปรแกรม Crocopat กับ Composite Design Pattern รูปแบบต่างๆ และการทดสอบส่วนที่สองเป็นการทดสอบโปรแกรม Design Pattern Recognition กับ source code ของผู้ที่นำหลักการของ Design Pattern มาใช้

4.1 ทดสอบโปรแกรม Crocopat โดยใช้ Composite Pattern

Composite Pattern สามารถเขียนอธิบายความสัมพันธ์ในรูปแบบของ UML ได้ดังรูปที่ 4.1 ซึ่งสามารถนำความสัมพันธ์ของแต่ละ node มาเขียนเป็นภาษา RML โดยที่แต่ละเส้นเชื่อมระหว่าง node จะแทนหนึ่งความสัมพันธ์ ได้ดังรูปที่ 4.2 และนอกจากนี้ยังสามารถเขียนเป็นภาษา RSF ได้ดังรูปที่ 4.3 โดยโปรแกรมที่มีลักษณะโครงสร้างแบบรูปที่ 4.1 ได้แก่ โปรแกรมสร้าง Link list ซึ่งประกอบด้วยไฟล์ IntList.java, NodeIntList.java และ Empty-List.java โดยสามารถดู source code ได้จากภาคผนวก ก.



รูปที่ 4.1 UML ของ Composite Pattern รูปแบบทั่วไป

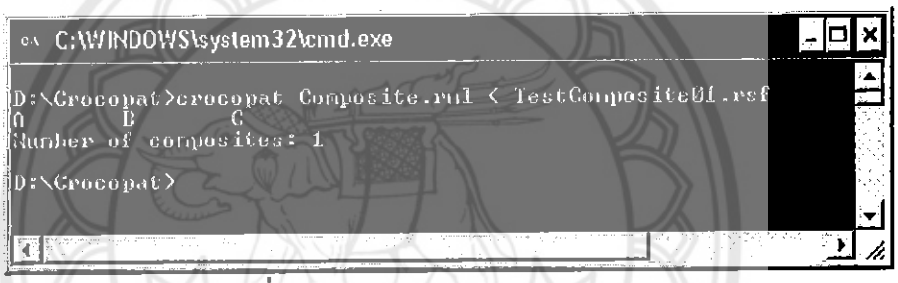
```
CompPat(A, B, C) :=
                Inherit(B, A)
                & Inherit(C, A)
                & Contain(C, A)
                & ! Contain(B, A);
PRINT CompPat(A, B, C);
PRINT "Number of composites: ", #(CompPat(A,_,_)), ENDL TO STDERR;
```

รูปที่ 4.2 ภาษา RML จากรูปที่ 4.1

```
Inherit   B   A
Inherit   C   A
Contain   C   A
```

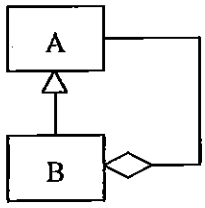
รูปที่ 4.3 ภาษา RSF จากรูปที่ 4.1

เมื่อได้ไฟล์ข้อมูลครบทั้งไฟล์ข้อมูลนำเข้า (RML) และไฟล์ข้อมูลที่ต้องการนำมาตรวจหาความสัมพันธ์ (RSF) แล้ว จากนั้นจะให้โปรแกรม Crocopat ทำงานเพื่อตรวจหาความสัมพันธ์และโครงสร้างแบบ Composite Pattern ซึ่งผลจากการรันโปรแกรม Crocopat จะได้เป็นรูปที่ 4.4



รูปที่ 4.4 ผลการรันโปรแกรม Crocopat

Composite Pattern นั้นสามารถเขียนแสดงเป็น UML ได้หลายรูปแบบ โดยแต่ละรูปแบบจะมีความแตกต่างกันขึ้นอยู่กับการนำ Composite Pattern ไปประยุกต์ใช้ในการแก้ไขปัญหาที่แตกต่างกัน ซึ่งจากการทดลองพบว่า รูปแบบของ Composite Pattern มีความเป็นไปได้อย่างน้อย 6 แบบ ดังรูปที่ 4.5, 4.7, 4.9, 4.11 และ 4.13

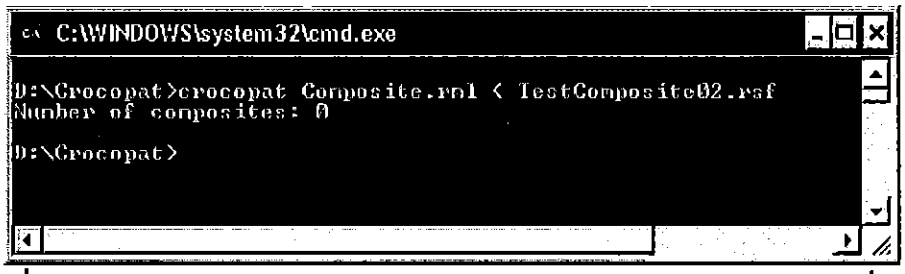


a

```
Inherit   B   A
Contain   B   A
```

b

รูปที่ 4.5 a. UML ของ Composite Pattern, b. ภาษา RSF ที่แสดงความสัมพันธ์ของรูป a

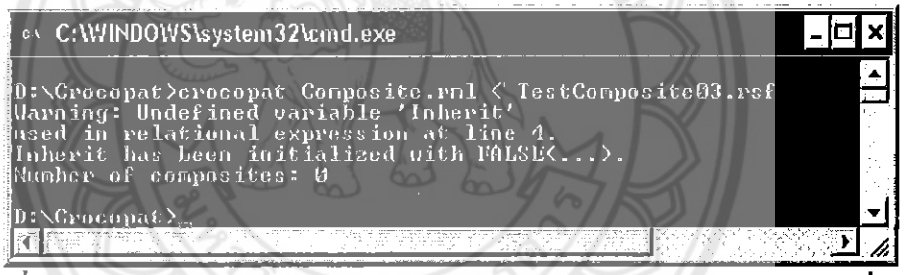


รูปที่ 4.6 ผลลัพธ์จากการรันโปรแกรม Crocopat โดยใช้ความสัมพันธ์ของรูปที่ 4.5

จากรูปที่ 4.5 เป็นโครงสร้างของ Composite Pattern รูปแบบหนึ่ง ซึ่ง source code ที่มีลักษณะดังรูปที่ 4.5 คือ ไฟล์ BinaryPlus.java ในภาคผนวก ก.

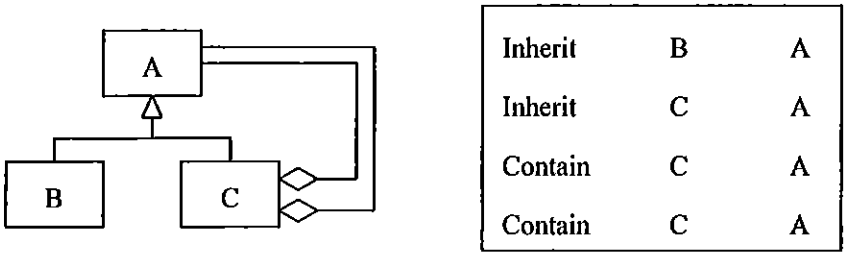
a b

รูปที่ 4.7 a. UML ของ Composite Pattern, b. ภาษา RSF ที่แสดงความสัมพันธ์ของรูป a



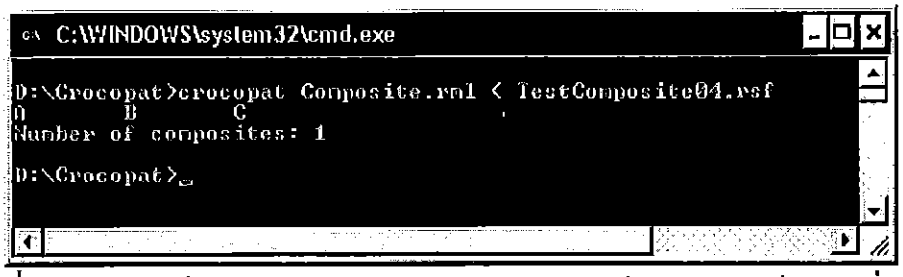
รูปที่ 4.8 ผลลัพธ์จากการรันโปรแกรม Crocopat โดยใช้ความสัมพันธ์ของรูปที่ 4.7

จากรูปที่ 4.7 เป็นโครงสร้างของ Composite Pattern รูปแบบหนึ่ง ซึ่ง source code ที่มีลักษณะดังรูปที่ 4.7 คือ ไฟล์ Employee.java ในภาคผนวก ก.



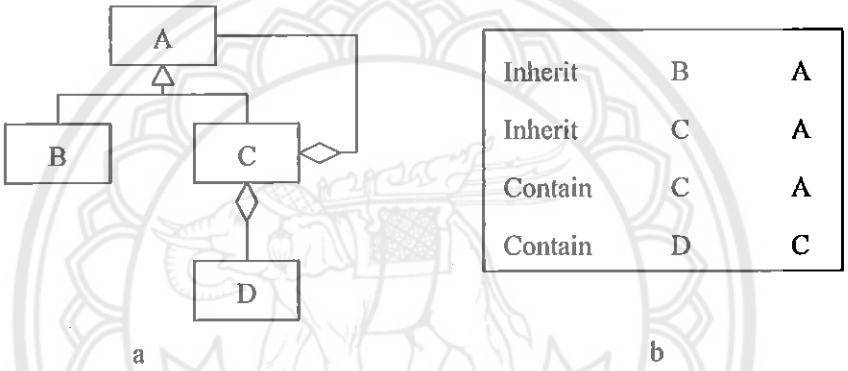
a b

รูปที่ 4.9 a. UML ของ Composite Pattern, b. ภาษา RSF ที่แสดงความสัมพันธ์ของรูป a

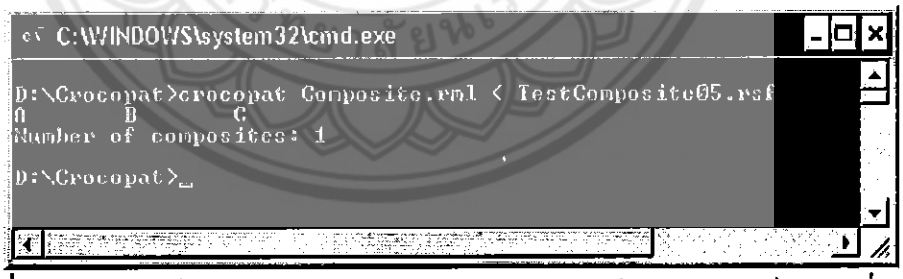


รูปที่ 4.10 ผลลัพธ์จากการรันโปรแกรม Crocopat โดยใช้ความสัมพันธ์ของรูปที่ 4.9

จากรูปที่ 4.9 เป็นโครงสร้างของ Composite Pattern รูปแบบหนึ่ง ซึ่ง source code ที่มีลักษณะดังรูปที่ 4.9 คือ โปรแกรมสร้าง binary tree ซึ่งได้แก่ ไฟล์ BinaryTree.java, Node-BianryTree.java และ EmptyBinaryTree.java ในภาคผนวก ก.

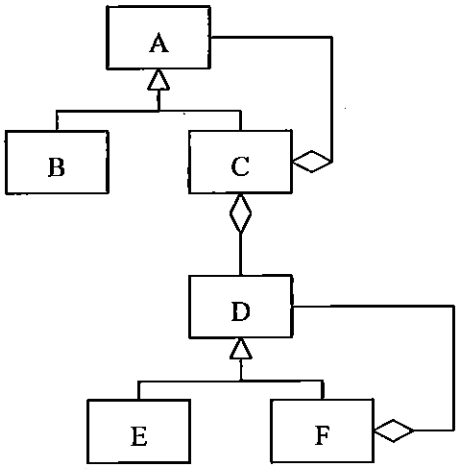


รูปที่ 4.11 a. UML ของ Composite Pattern, b. ภาษา RSF ที่แสดงความสัมพันธ์ของรูป a



รูปที่ 4.12 ผลลัพธ์จากการรันโปรแกรม Crocopat โดยใช้ความสัมพันธ์ของรูปที่ 4.11

จากรูปที่ 4.11 เป็นโครงสร้างของ Composite Pattern รูปแบบหนึ่ง ซึ่ง source code ที่มีลักษณะดังรูปที่ 4.11 คือ โปรแกรม IntList2 ซึ่งประกอบด้วย ไฟล์ IntList2.java, NodeInt-List2.java และ EmptyIntList2.java ในภาคผนวก ก.

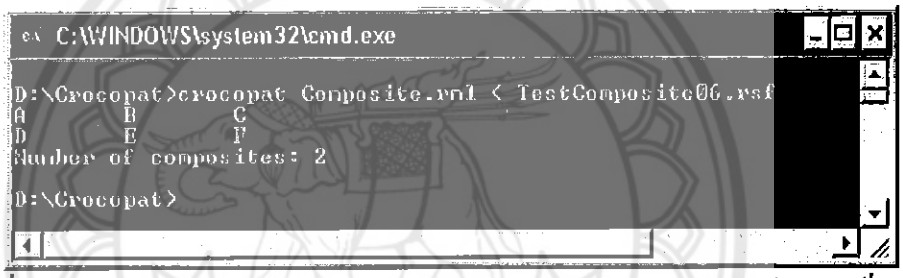


a

Inherit	B	A
Inherit	C	A
Contain	C	A
Contain	D	C
Inherit	E	D
Inherit	F	D
Contain	F	D

b

รูปที่ 4.13 a. UML ของ Composite Pattern, b. ภาษา RSF ที่แสดงความสัมพันธ์ของรูป a



รูปที่ 4.14 ผลลัพธ์จากการรันโปรแกรม Crocopat โดยใช้ความสัมพันธ์ของรูปที่ 4.13

จากรูปที่ 4.13 เป็นโครงสร้างของ Composite Pattern รูปแบบหนึ่ง ซึ่ง source code ที่มีลักษณะดังรูปที่ 4.13 คือ โปรแกรม IntList3 ซึ่งประกอบด้วย ไฟล์ IntList3.java, NodeIntList3.java และ EmptyIntList3.java ในภาคผนวก ก.

Composite Pattern ที่มีโครงสร้างแตกต่างกัน ดังรูปที่ 4.5, 4.7, 4.9, 4.11 และ 4.13 จึงได้นำโครงสร้างและความสัมพันธ์แบบต่างๆ มาทดสอบ โดยใช้โปรแกรมภาษา RML ของ Composite Pattern รูปแบบทั่วไป (รูปที่ 4.2) ผลปรากฏว่า โครงสร้าง pattern บางรูปแบบ โปรแกรม Crocopat ไม่สามารถระบุได้ว่าเป็นโครงสร้างแบบ Composite Pattern ซึ่งมีผลมาจาก

1. จำนวนของ node มีมากกว่าหรือน้อยกว่าโครงสร้างแบบ Composite Pattern รูปแบบทั่วไป (รูปที่ 4.1)

2. ความสัมพันธ์บางอย่างอาจหายไป ส่งผลให้โปรแกรมไม่สามารถระบุได้ว่าเป็นโครงสร้างแบบ Composite Pattern เช่น รูปที่ 4.5 และรูปที่ 4.7 เป็นต้น

ดังนั้น ถ้าต้องการให้โปรแกรม Crocopat สามารถระบุได้ว่าเป็นโครงสร้างแบบ Composite Pattern จะต้องมีการปรับปรุงฐานข้อมูลโครงสร้าง RML โดยเขียนเพิ่มขึ้นอีก 3

รูปแบบ โดยรูปที่ 4.15 จะเป็นโครงสร้าง RML สำหรับตรวจหา Composite Pattern ที่มีโครงสร้างเหมือนกับรูปที่ 4.5 และรูปที่ 4.16 จะเป็นโครงสร้าง RML สำหรับตรวจหา Composite Pattern ที่มีโครงสร้างเหมือนกับรูปที่ 4.7


```
Composite2 (Component, Composite) :=
                    Inherit (Composite, Component)
                  & Contain (Composite, Component);
PRINT "Composite Type2: " , #(Composite2(Component,_,_,_)) , ";" , ENDL;
```

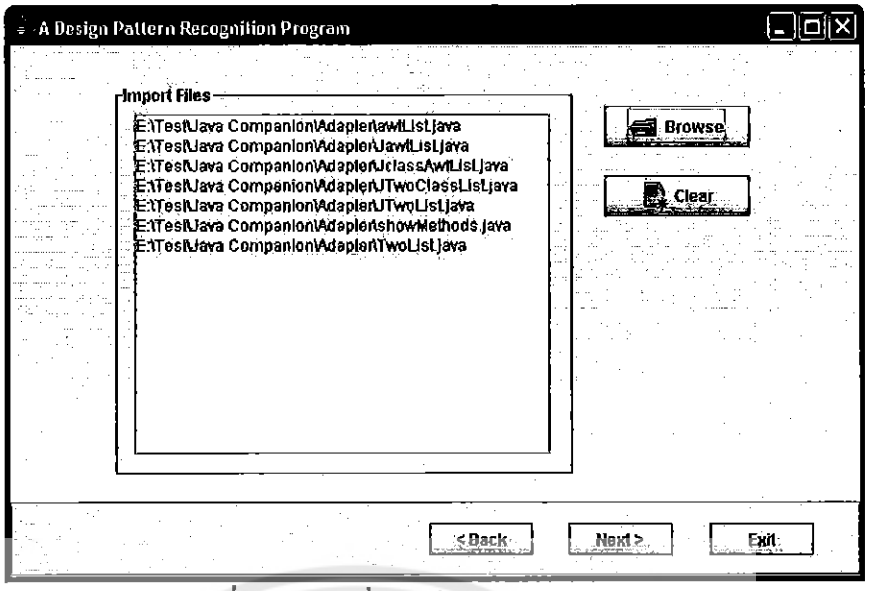

รูปที่ 4.15 โครงสร้าง RML สำหรับตรวจหา Composite Pattern ที่มีโครงสร้างเหมือนกับ รูปที่ 4.5

```
Composite3 (Component) :=
                    Contain (Component, Component);
PRINT "Composite Type3: " , #(Composite3(Component,_,_,_)) , ";" , ENDL;
```

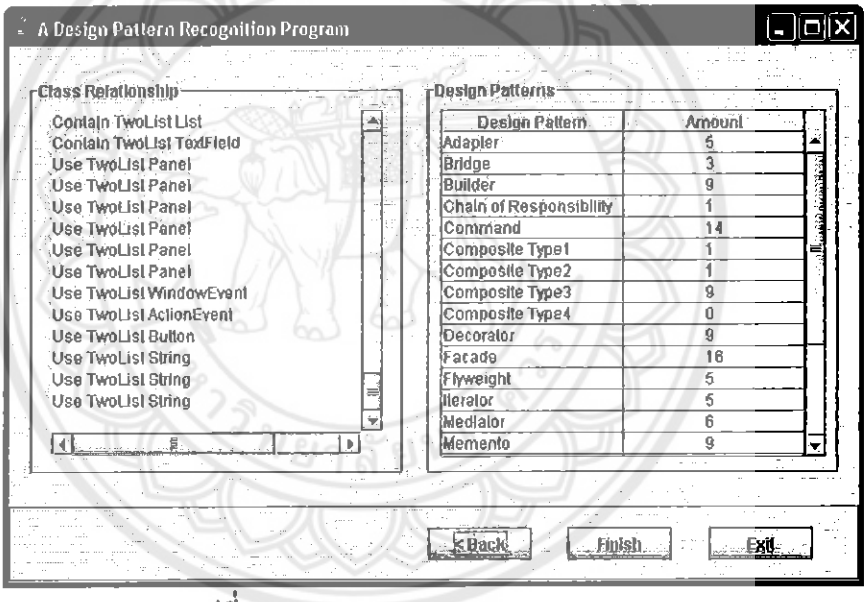
รูปที่ 4.16 โครงสร้าง RML สำหรับตรวจหา Composite Pattern ที่มีโครงสร้างเหมือนกับรูปที่ 4.7

4.2 การทดสอบโปรแกรม Design Pattern Recognition

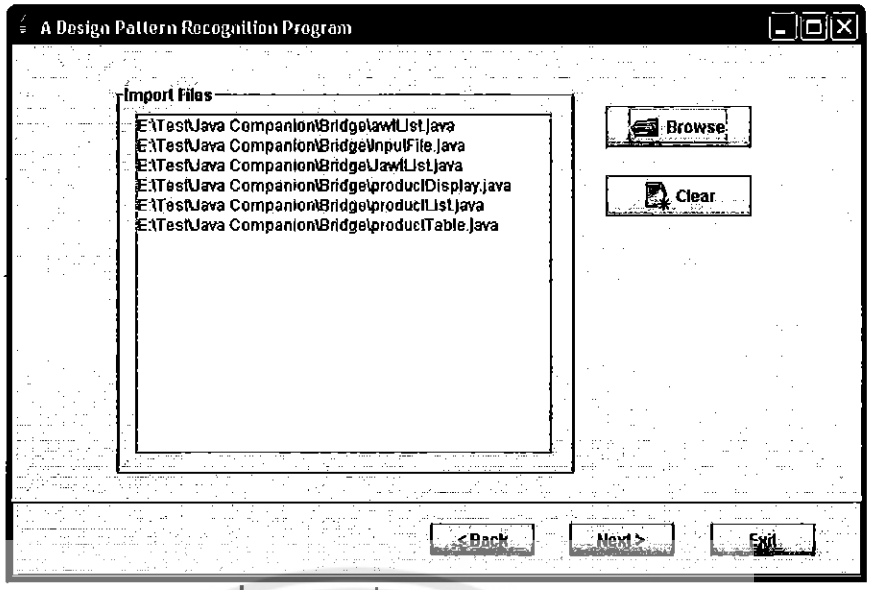
4.2.1 ทดสอบโปรแกรมโดยใช้ source code จากหนังสือ Design Patterns Java Companion ของ James W. Cooper ซึ่งผลการทดสอบโปรแกรมจะแสดงผลลัพธ์จำนวน 2 รูปต่อการทดสอบ 1 โปรแกรม โดยรูปลำดับที่ 1 จะแสดงรายชื่อไฟล์ที่ใช้ทดสอบ ส่วนรูปลำดับที่ 2 แสดงผลลัพธ์ของการตรวจหา Design Pattern



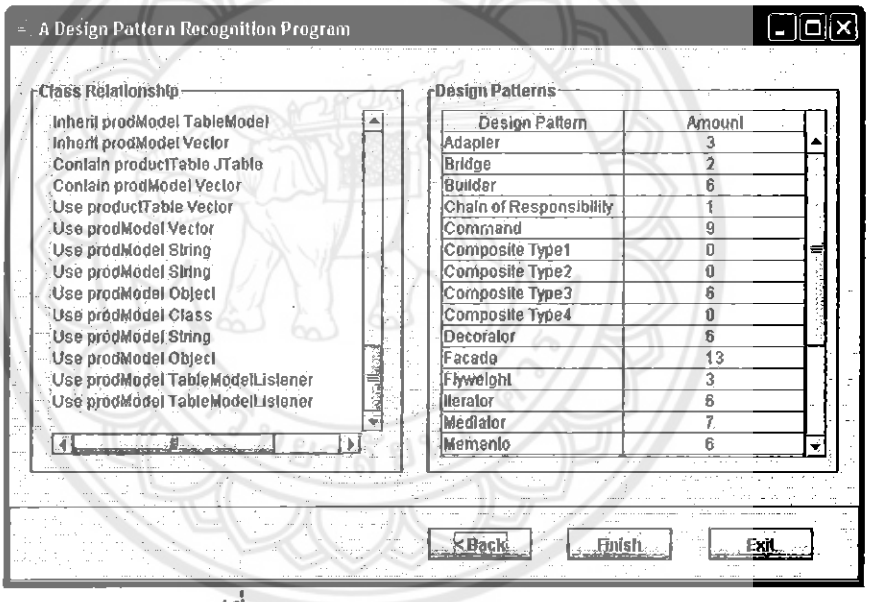
รูปที่ 4.17 รายชื่อไฟล์ของ Adapter Pattern



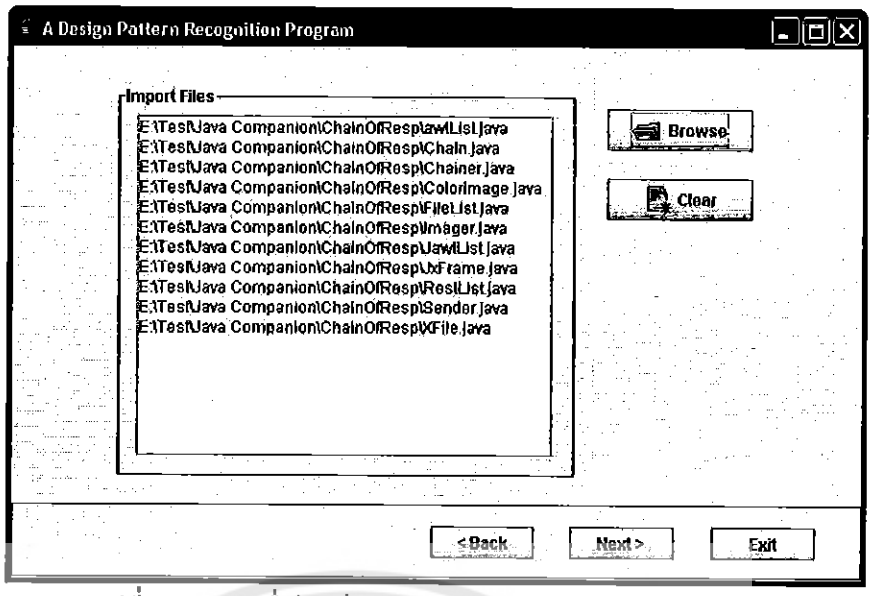
รูปที่ 4.18 ผลการทดสอบ Adapter Pattern



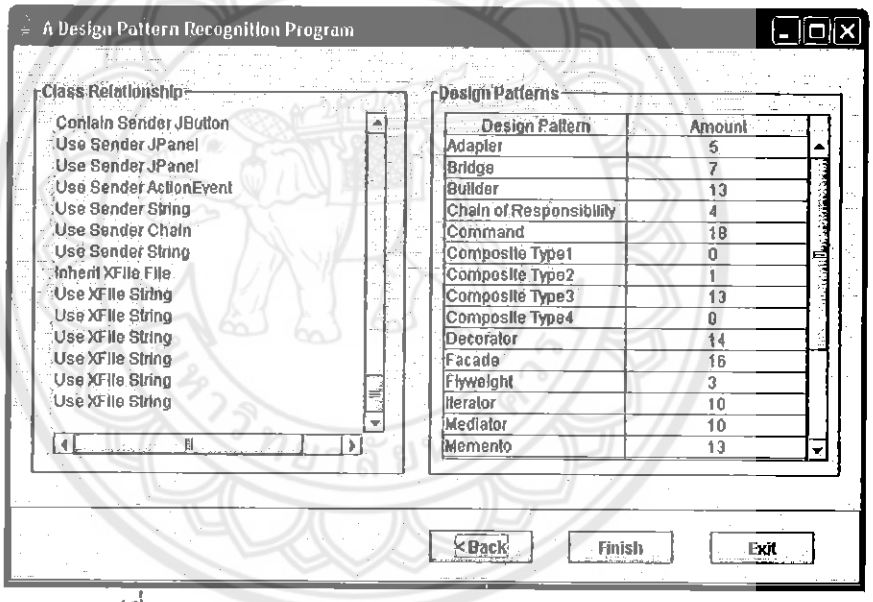
รูปที่ 4.19 รายชื่อไฟล์ของ Bridge Pattern



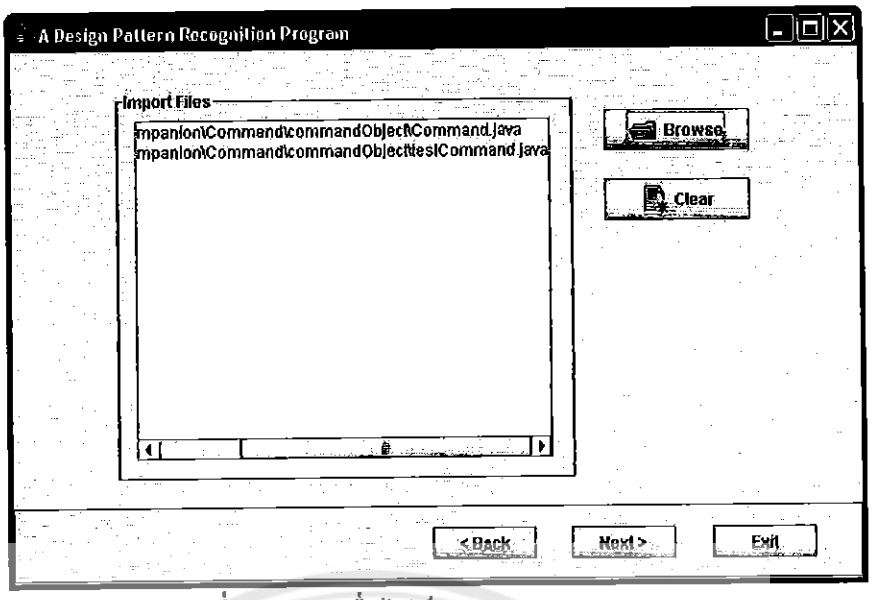
รูปที่ 4.20 ผลการทดสอบ Bridge Pattern



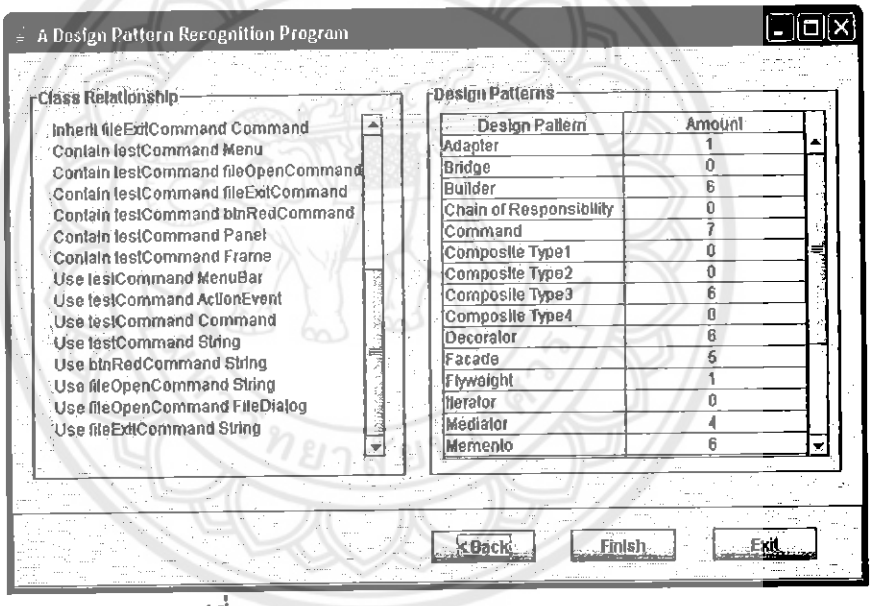
รูปที่ 4.21 รายชื่อไฟล์ของ Chain of Responsibility Pattern



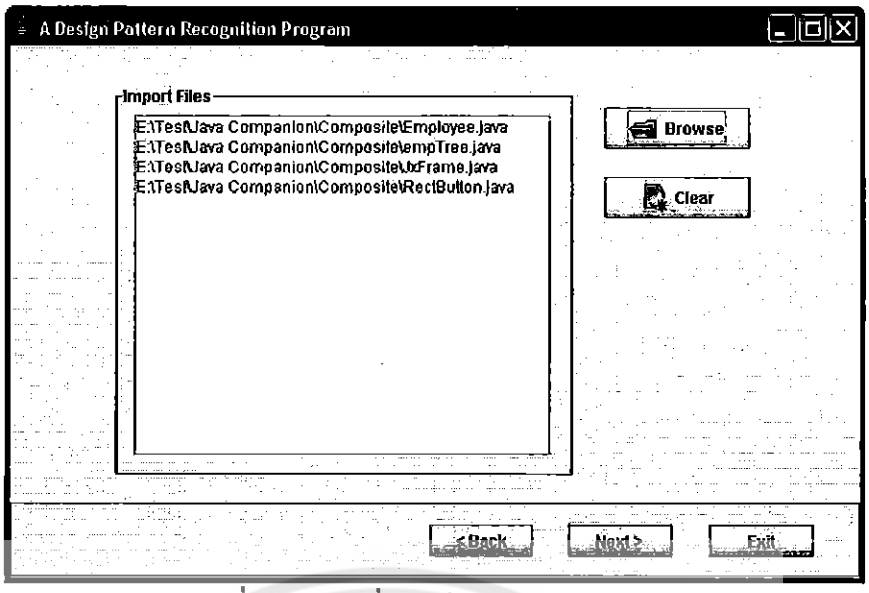
รูปที่ 4.22 ผลการทดสอบ Chain of Responsibility Pattern



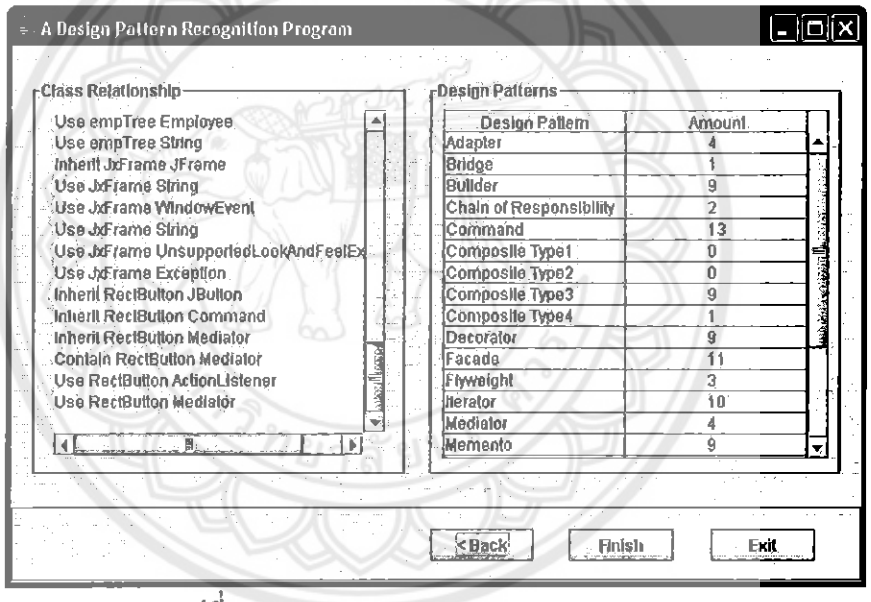
รูปที่ 4.23 รายชื่อไฟล์ของ Command Pattern



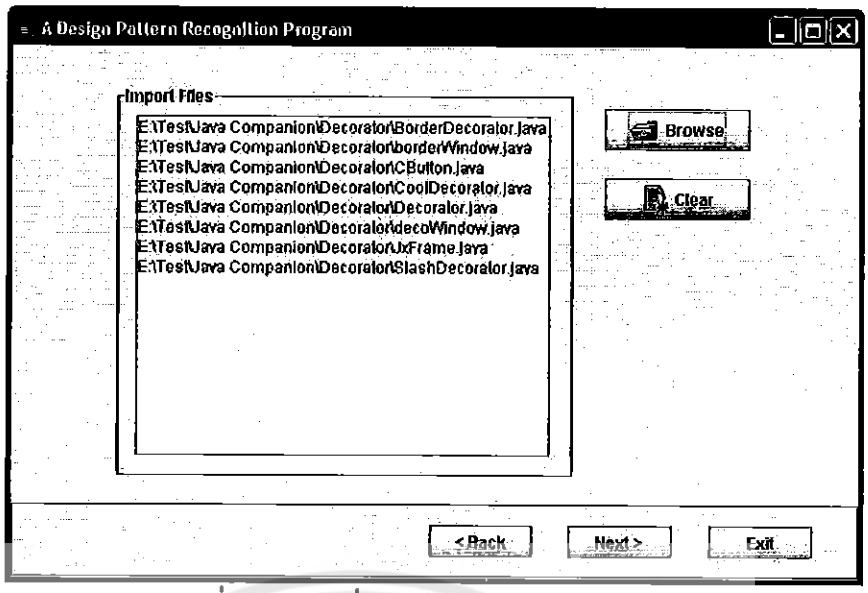
รูปที่ 4.24 ผลการทดสอบ Command Pattern



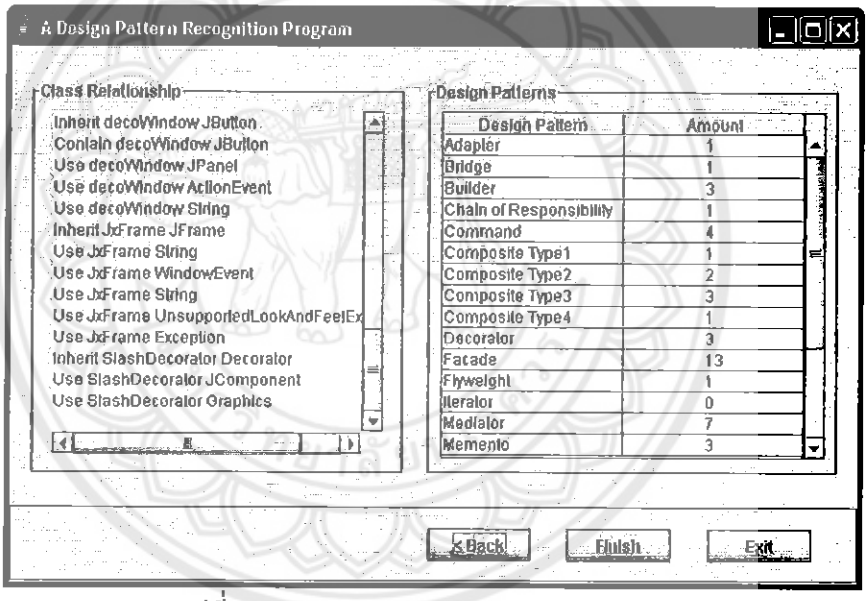
รูปที่ 4.25 รายชื่อไฟล์ของ Composite Pattern



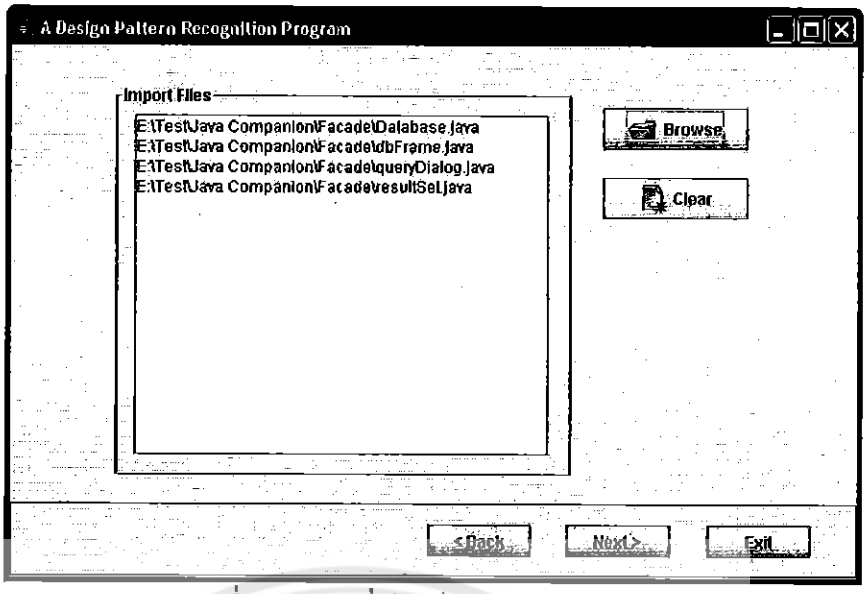
รูปที่ 4.26 ผลการทดสอบ Composite Pattern



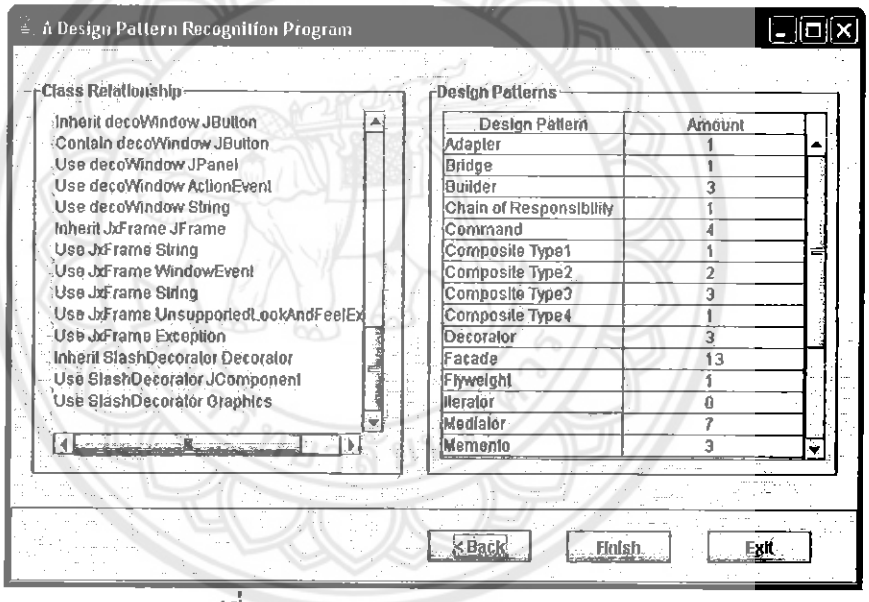
รูปที่ 4.27 รายชื่อไฟล์ของ Decorator Pattern



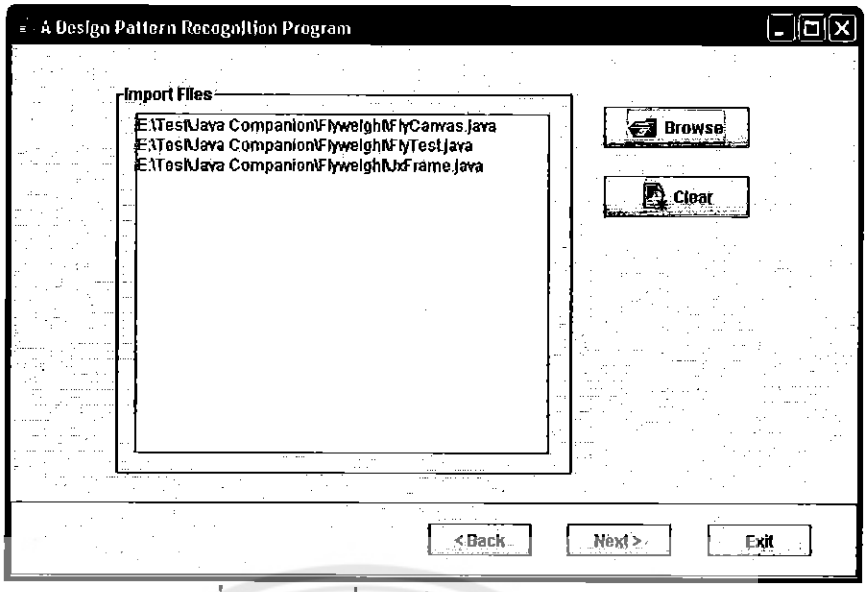
รูปที่ 4.28 ผลการทดสอบ Decorator Pattern



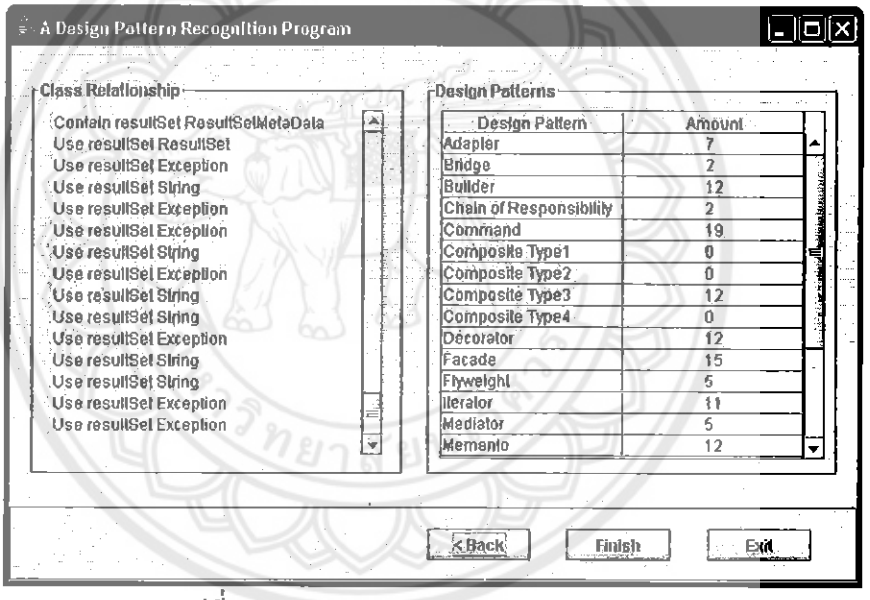
รูปที่ 4.29 รายชื่อไฟล์ของ Façade Pattern



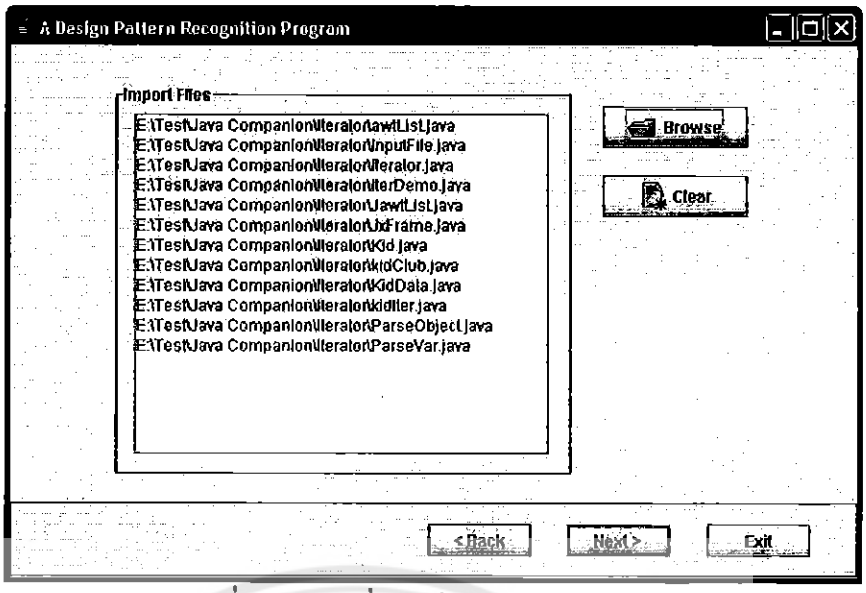
รูปที่ 4.30 ผลการทดสอบ Façade Pattern



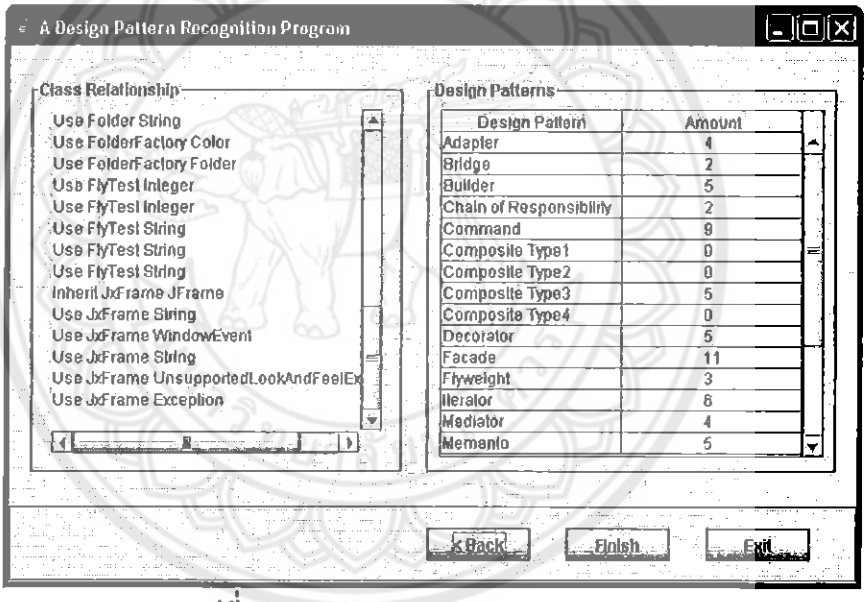
รูปที่ 4.31 รายชื่อไฟล์ของ Flyweight Pattern



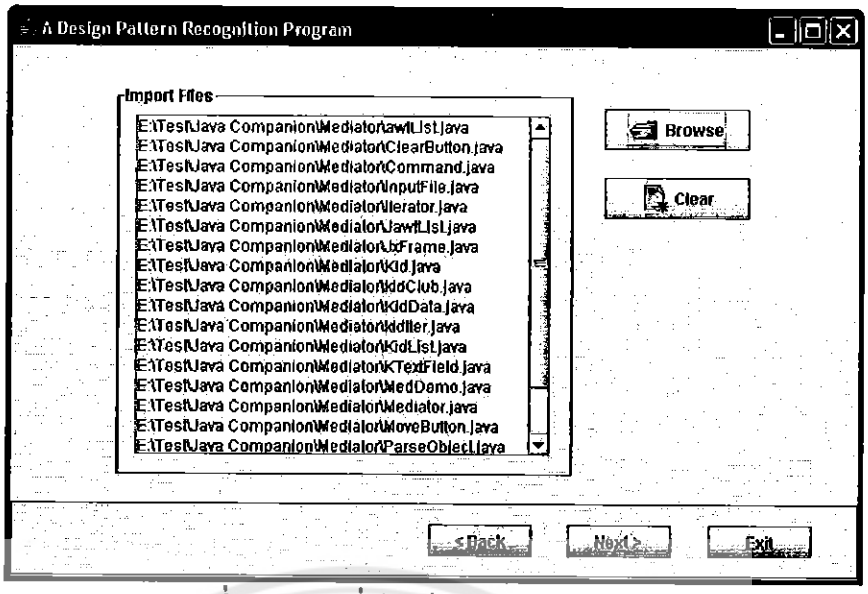
รูปที่ 4.32 ผลการทดสอบ Flyweight Pattern



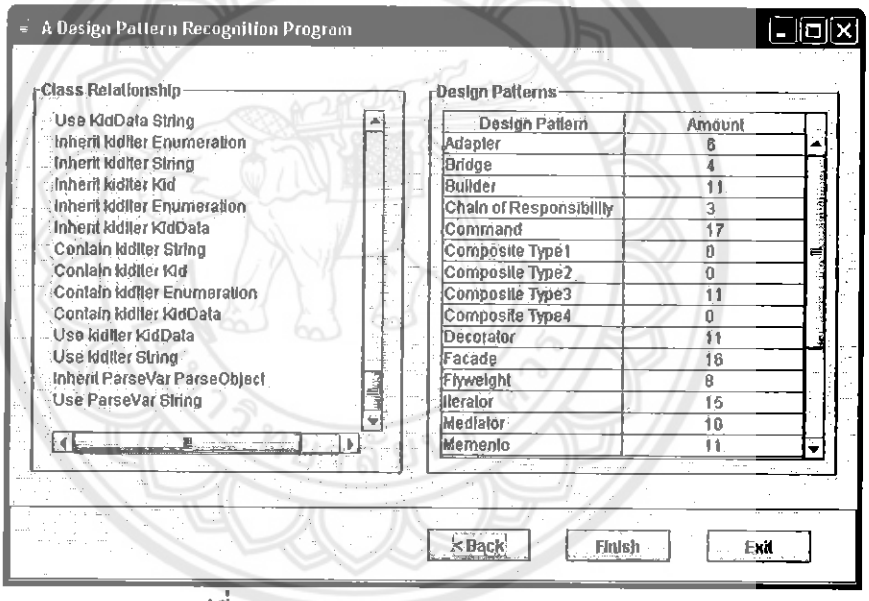
รูปที่ 4.33 รายชื่อไฟล์ของ Iterator Pattern



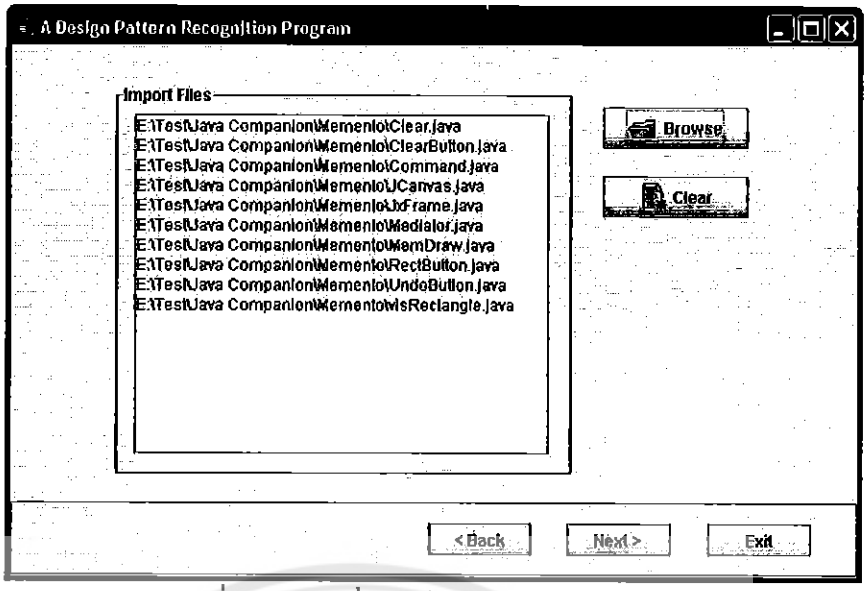
รูปที่ 4.34 ผลการทดสอบ Iterator Pattern



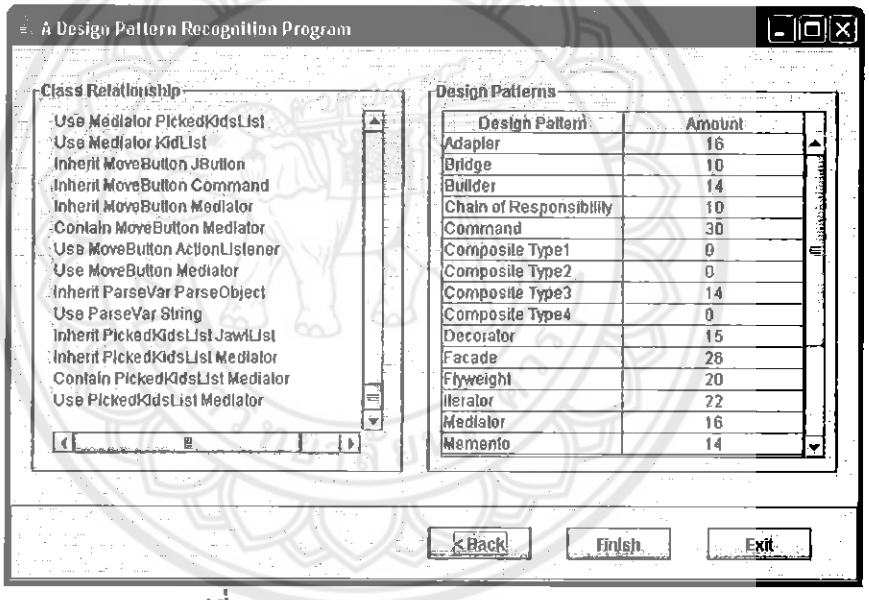
รูปที่ 4.35 รายชื่อไฟล์ของ Mediator Pattern



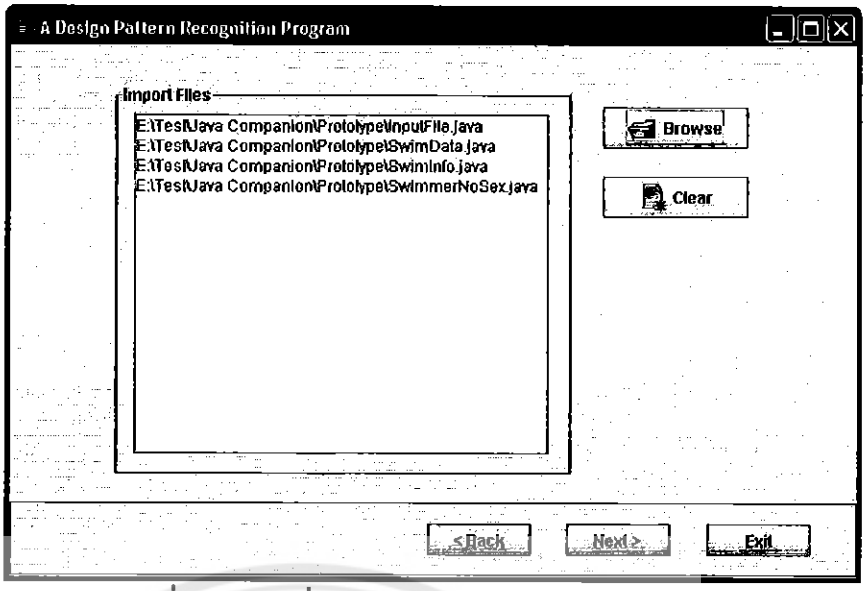
รูปที่ 4.36 ผลการทดสอบ Mediator Pattern



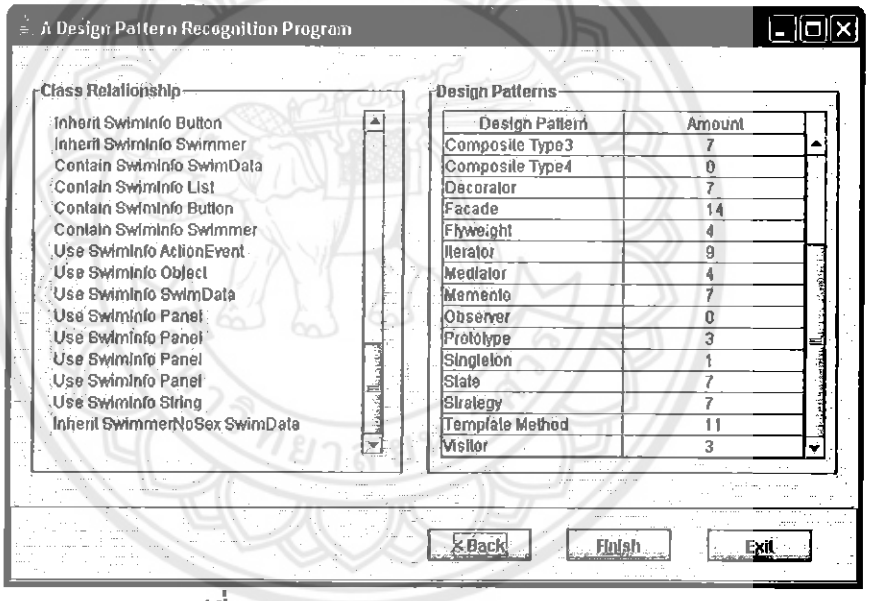
รูปที่ 4.37 รายชื่อไฟล์ของ Memento Pattern



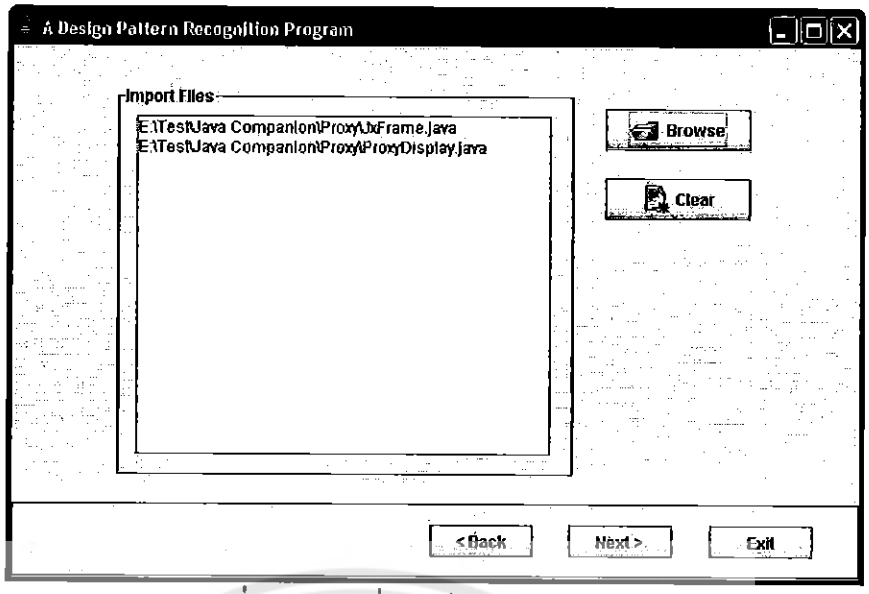
รูปที่ 4.38 ผลการทดสอบ Memento Pattern



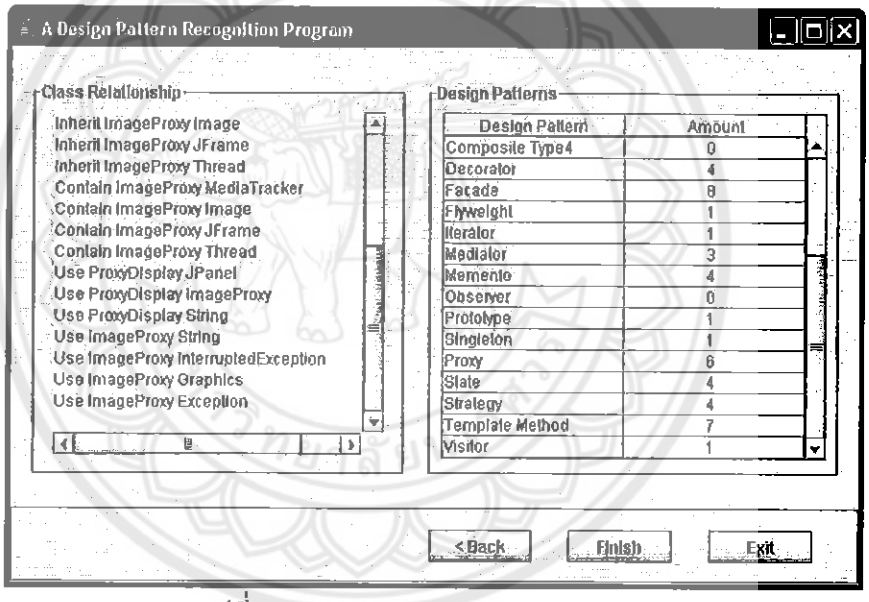
รูปที่ 4.39 รายชื่อไฟล์ของ Prototype Pattern



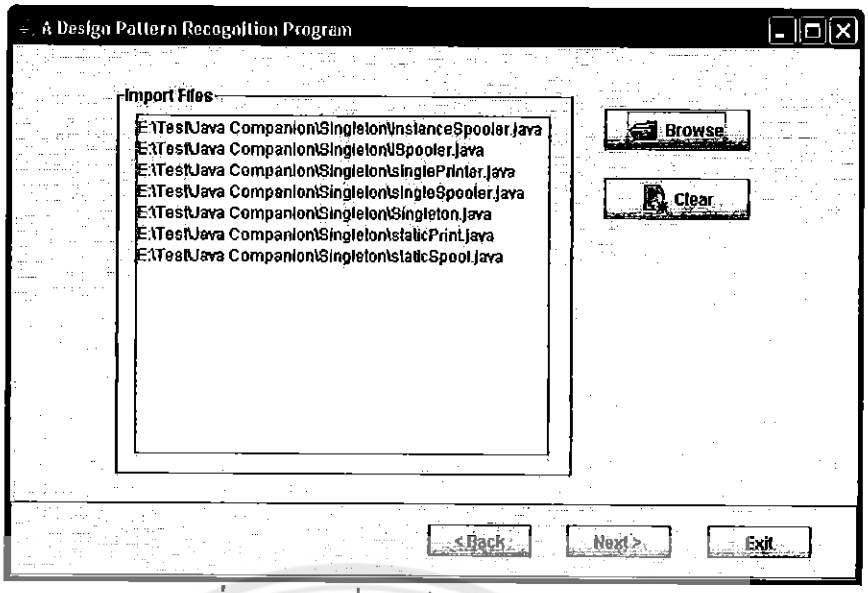
รูปที่ 4.40 ผลการทดสอบ Prototype Pattern



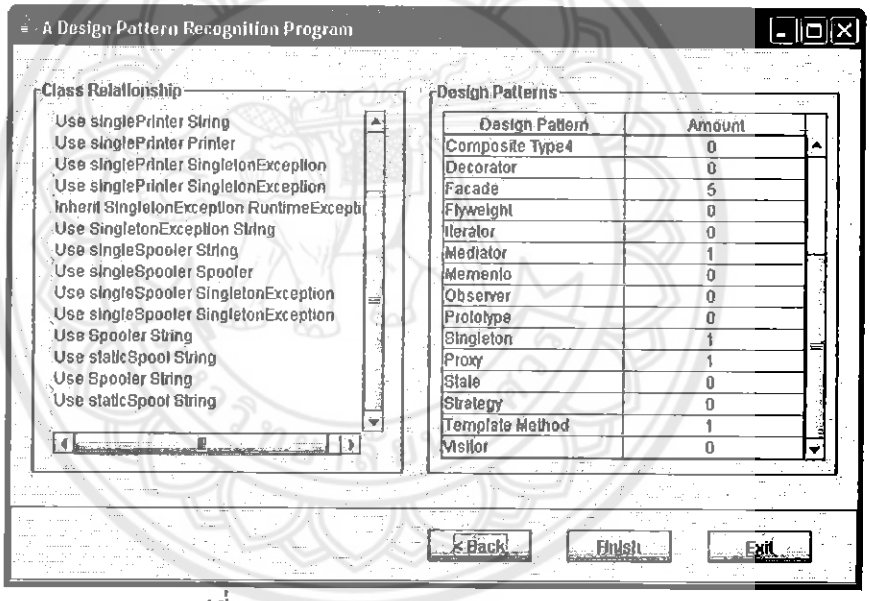
รูปที่ 4.41 รายชื่อไฟล์ของ Proxy Pattern



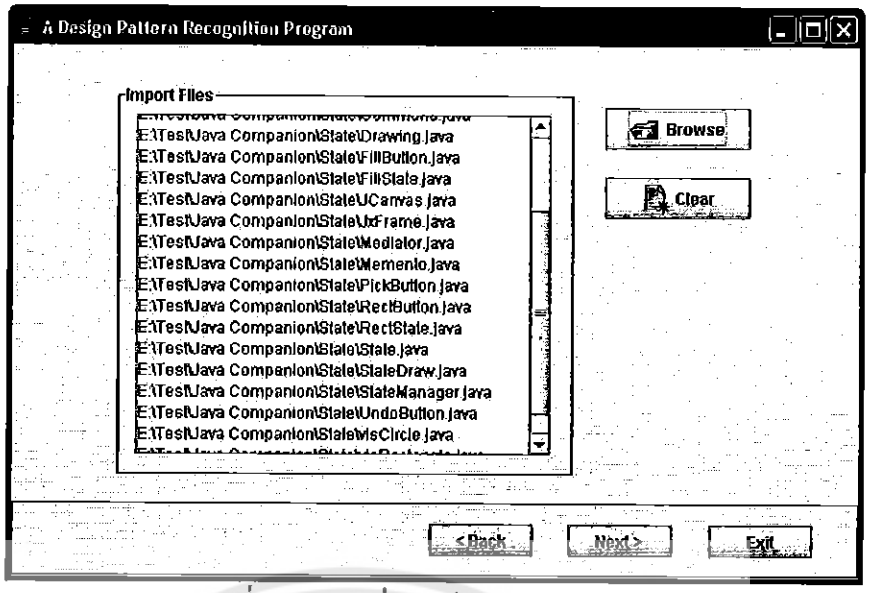
รูปที่ 4.42 ผลการทดสอบ Proxy Pattern



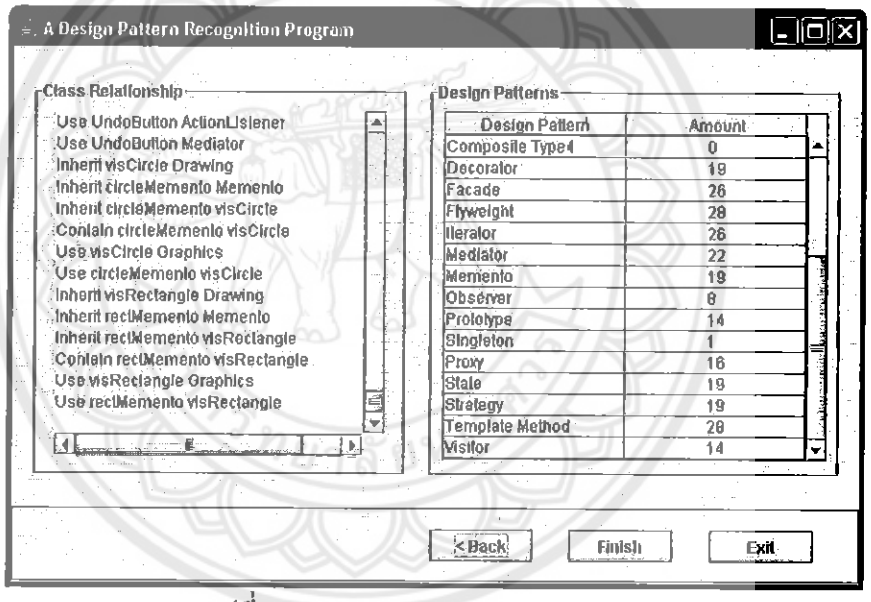
รูปที่ 4.43 รายชื่อไฟล์ของ Singleton Pattern



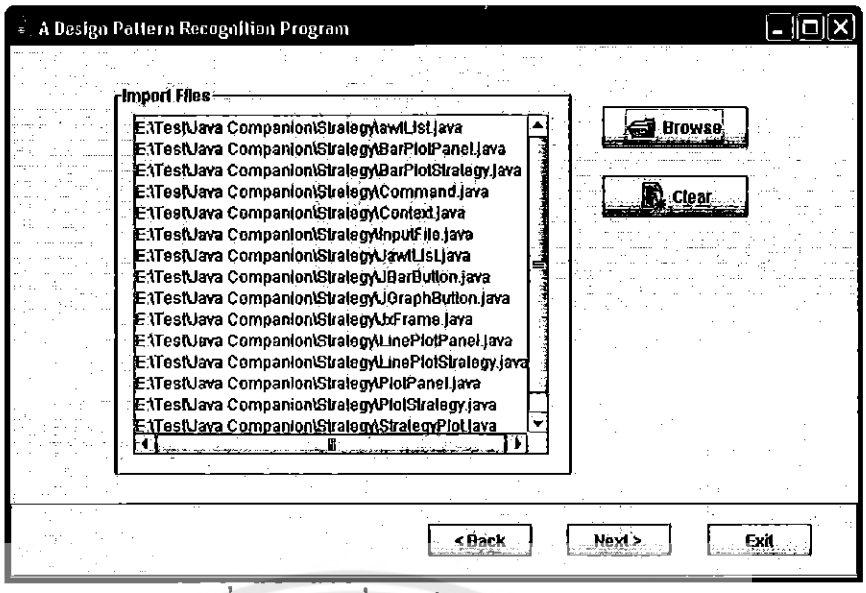
รูปที่ 4.44 ผลการทดสอบ Singleton Pattern



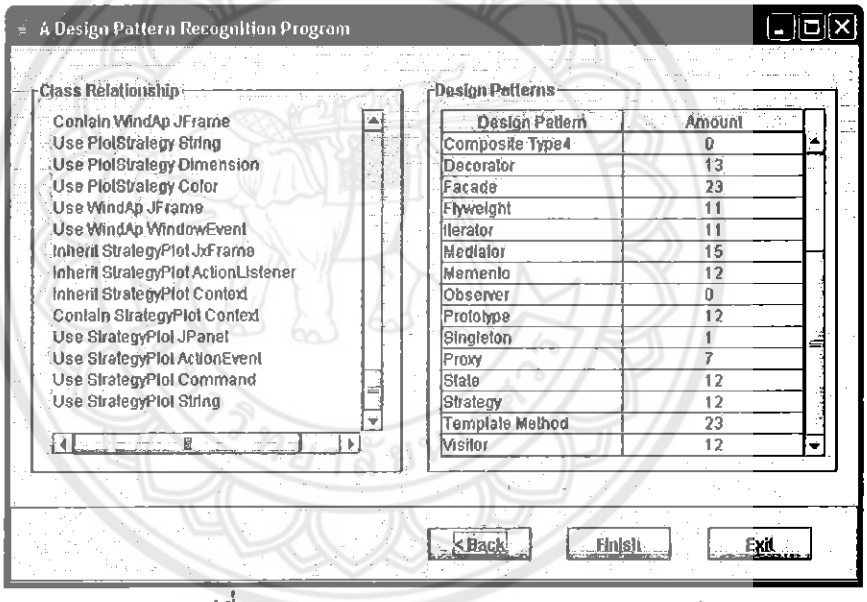
รูปที่ 4.45 รายชื่อไฟล์ของ State Pattern



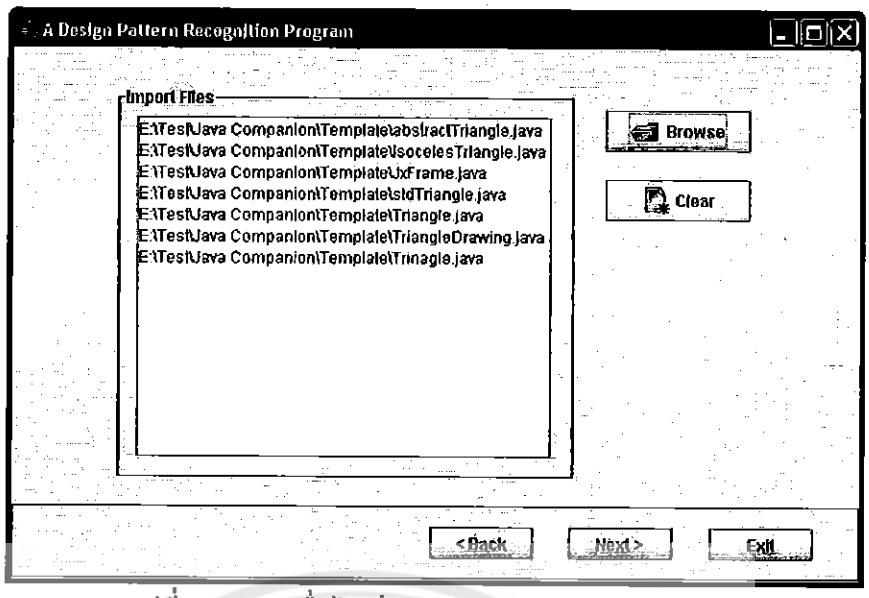
รูปที่ 4.46 ผลการทดสอบ State Pattern



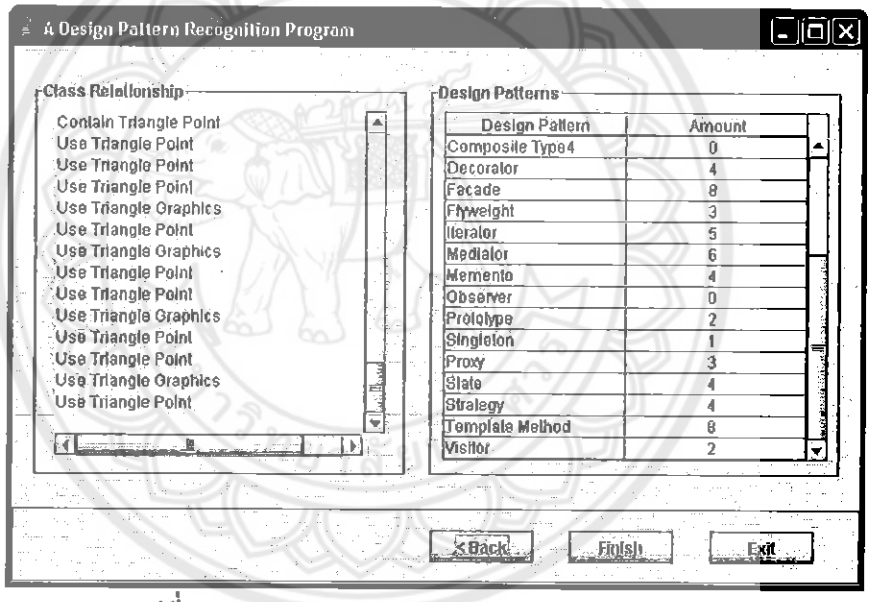
รูปที่ 4.47 รายชื่อไฟล์ของ Strategy Pattern



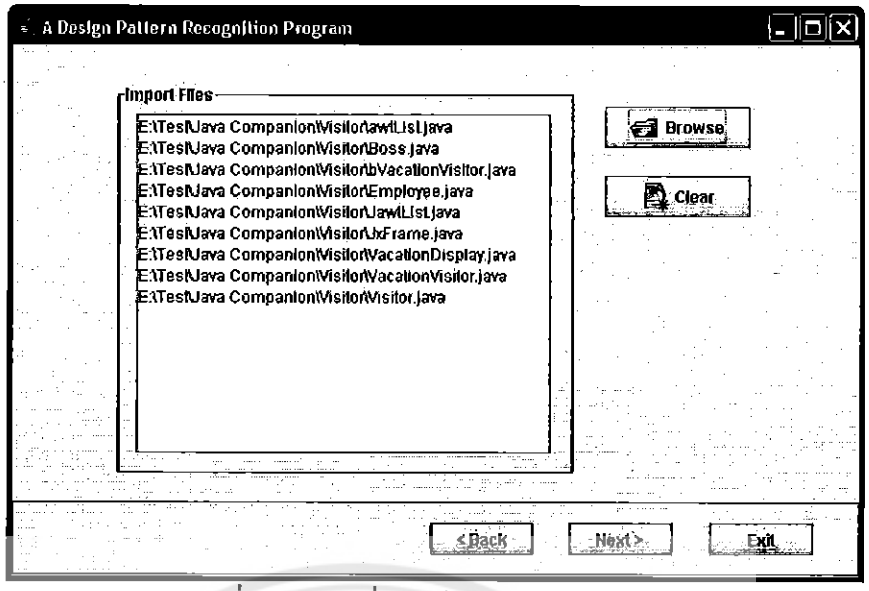
รูปที่ 4.48 ผลการทดสอบ Strategy Pattern



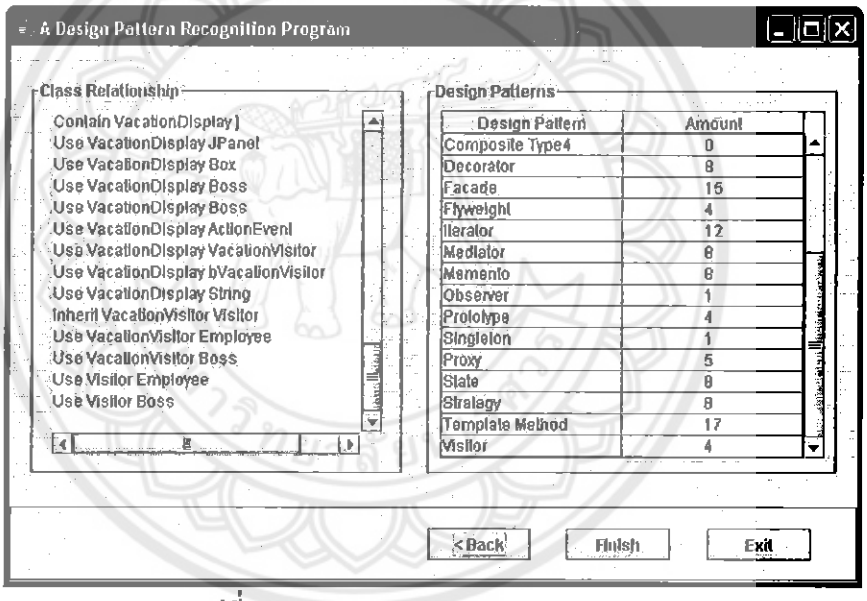
รูปที่ 4.49 รายชื่อไฟล์ของ Templete Method Pattern



รูปที่ 4.50 ผลการทดสอบ Templete Method Pattern

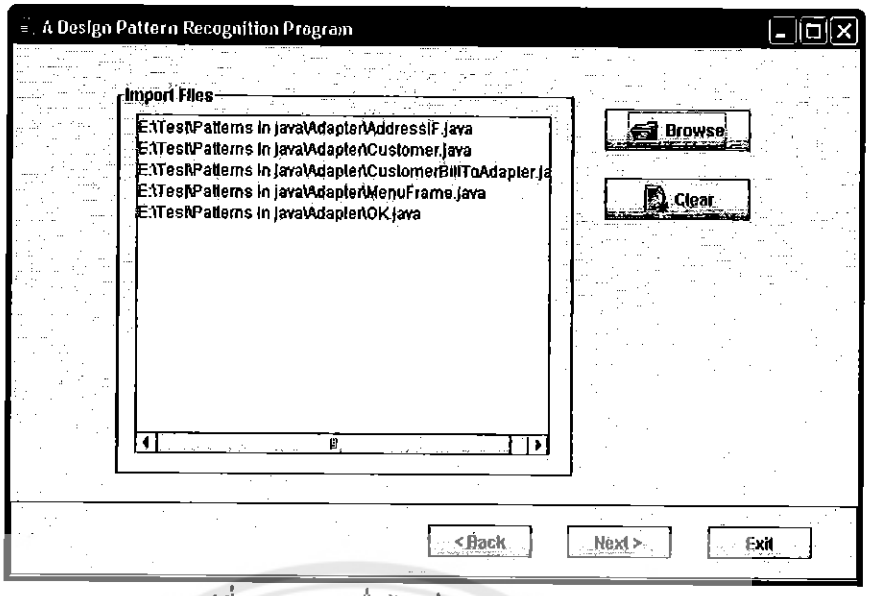


รูปที่ 4.51 รายชื่อไฟล์ของ Visitor Pattern

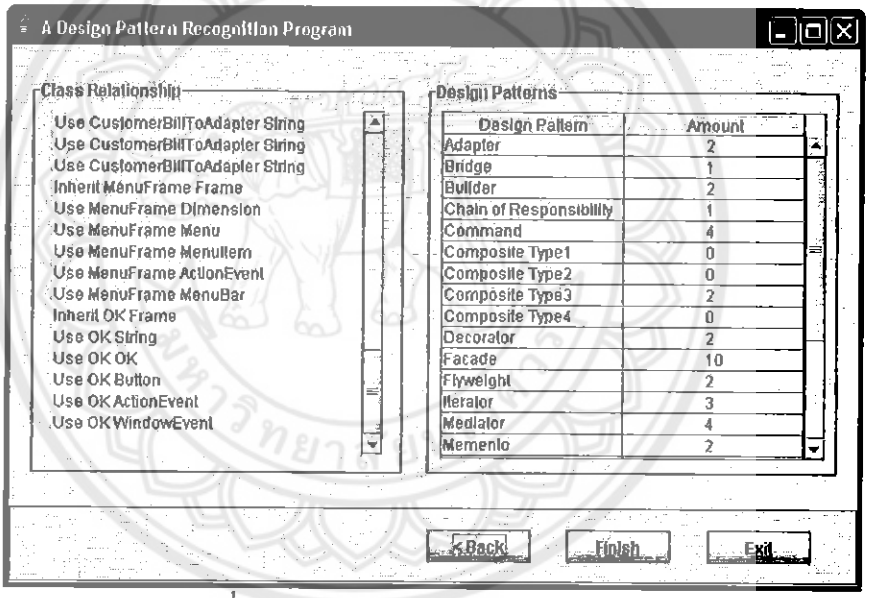


รูปที่ 4.52 ผลการทดสอบ Visitor Pattern

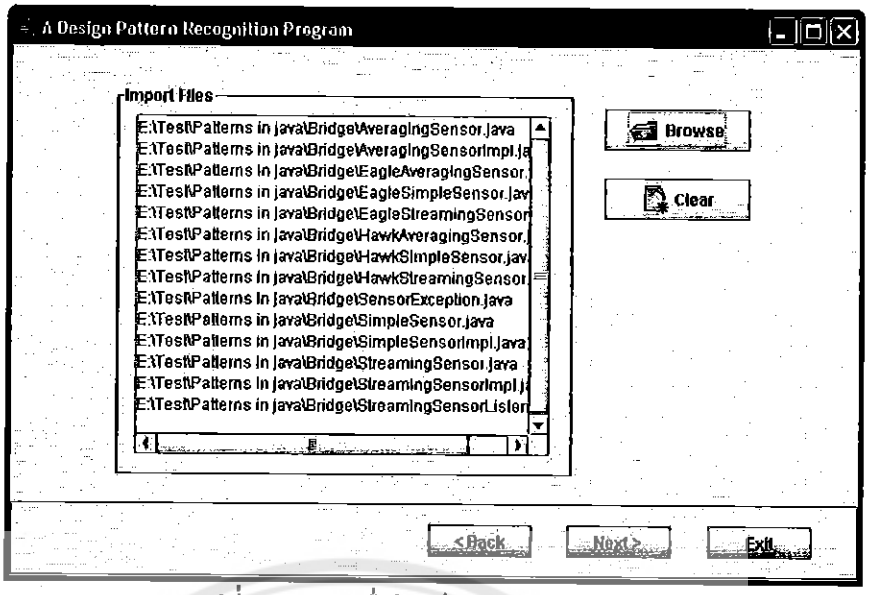
4.2.2 ทดสอบโปรแกรมโดยใช้ source code จากหนังสือ Patterns in Java ของ Mark Grand ซึ่งผลการทดสอบโปรแกรมจะแสดงผลลัพธ์จำนวน 2 รูป ต่อการทดสอบ 1 โปรแกรม โดยรูปลำดับที่ 1 จะแสดงรายชื่อไฟล์ที่ใช้ทดสอบ ส่วนรูปลำดับที่ 2 แสดงผลลัพธ์ของการตรวจหา Design Pattern



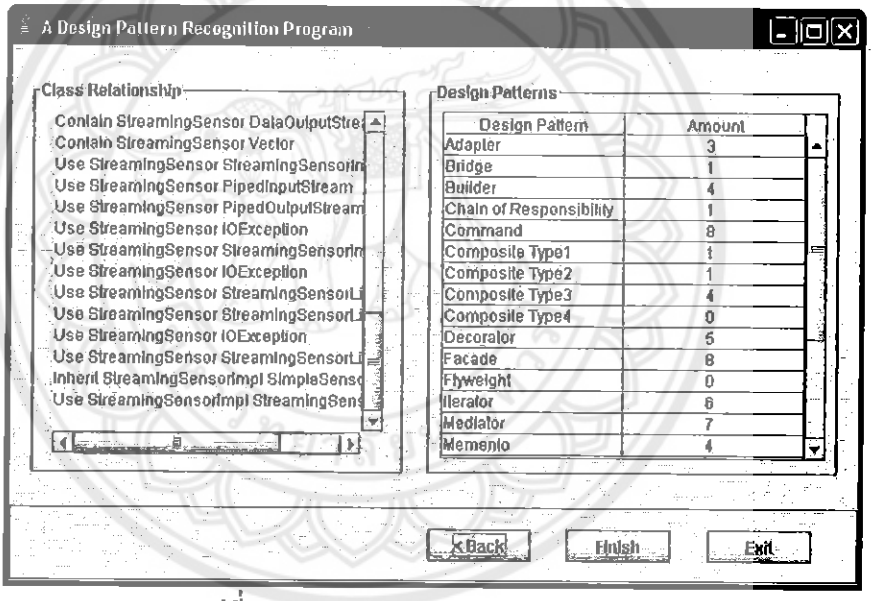
รูปที่ 4.53 รายชื่อไฟล์ของ Adapter Pattern



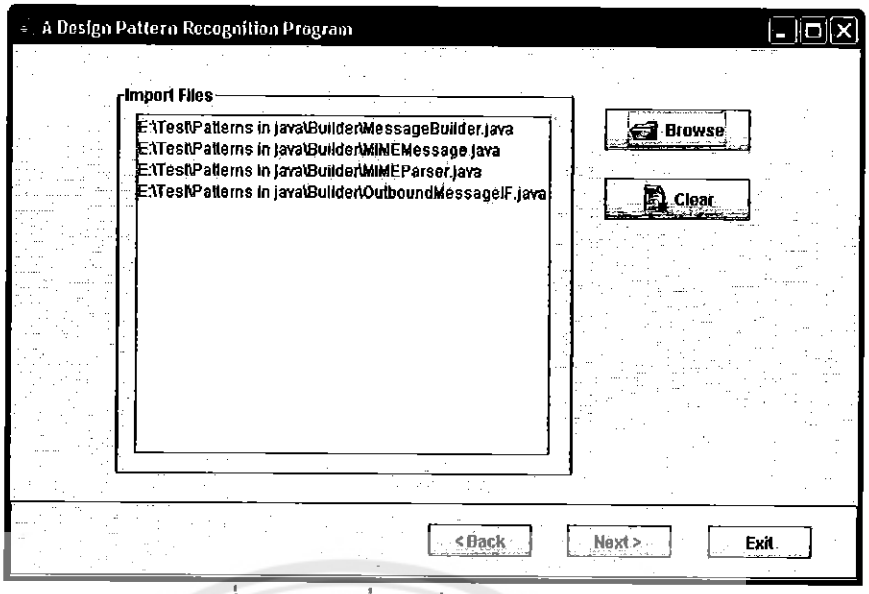
รูปที่ 4.54 ผลการทดสอบ Adapter Pattern



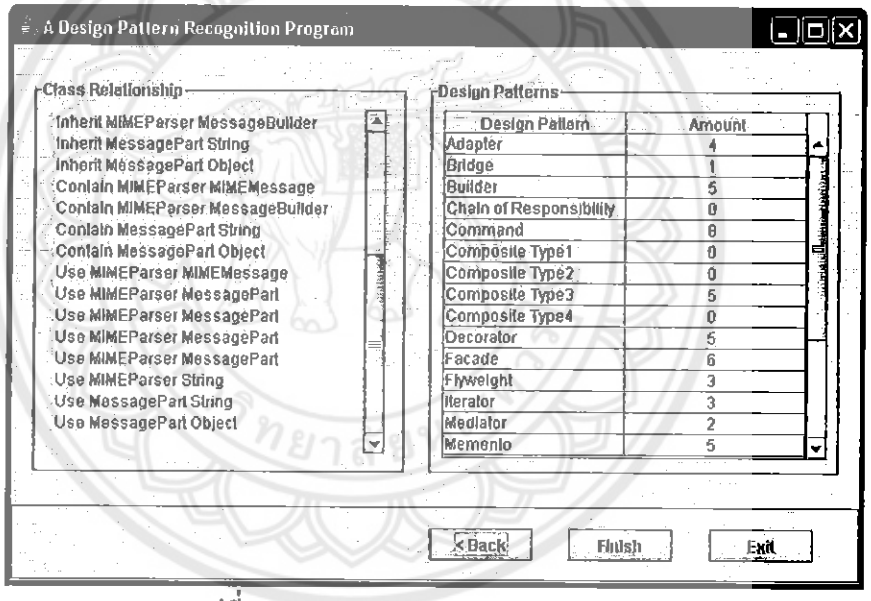
รูปที่ 4.55 รายชื่อไฟล์ของ Bridge Pattern



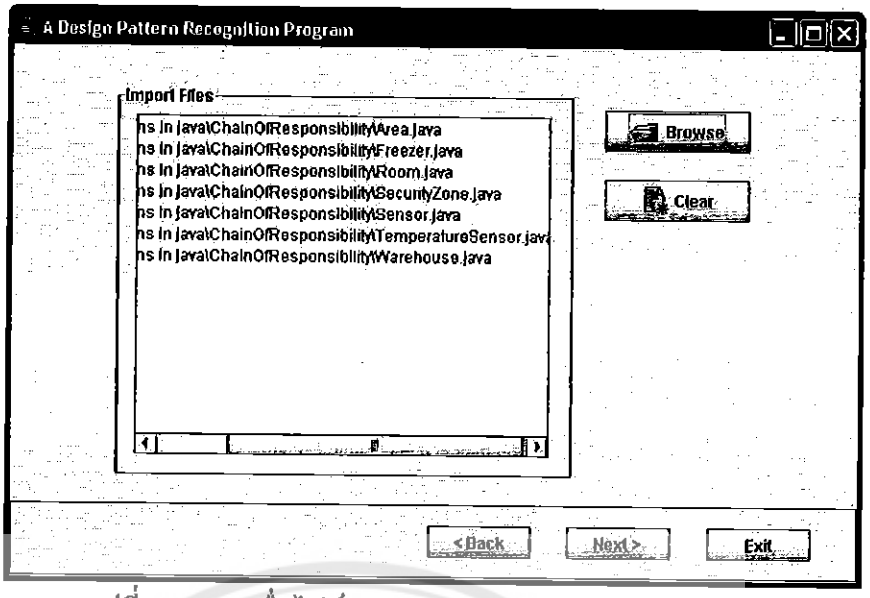
รูปที่ 4.56 ผลการทดสอบ Bridge Pattern



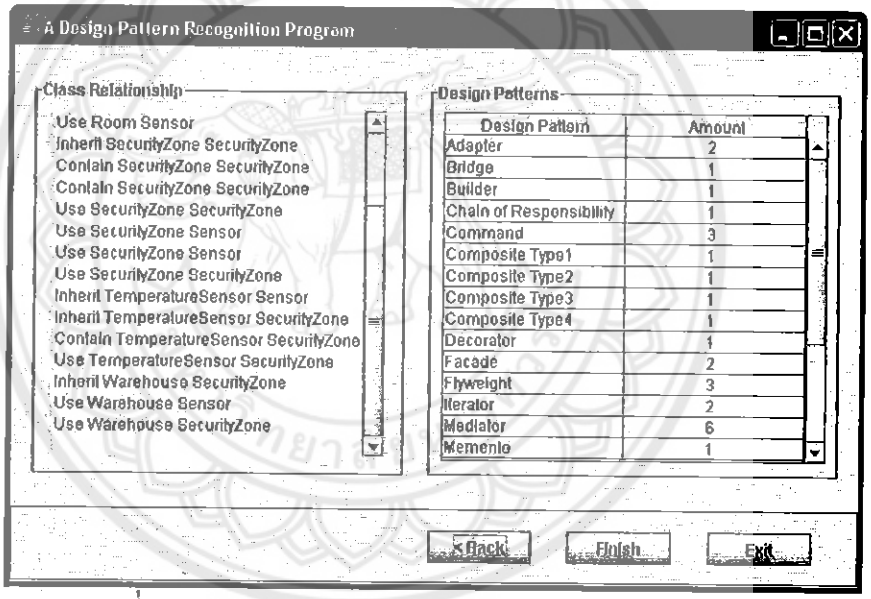
รูปที่ 4.57 รายชื่อไฟล์ของ Builder Pattern



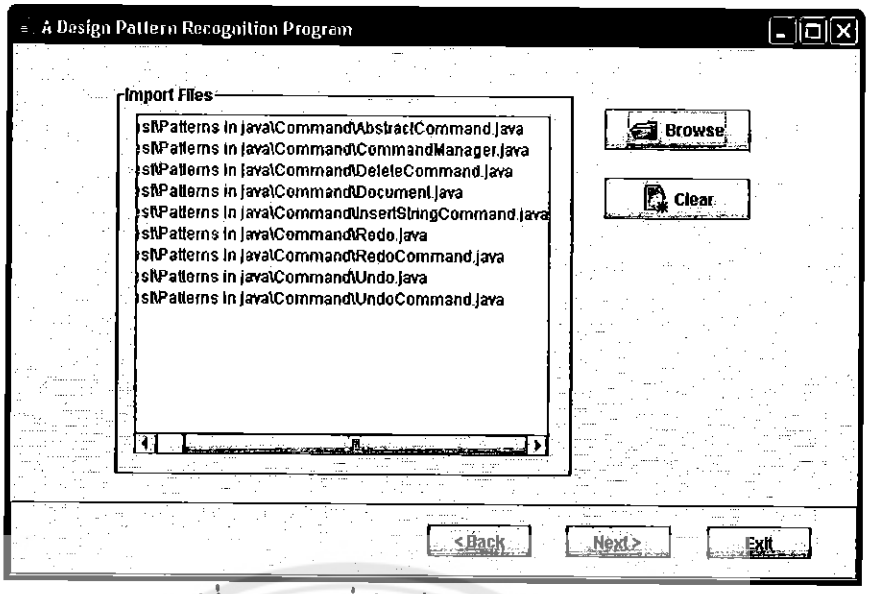
รูปที่ 4.58 ผลการทดสอบ Builder Pattern



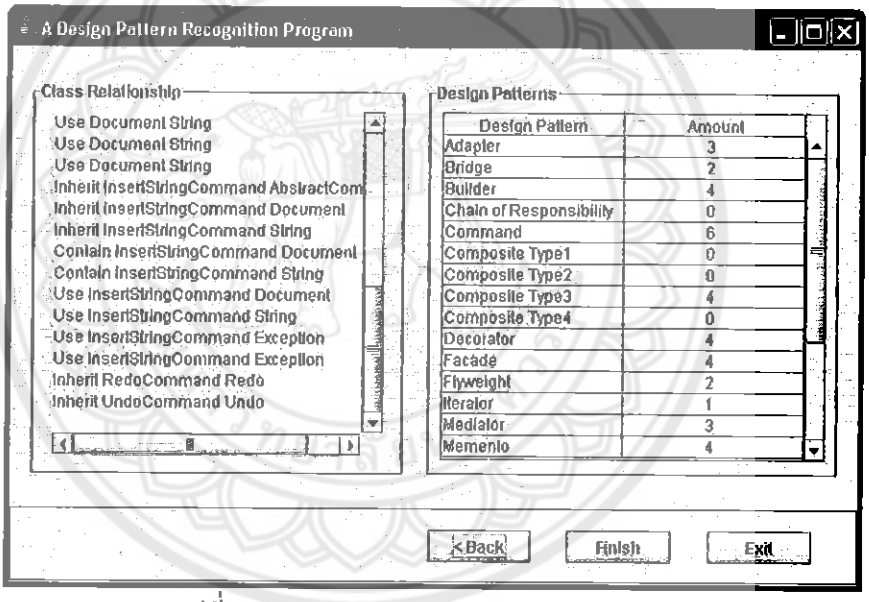
รูปที่ 4.59 รายชื่อไฟล์ของ Chain of Responsibility Pattern



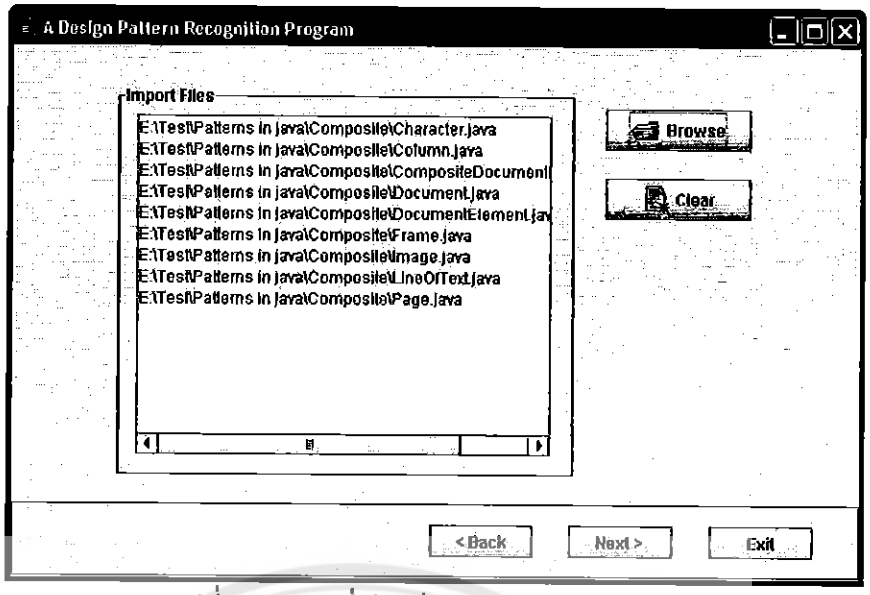
รูปที่ 4.60 ผลการทดสอบ Chain of Responsibility Pattern



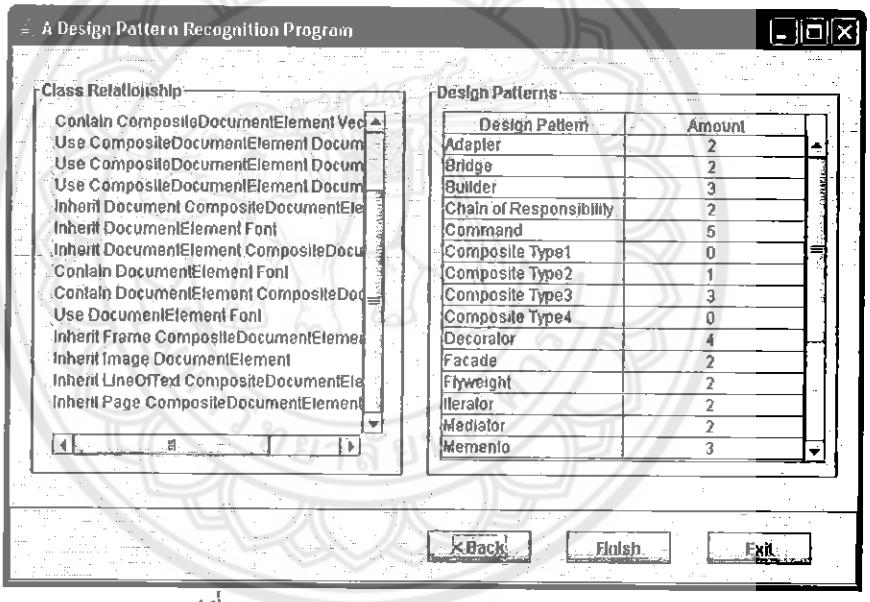
รูปที่ 4.61 รายชื่อไฟล์ของ Command Pattern



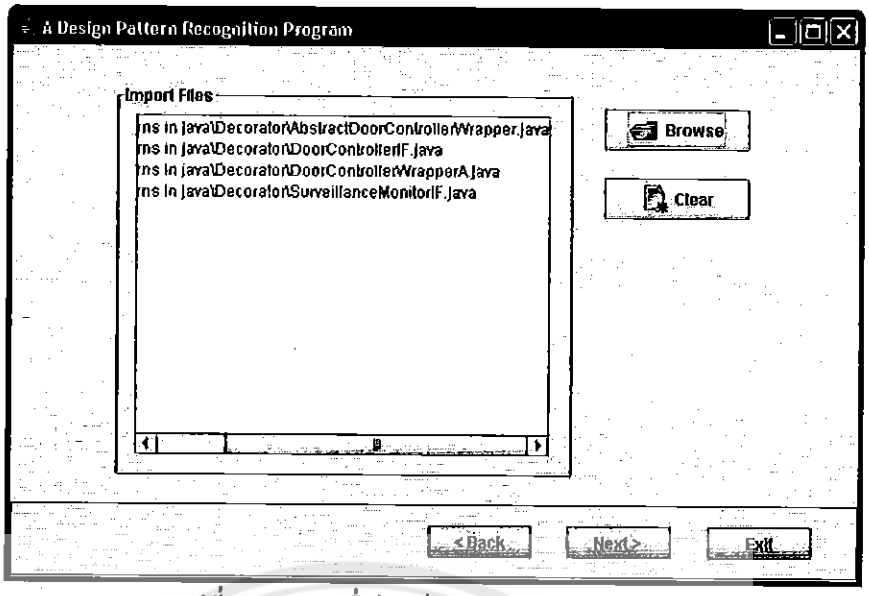
รูปที่ 4.62 ผลการทดสอบ Command Pattern



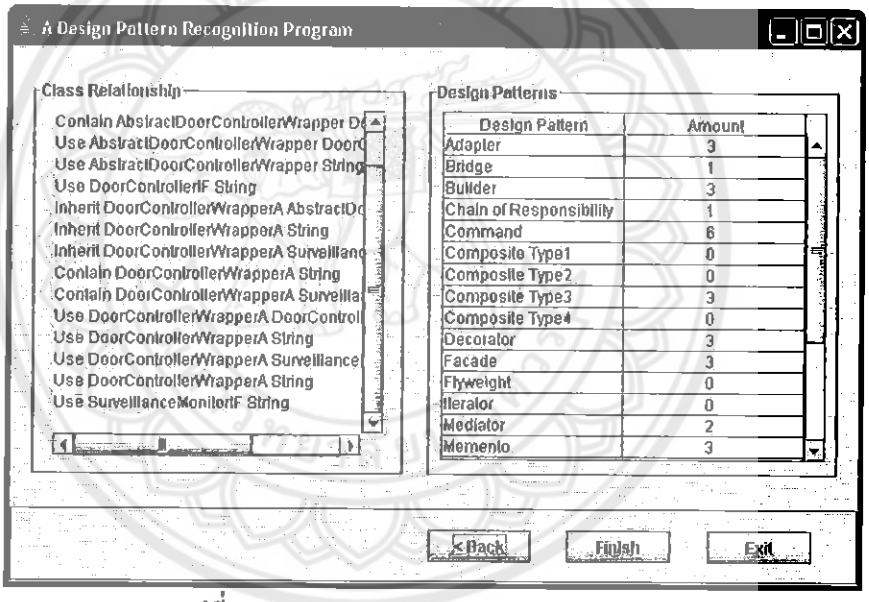
รูปที่ 4.63 รายชื่อไฟล์ของ Composite Pattern



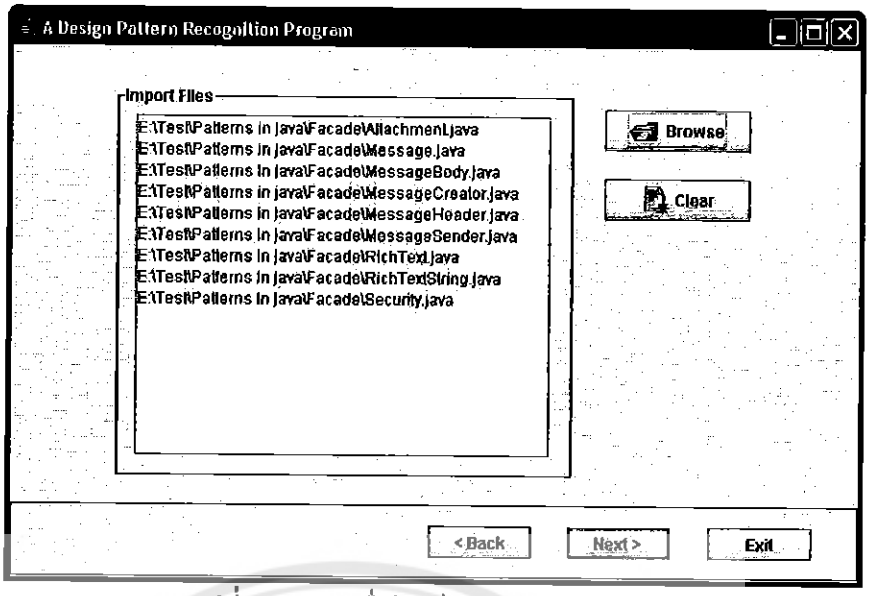
รูปที่ 4.64 ผลการทดสอบ Composite Pattern



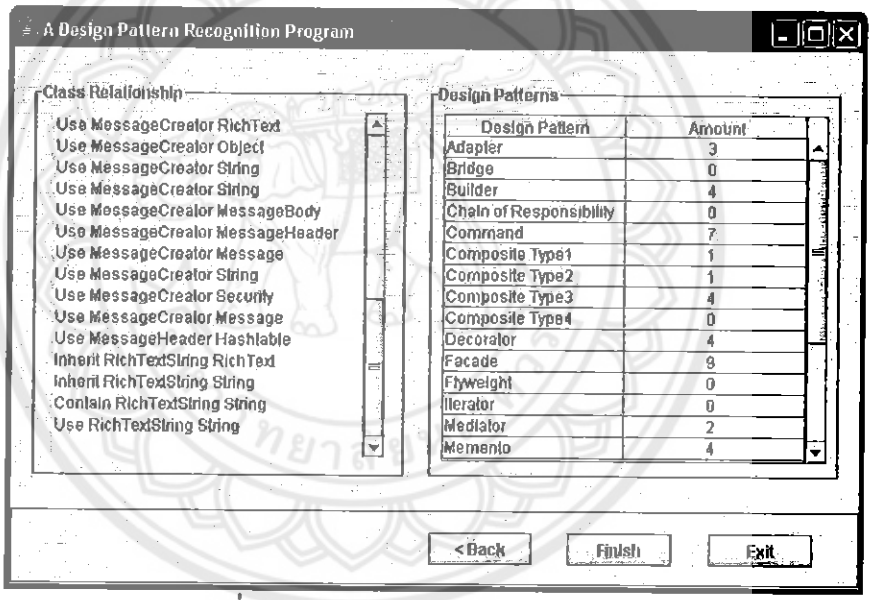
รูปที่ 4.65 รายชื่อไฟล์ของ Decorator Pattern



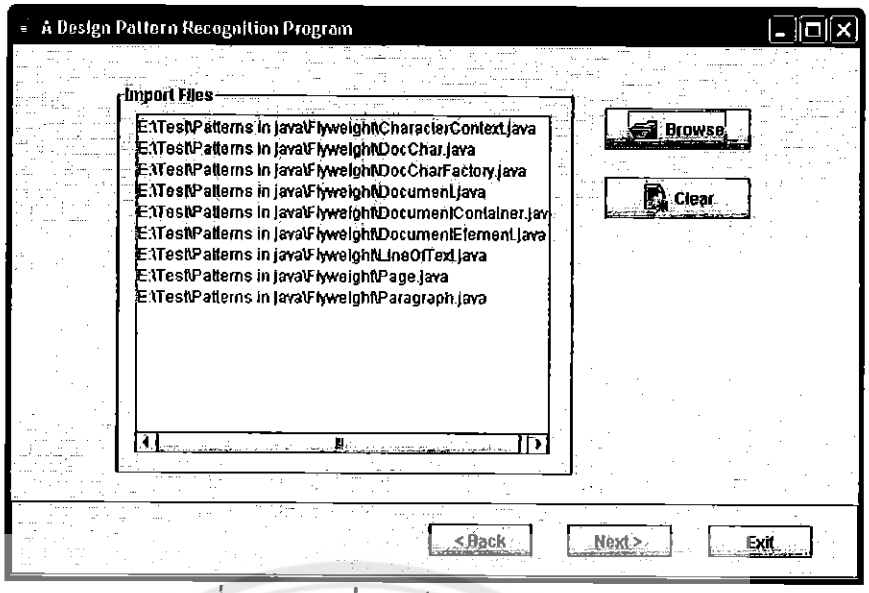
รูปที่ 4.66 ผลการทดสอบ Decorator Pattern



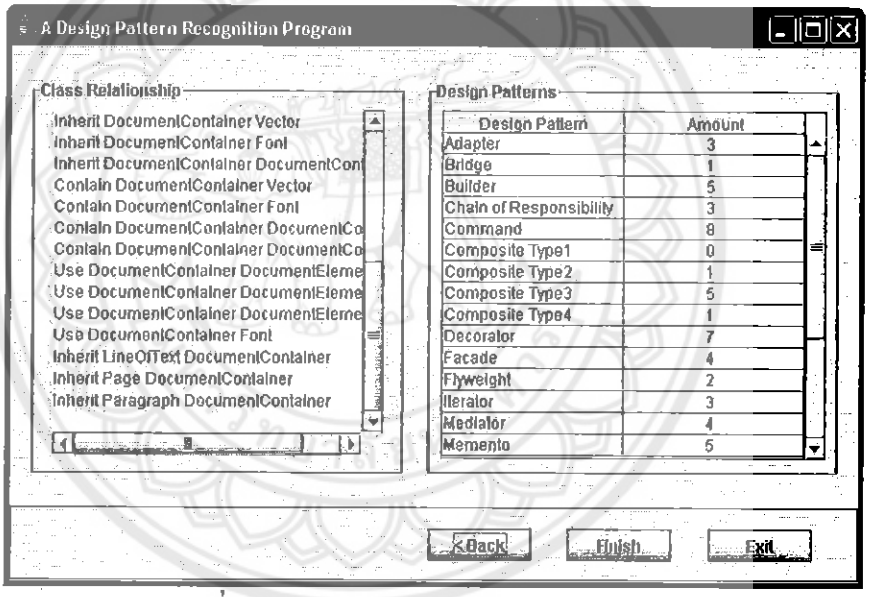
รูปที่ 4.67 รายชื่อไฟล์ของ Façade Pattern



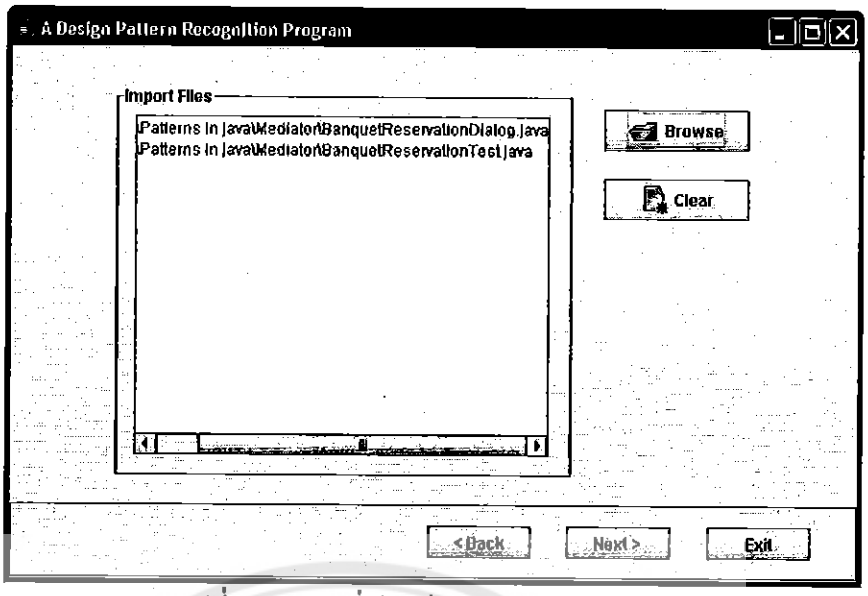
รูปที่ 4.68 ผลการทดสอบ Façade Pattern



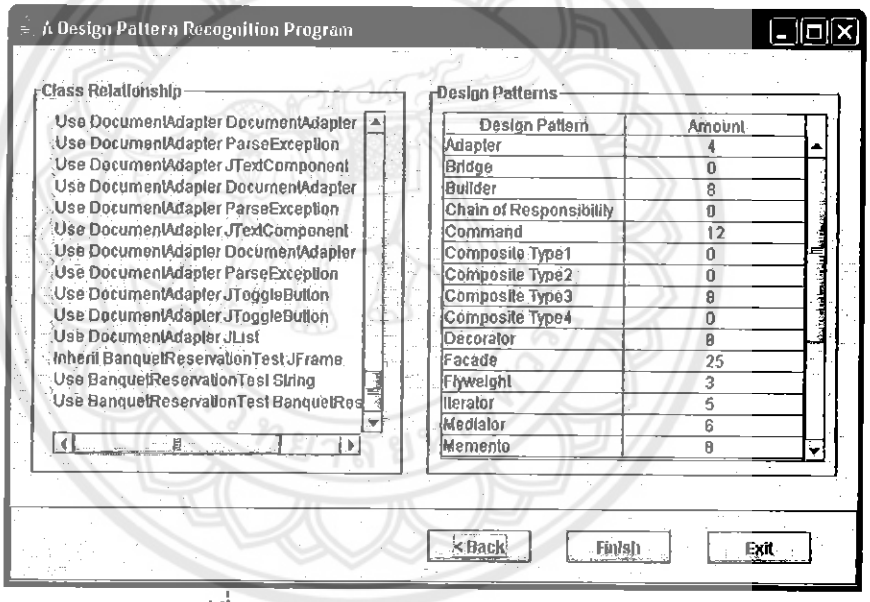
รูปที่ 4.69 รายชื่อไฟล์ของ Flyweight Pattern



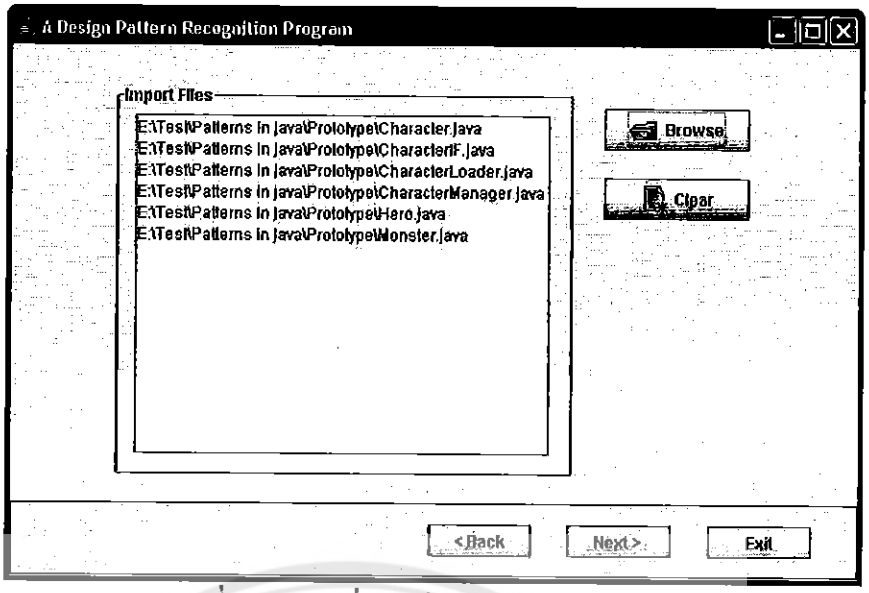
รูปที่ 4.70 ผลการทดสอบ Flyweight Pattern



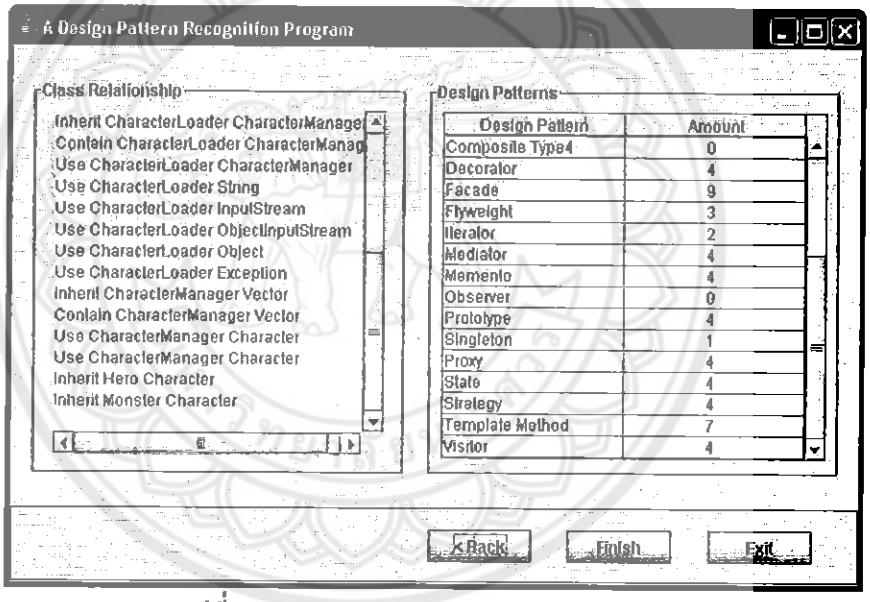
รูปที่ 4.71 รายชื่อไฟล์ของ Mediator Pattern



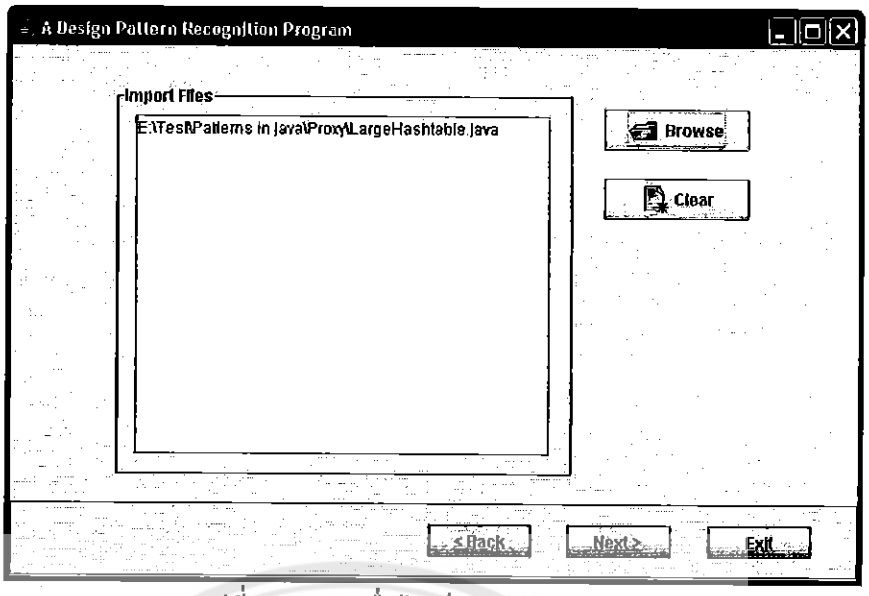
รูปที่ 4.72 ผลการทดสอบ Mediator Pattern



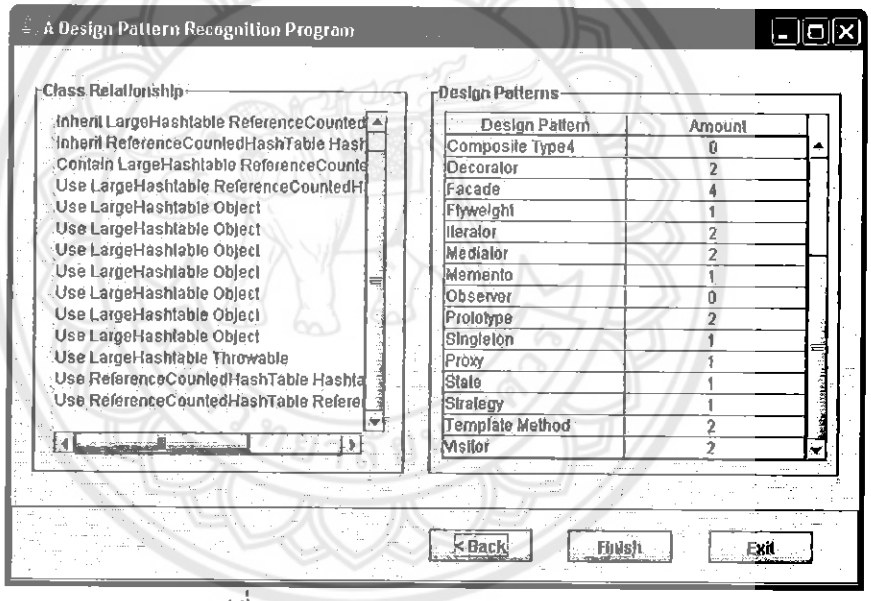
รูปที่ 4.73 รายชื่อไฟล์ของ Prototype Pattern



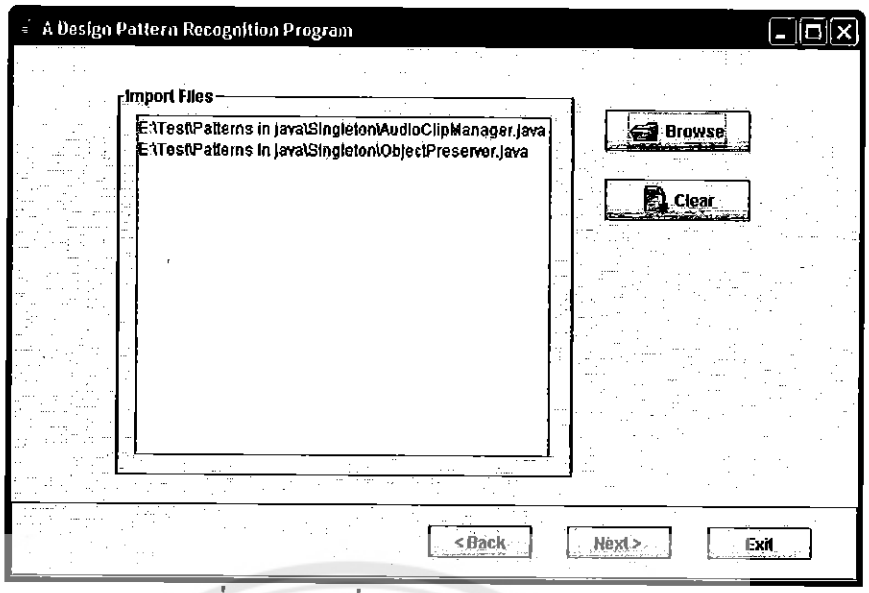
รูปที่ 4.74 ผลการทดสอบ Prototype Pattern



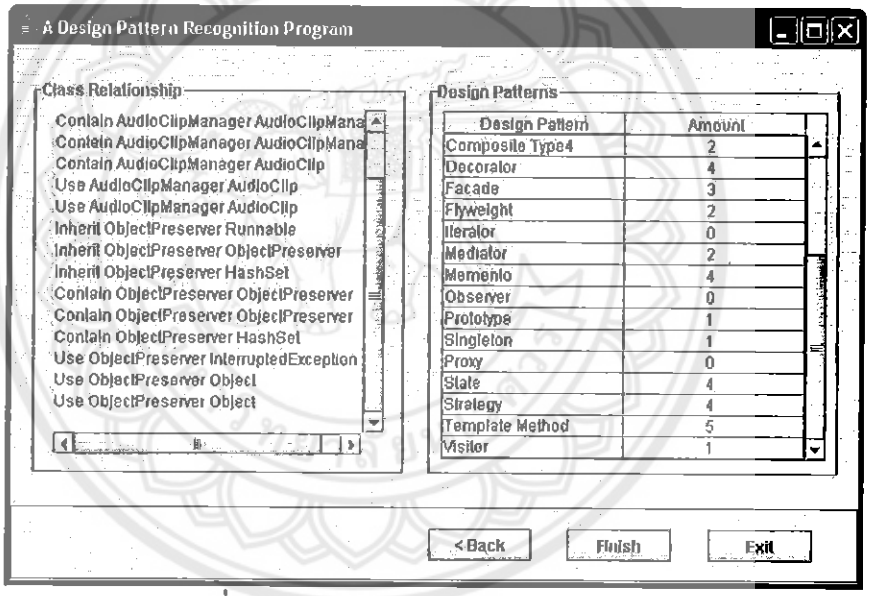
รูปที่ 4.75 รายชื่อไฟล์ของ Proxy Pattern



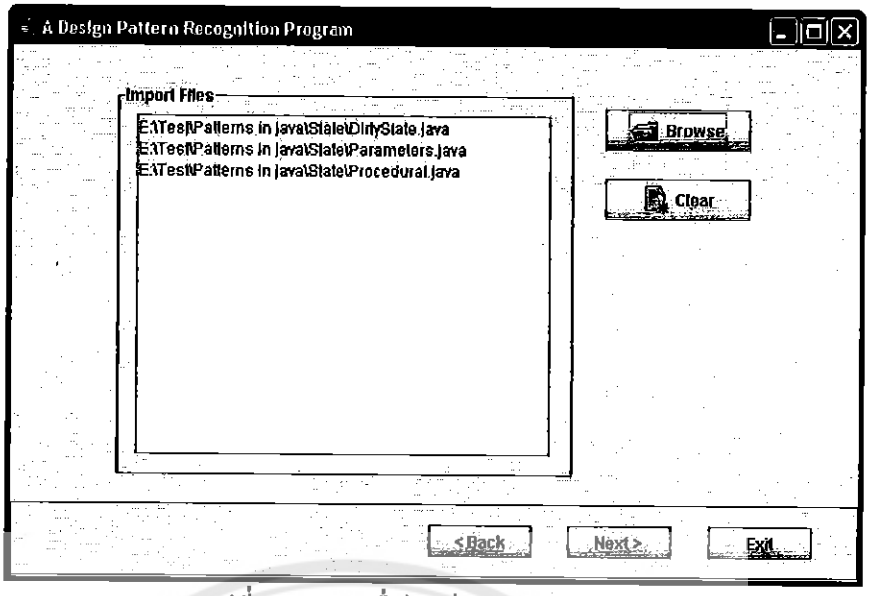
รูปที่ 4.76 ผลการทดสอบ Proxy Pattern



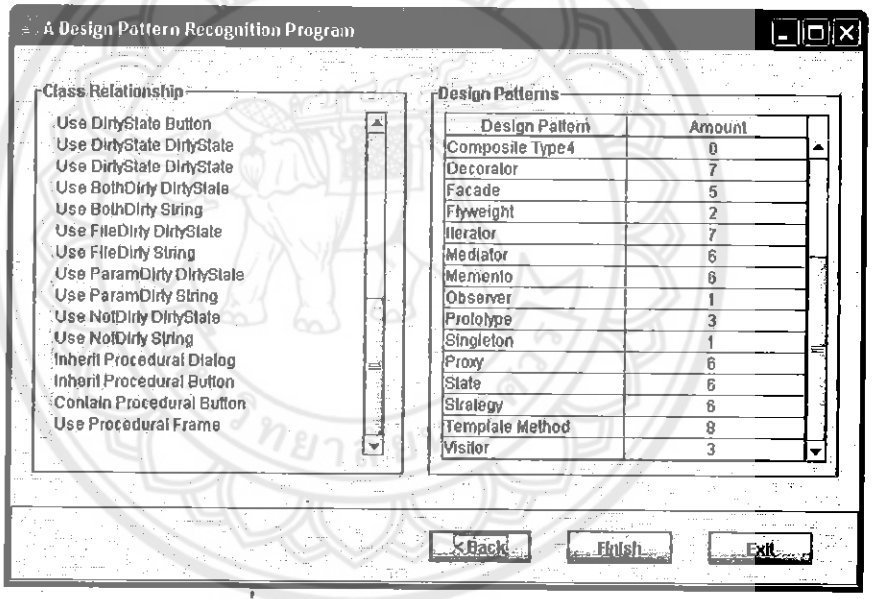
รูปที่ 4.77 รายชื่อไฟล์ของ Singleton Pattern



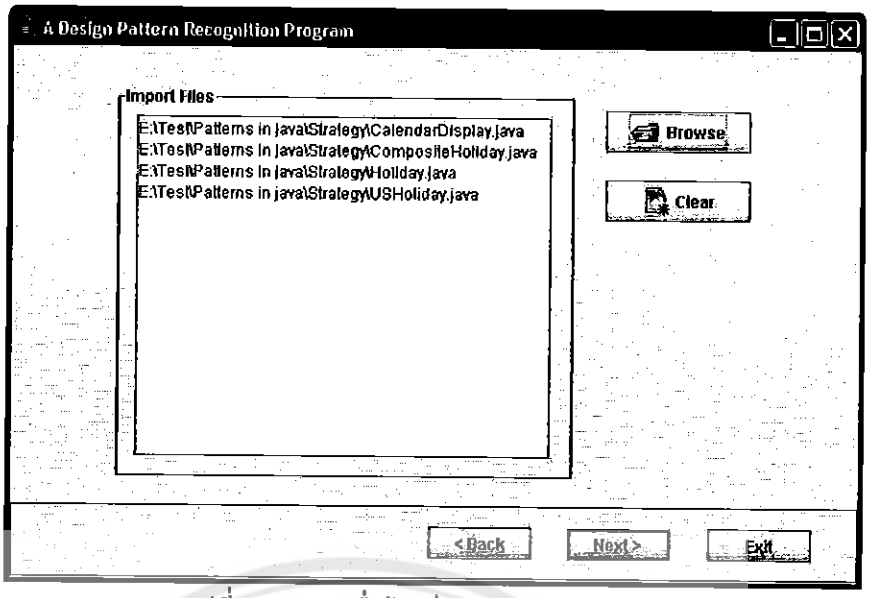
รูปที่ 4.78 ผลการทดสอบ Singleton Pattern



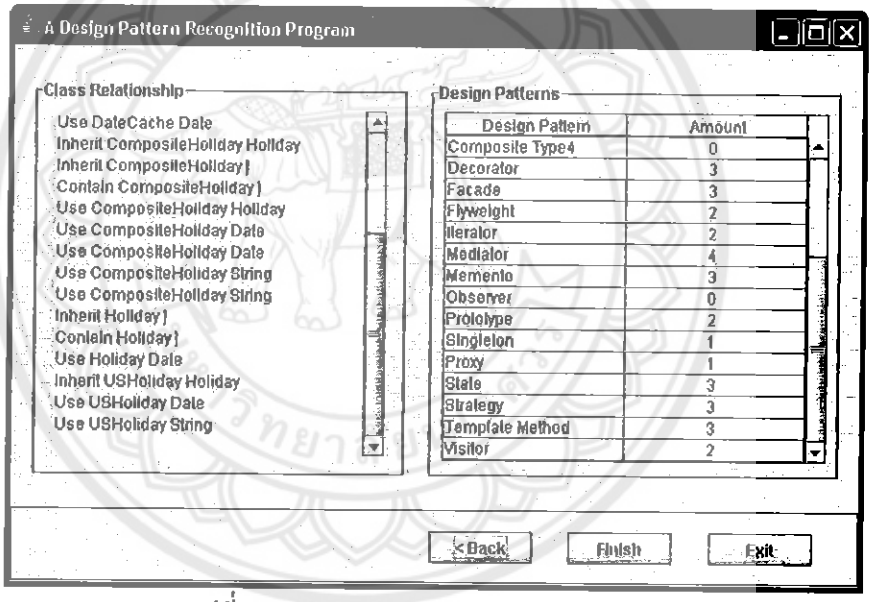
รูปที่ 4.79 รายชื่อไฟล์ของ State Pattern



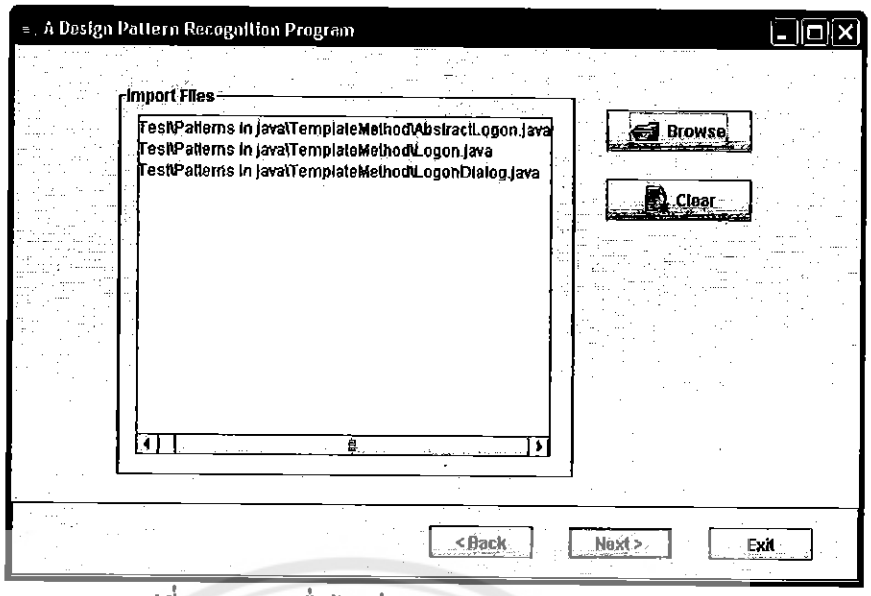
รูปที่ 4.80 ผลการทดสอบ State Pattern



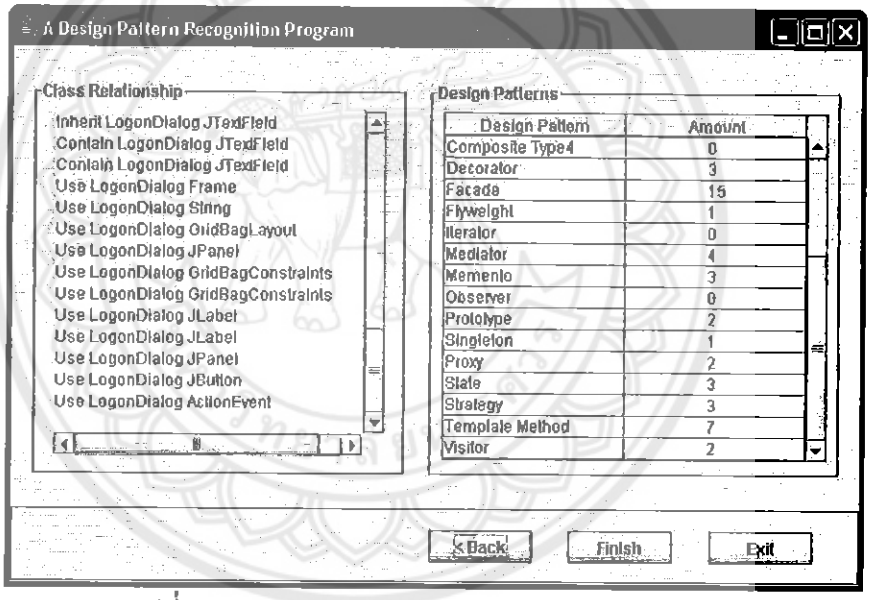
รูปที่ 4.81 รายชื่อไฟล์ของ Strategy Pattern



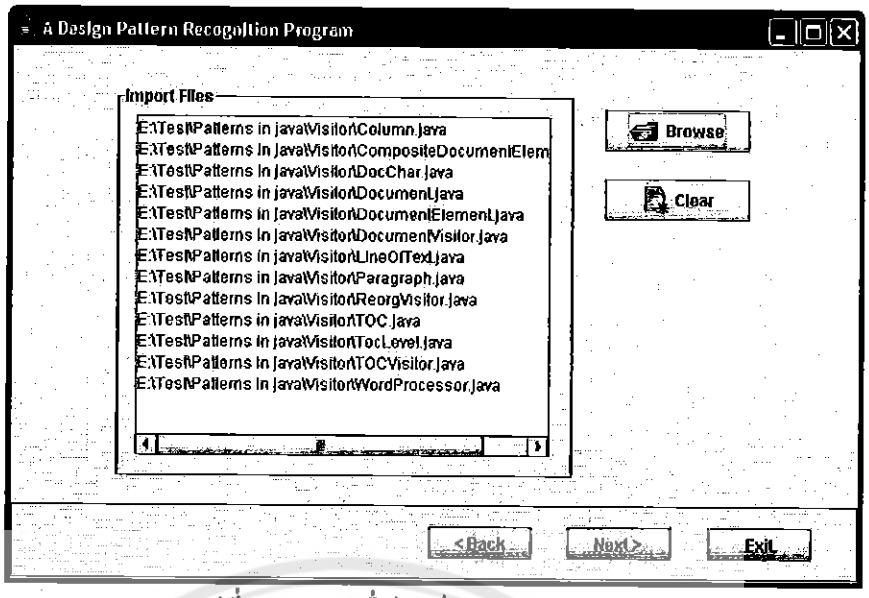
รูปที่ 4.82 ผลการทดสอบ Strategy Pattern



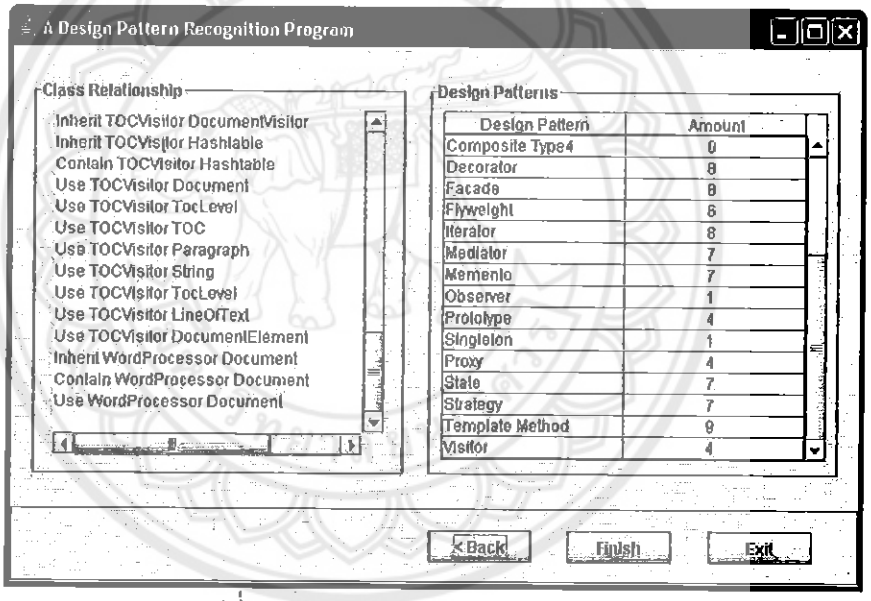
รูปที่ 4.83 รายชื่อไฟล์ของ Templete Method Pattern



รูปที่ 4.84 ผลการทดสอบ Templete Method Pattern



รูปที่ 4.85 รายชื่อไฟล์ของ Visitor Pattern



รูปที่ 4.86 ผลการทดสอบ Visitor Pattern

บทที่ 5
บทสรุป

โครงงานนี้ได้พัฒนาโปรแกรมเพื่อใช้สำหรับตรวจหา Design Pattern จาก source code ภาษาจาวา

5.1 วิเคราะห์ผลการทดลอง

จากผลการทดลองบทที่ 4 ในส่วนของการทดสอบโปรแกรม Design Pattern Recognition โดยใช้ source code จากหนังสือ Design Pattern Java Companion และจากหนังสือ Patterns in Java Volume 1 และใช้ฐานข้อมูลโครงสร้างสำหรับเปรียบเทียบ pattern ของโปรแกรม Crocopat จากภาคผนวก ข. ได้ผลการทดลอง ดังตารางที่ 5.1

ตารางที่ 5.1 ผลการทดสอบโปรแกรมโดยใช้ source code จากหนังสือ Design Pattern Java Companion และจากหนังสือ Patterns in Java Volume 1

Design Patterns	Design Pattern Java Companion	Patterns in Java Volume 1
Adapter	✓	✓
Bridge	✓	✓
Builder	-	✓
Chain of Responsibility	✓	✓
Command	✓	✓
Composite	✓	✓
Decorator	✓	✓
Façade	✓	✓
Flyweight	✓	✓
Iterator	✓	-
Mediator	✓	✓
Memento	✓	-
Prototype	✓	✓
Singleton	✓	✓

Design Patterns	Design Pattern Java Companion	Patterns in Java Volume 1
State	✓	✓
Strategy	✓	✓
Template Method	✓	✓
Visitor	✓	✓
รวม	**17**	**16**
ค่าเฉลี่ย	**1**	**1**

จากตารางที่ 5.1 เครื่องหมายถูก แสดงว่า โปรแกรมสามารถตรวจหา Design Pattern ที่กำหนดได้ และเครื่องหมายยัติภังค์ แสดงว่า หนังสือเล่มนั้นไม่มี source code ใน pattern ที่ต้องการตรวจสอบ ซึ่งจากตารางที่ 5.1 พบว่า โปรแกรม Design Pattern Recognition สามารถตรวจหา Design Pattern จาก source code ของหนังสือทั้งสองเล่ม ได้ทุก pattern ที่มีในฐานข้อมูลในภาคผนวก ข.

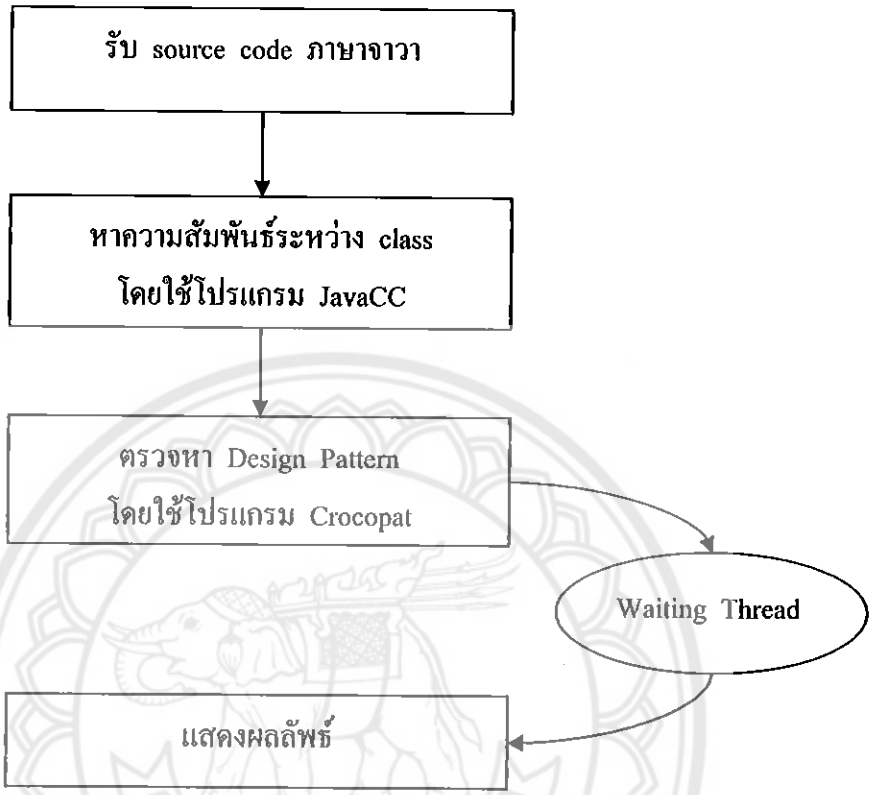
5.2 ปัญหาและแนวทางแก้ไข

ในระหว่างการทำโครงงานนี้ได้ประสบปัญหาทางด้านการเขียนโปรแกรมเป็นส่วนใหญ่ ได้แก่

1. เมื่อสั่งให้โปรแกรม Design Pattern Recognition ควบคุมการทำงานโปรแกรม Crocopat ผลปรากฏว่า โปรแกรม Design Pattern Recognition ทำงานตามคำสั่งต่อไป โดยไม่มีการหยุดรอให้โปรแกรม Crocopat ทำงานที่ได้รับมอบหมายเสร็จสิ้นก่อน ส่งผลให้โปรแกรม Design Pattern Recognition มีการแสดงผลผิดพลาด โดยวิธีการแก้ปัญหานี้คือ ให้โปรแกรม Design Pattern Recognition มี Waiting Thread เพื่อให้หยุดรอการทำงานของโปรแกรม Crocopat ดังรูปที่ 5.1

2. ปัญหาเกี่ยวกับภาษา RML ที่ใช้เขียนแทนโครงสร้างของ Design Pattern ของแต่ละรูปแบบ ซึ่งได้แสดงปัญหาและวิธีการแก้ไขของ Composite Pattern ไว้ในบทที่ 4 แต่ในความเป็นจริงแล้ว ไม่ใช่แค่ Composite Pattern เท่านั้นที่พบปัญหาเหล่านี้ Design Pattern บางรูปแบบน่าจะพบปัญหาเหล่านี้เช่นกัน เนื่องจาก Design Pattern ก็เปรียบเสมือนเป็น Algorithm หนึ่งที่ช่วยในการแก้ปัญหา ซึ่งการนำ Algorithm มาช่วยในการแก้ปัญหาผู้พัฒนาโปรแกรมสามารถหาวิธีการเขียนโปรแกรมเพื่อประยุกต์ใช้ Algorithm ดังกล่าวได้หลายวิธีการ ดังนั้น ข้อผิดพลาดที่เกิดจากการตรวจหา Design Pattern ไม่พบ มีสาเหตุมาจาก

ฐานข้อมูลโครงสร้างของ Design Pattern ในรูปแบบของภาษา RML มีไม่เพียงพอ ทำให้โปรแกรม Crocopat ไม่สามารถตรวจหา Design Pattern ในรูปแบบที่ต้องการได้



รูปที่ 5.1 ขั้นตอนการทำงานของโปรแกรม เมื่อเพิ่ม waiting thread

5.3 สรุปผลการทดลอง

Design Pattern เป็นรูปแบบที่ช่วยให้การออกแบบและพัฒนาโปรแกรมมีประสิทธิภาพเพิ่มมากขึ้น ซึ่งถ้าผู้พัฒนาโปรแกรมต้องการตรวจสอบ Design Pattern จากโปรแกรมที่สร้างขึ้นว่าประกอบด้วย pattern ใดบ้าง อาจจะใช้โปรแกรม Design Pattern Recognition ช่วยตรวจหา pattern ได้ ซึ่งโปรแกรมนี้มีขั้นตอนการทำงานที่สำคัญ 2 ขั้นตอน คือ ขั้นตอนวิเคราะห์เพื่อสร้างความสัมพันธ์ในรูปแบบของ RSF และขั้นตอนการตรวจหา pattern ด้วยโปรแกรม Crocopat ซึ่งจากผลการทดลองแสดงให้เห็นว่า โปรแกรมนี้สามารถตรวจหาพร้อมทั้งนับจำนวนของ Design Pattern ได้ครบทุก pattern ที่มีอยู่ในฐานข้อมูล

5.4 ข้อเสนอแนะ

1. ในขั้นตอนการวิเคราะห์ source code เพื่อสร้างความสัมพันธ์ในรูปแบบของ RSF อาจจะใช้รูปแบบความสัมพันธ์รูปแบบอื่นในการพัฒนาได้ เช่น Abstract syntax tree เป็นต้น

2. สามารถเลือกหรือเปลี่ยนโปรแกรม Crocopat ที่นำมาใช้เปรียบเทียบความสัมพันธ์เป็นโปรแกรมอื่นได้ โดยไม่กระทบการทำงานทั้งหมดของโปรแกรม

เอกสารอ้างอิง

[1] วีระศักดิ์ ซึงถาวร. **Java Programming Volume I.** กรุงเทพมหานคร : ซีเอ็ดยูเคชั่น. 2546.

[2] สิทธิโชค เชาวกุล. **เอกสารประกอบการเรียนการสอนวิชา Compiler Constructor.** 2547

[3] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne. **Operating System Concepts.** 6th Edition. New York: John Wiley and Sons Inc. 2003.

[4] Dirk Beyer and Andreas Noack. **"Crocopat 2.1 Introduction and Reference Manual".** [Online]. Available: http://mtc.epfl.ch/~beyer/CrocoPat/. 2004.

[5] Dirk Beyer, Andreas Noack and Claus Lewerentz. **"Efficient Relational Calculation for Software Analysis".** IEEE Trans. On Software Engineering. Vol. 31. No. 2. Febuary 2005. pp. 137 – 149.

[6] Theodore S Norvell. **"The JavaCC FAQ".** [Online]. Available: http://www.engr.mun.ca/~theo/JavaCC-FAQ/. 2006.

[7] James W. Cooper. **Design Pattern Java Companion.** Addison-Wesley. 1998.

[8] Mark Grand. **Patterns in Java Volume 1.** 2nd Edition. United States of America: Wiley Publishing Inc. 2002.

[9] Michael C. Daconta. **"When Runtime.exe() won't".** [Online]. Available: http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html. 2000.

ภาคผนวก ก.
ตัวอย่างโปรแกรม

ไฟล์ ListOfStudentName.java

```
/**
 * Data Definition
 * ---------------
 * A ListOfStudentName is either
 *   - empty
 * or a structure consisting of
 *   - name as String
 *   - rest as ListOfStudentName
 */
abstract public class ListOfStudentName
{
 /* Code Pattern
  * ... M(...)
  * {
  *     ...
  * }
  */

 /**
  * Count the number of occurrences of a particular name in the list
  * @param nameToCount the name to be counted
  * @return the number of occurrences of a particular name in the list
  * @contract count : String -> int
  */
 abstract public int count(String nameToCount);

 /**
```

```
    * Remove all occurrences of a particular name in the list
    * @param nameToRemove the name to be removed
    * @return the new list with all occurrences of nameToRemove removed
    * @contract remove : String -> ListOfStudentName
    */
   abstract public ListOfStudentName remove(String nameToRemove);

   /**
    * Convert to the representative string
    * @return the representative string
    * @contract toString : -> String
    */
   public String toString()
   {
    return "ListOfStudentName";
   }
}
```

ไฟล์ NodeListOfStudentName.java

```
/**
 * Data Definition
 * ---------------
 * A ListOfStudentName is either
 *   - empty
 * or a structure consisting of
 *   - name as String
 *   - rest as ListOfStudentName
 */
public class NodeListOfStudentName extends ListOfStudentName
{
 private String name;
 private ListOfStudentName rest;
```

```
/* Code Pattern
 * ... M(...)
 * {
 *    ... this.name ...
 *    ... this.rest ...
 * }
 *
 * ... M(...)
 * {
 *    ... this.name ...
 *    ... this.rest.M(...) ...
 * }
 */

/**
 * Constructor
 * @param n the initial name
 * @param r the initial rest
 * @contract NodeListOfStudentName : String ListOfStudentName ->
 * @example NodeListOfStudentName("Tony",
 *              new EmptyListOfStudentName())
 * @example NodeListOfStudentName("Mike",
 *              new NodeListOfStudentName("Tony",
 *              new EmptyListOfStudentName()))
 * @example NodeListOfStudentName("Paul",
 *              new NodeListOfStudentName("Mike",
 *              new NodeListOfStudentName("Tony",
 *              new EmptyListOfStudentName())))
 */
public NodeListOfStudentName(String n, ListOfStudentName r)
{
```

```
  this.name = n;
  this.rest = r;
 }

 /**
  * Count the number of occurrences of a particular name in the list
  * @param nameToCount the name to be counted
  * @return the number of occurrences of a particular name in the list
  * @contract count : String -> int
  * @example NodeListOfStudentName("Tony",
  *                new EmptyListOfStudentName()).count("Tony") == 1
  * @example NodeListOfStudentName("Mike",
  *                new NodeListOfStudentName("Tony",
  *                new EmptyListOfStudentName())).count("Tony") == 1
  * @example NodeListOfStudentName("Paul",
  *                new NodeListOfStudentName("Mike",
  *                new NodeListOfStudentName("Tony",
  *                new EmptyListOfStudentName()))).count("Mike") == 1
  */
 public int count(String nameToCount)
 {
  if (this.name.equals(nameToCount))
  {
   return 1 + this.rest.count(nameToCount);
  }
  else
  {
   return 0 + this.rest.count(nameToCount);
  }
 }

 /**
```

```
 * Remove all occurrences of a particular name in the list
 * @param nameToRemove the name to be removed
 * @return the new list with all occurrences of nameToRemove removed
 * @contract remove : String -> ListOfStudentName
 * @example NodeListOfStudentName("Tony",
 *              new EmptyListOfStudentName()).remove("Tom")
 *  == new NodeListOfStudentName("Tony", new EmptyListOfStudentName())
 * @example NodeListOfStudentName("Mike",
 *              new NodeListOfStudentName("Tony",
 *              new EmptyListOfStudentName())).remove("Tony")
 *  == new NodeListOfStudentName("Mike", new EmptyListOfStudentName())
 * @example NodeListOfStudentName("Paul",
 *              new NodeListOfStudentName("Mike",
 *              new NodeListOfStudentName("Tony",
 *              new EmptyListOfStudentName()))).remove("Mike")
 *  == new NodeListOfStudentName("Paul",
 *              new NodeListOfStudentName("Tony",
 *              new EmptyListOfStudentName()))
 */
public ListOfStudentName remove(String nameToRemove)
{
 if (this.name.equals(nameToRemove))
 {
  return this.rest.remove(nameToRemove);
 }
 else
 {
  return new NodeListOfStudentName(this.name, this.rest.remove(nameToRemove));
 }
}

/**
```

```
   * Convert to the representative string
   * @return the representative string
   * @contract toString : -> String
   * @example NodeListOfStudentName("Tony",
   *              new EmptyListOfStudentName()).toString() == (Tony, empty)
   * @example NodeListOfStudentName("Mike",
   *              new NodeListOfStudentName("Tony",
   *              new EmptyListOfStudentName())).toString() == (Mike, (Tony, empty))
   * @example NodeListOfStudentName("Paul",
   *              new NodeListOfStudentName("Mike",
   *              new NodeListOfStudentName("Tony",
   *              new EmptyListOfStudentName()))).toString()
   *                                          == (Paul, (Mike, (Tony, empty)))
   */
  public String toString()
  {
   return "(" + this.name + ", " + this.rest.toString() + ")";
  }
}
```

ไฟล์ Employee.java

```
import java.util.*;

public class Employee
{
  String name;
  float salary;
  Vector subordinates;
  boolean isLeaf;
  Employee parent = null;
//--------------------------------------
  public Employee(String _name, float _salary)
```

```
{
  name = _name;
  salary = _salary;
  subordinates = new Vector();
  isLeaf = false;
}
//---------------------------------------
public Employee(Employee _parent, String _name, float _salary)
{
  name = _name;
  salary = _salary;
  parent = _parent;
  subordinates = new Vector();
  isLeaf = false;
}
//---------------------------------------
public void setLeaf(boolean b)
{
  isLeaf = b;   //if true, do not allow children
}
//---------------------------------------
public float getSalary()
{
  return salary;
}
//---------------------------------------
public String getName()
{
  return name;
}
//---------------------------------------
public boolean add(Employee e)
```

```
{
  if (! isLeaf)
    subordinates.addElement(e);
  return isLeaf;   //false if unsuccessful
}
//-------------------------------------
public void remove(Employee e)
{
  if (! isLeaf)
    subordinates.removeElement(e);
}
//-------------------------------------
public Enumeration elements()
{
  return subordinates.elements();
}
//-------------------------------------
public Employee getChild(String s)
 {
   Employee newEmp = null;

   if(getName().equals(s))
     return this;
   else
   {
     boolean found = false;
     Enumeration e = elements();
     while(e.hasMoreElements() && (! found))
     {
       newEmp = (Employee)e.nextElement();
       found = newEmp.getName().equals(s);
       if (! found)
```

```
                {
                newEmp = newEmp.getChild(s);
                found =(newEmp != null);
                }
            }
            if (found)
              return newEmp;
            else
              return null;
          }
        }

        //-----------------------------------
        public float getSalaries()
        {
          float sum = salary;
          for(int i = 0; i < subordinates.size(); i++)
          {
            sum += ((Employee)subordinates.elementAt(i)).getSalaries();
          }
          return sum;
        }
      }
```

ไฟล์ IntList.java

```
/**
 * A representation of a list
 * @author Suradet Jitprapaikulsarn
 * @date 2005 December 15
 * @note JDK 1.5.0_06 (Windows XP SP2)
 */
```

```
/**
 * Data Definition
 * ---------------
 * An IntList is either
 *   - an empty list
 * or a structure consisting of
 *   - data as int
 *   - rest as IntList
 */
abstract class IntList
{
 /* Code Pattern
  * ... M(...)
  * {
  *    ...
  * }
  */

 /**
  * Convert to a representative string
  * @return a representative string
  * @contract toString : -> String
  */
 abstract public String toString();

 /**
  * A helper methor for converting to a representative string
  * @return a representative string
  * @contract toStringHelper : -> String
  */
 abstract String toStringHelper();
```

```
 /**
  * Change the sign of every element in the list
  * @return a new list whose elements have the opposite sign to the current list
  * @contract flip : -> IntList
  */
 abstract public IntList flip();
}
```

ไฟล์ **NodeIneList.java**

```
/**
 * A representation of a node in a list
 * @author Suradet Jitprapaikulsarn
 * @date 2005 December 15
 * @note JDK 1.5.0_06 (Windows XP SP2)
 */

/**
 * Data Definition
 * ---------------
 * An IntList is either
 *   - an empty list
 * or a structure consisting of
 *   - data as int
 *   - rest as IntList
 */
public class NodeIntList extends IntList
{
 private int data;
 private IntList rest;

 /* Code Pattern
  * ... M(...)
```

```
 * {
 *    ... this.data ...
 *    ... this.rest ...
 * }
 *
 * ... M(...)
 * {
 *    ... this.data ...
 *    ... this.rest.M(...) ...
 * }
 */

/**
 * Constructor
 * @param d the initial data
 * @param r the initial rest
 * @contract NodeIntList : int IntList ->
 * @example NodeIntList(2, new EmptyIntList())
 * @example NodeIntList(3, new NodeIntList(2, new EmptyIntList()))
 *    @example NodeIntList(5, new NodeIntList(3, new NodeIntList(2, new
EmptyIntList())))
 */
public NodeIntList(int d, IntList r)
{
 this.data = d;
 this.rest = r;
}

/**
 * Convert to a representative string
 * @return a representative string
 * @contract toString : -> String
```

```
  * @example NodeIntList(2, new EmptyIntList()).toString() == "[2]"
  * @example NodeIntList(3, new NodeIntList(2, new EmptyIntList())).toString() ==
"[3, 2]"
  * @example NodeIntList(5, new NodeIntList(3, new NodeIntList(2, new
EmptyIntList()))).toString() == "[5, 3, 2]"
  */
 public String toString()
 {
  return "[" + this.data + this.rest.toStringHelper() + "]";
 }

 /**
  * A helper method for converting to a representative string
  * @return a representative string
  * @contract toStringHelper : -> String
  * @example NodeIntList(2, new EmptyIntList()).toStringHelper() == ", 2"
  * @example NodeIntList(3, new NodeIntList(2, new EmptyIntList())).toStringHelper()
== ", 3, 2"
  * @example NodeIntList(5, new NodeIntList(3, new NodeIntList(2, new
EmptyIntList()))).toStringHelper() == ", 5, 3, 2"
  */
 String toStringHelper()
 {
  return ", " + this.data + this.rest.toStringHelper();
 }

 /**
  * Change the sign of every element in the list
  * @return a new list whose elements have the opposite sign to the current list
  * @contract flip : -> IntList
  * @example
```

```
  * @example NodeIntList(2, new EmptyIntList()).flip() == NodeIntList(-2, new
EmptyIntList())
  * @example NodeIntList(3, new NodeIntList(2, new EmptyIntList())).flip()
  *       == NodeIntList(-3, new NodeIntList(-2, new EmptyIntList()))
  * @example NodeIntList(-5, new NodeIntList(3, new NodeIntList(2, new
EmptyIntList()))).flip()
  *                == NodeIntList(5, new NodeIntList(-3, new NodeIntList(-2, new
EmptyIntList())))
  */
 public IntList flip()
 {
  return new NodeIntList(-this.data, this.rest.flip());
 }
}
```

ไฟล์ EmptyList.java

```
/**
 * A representation of an empty list
 * @author Suradet Jitprapaikulsarn
 * @date 2005 December 15
 * @note JDK 1.5.0_06 (Windows XP SP2)
 */

/**
 * Data Definition
 * ---------------
 * An IntList is either
 *   - an empty list
 * or a structure consisting of
 *   - data as int
 *   - rest as IntList
 */
```

```
public class EmptyIntList extends IntList
{
 /* Code Pattern
  * ... M(...)
  * {
  *    ...
  * }
  */

 /**
  * Convert to a representative string
  * @return a representative string
  * @contract toString : -> String
  * @example EmptyIntList().toString() == "[]"
  */
 public String toString()
 {
  return "[]";
 }

 /**
  * A helper method for converting to a representative string
  * @return a representative string
  * @contract toStringHelper : -> String
  * @example EmptyIntList().toString() == ""
  */
 String toStringHelper()
 {
  return "";
 }

 /**
```

```
 * Change the sign of every element in the list
 * @return a new list whose elements have the opposite sign to the current list
 * @contract flip : -> IntList
 * @example EmptyIntList().flip() == EmptyIntList()
 */
public IntList flip()
{
 return new EmptyIntList();
}
}
```

ไฟล์ IntList2.java

```
abstract class IntList2
{
 abstract public String toString();
}
```

ไฟล์ NodeIneList2.java

```
public class NodeIntList2 extends IntList2
{
 private int data;
 private IntList2 rest;
 private IntList3 prev;

 public NodeIntList2(int d, IntList2 r, IntList3 p)
 {
  this.data = d;
  this.rest = r;
  this.prev = p;
 }

 public String toString()
```

```
 {
  return this.data + this.rest.toString();
 }
}
```

ไฟล์ **EmptyList2.java**

```
public class EmptyIntList2 extends IntList2
{
 public String toString()
 {
  return "[]";
 }
}
```

ไฟล์ **IntList3.java**

```
abstract class IntList3
{
 abstract public String toString();
}
```

ไฟล์ **NodeIneList3.java**

```
public class NodeIntList3 extends IntList3
{
 private int data;
 private IntList3 rest;
 private IntList prev;

 public NodeIntList2(int d, IntList2 r, IntList p)
 {
  this.data = d;
  this.rest = r;
  this.prev = p;
```

```
  }
  public String toString()
  {
   return this.data + this.rest.toString();
  }
 }
```

ไฟล์ **EmptyList3.java**

```
public class EmptyIntList3 extends IntList3
{
 public String toString()
 {
  return "[]";
 }
}
```

ไฟล์ **BinaryTree.java**

```
package binaryTree;
/**
 * Data Definition
 * ---------------
 * A BinaryTree is either
 *   - Empty
 * or a structure consisting of
 *   - data as int
 *   - left as BinaryTree
 *   - right as BinaryTree
 */
abstract public class BinaryTree
{
 /* Code Pattern
  * ...M(...)
```

```
     * {
     *   ...
     * }
     */

    /**
     * Find the extreme value from the tree
     * @param C the comparison operator
     * @return the extreme value from the tree
     * @contract extremeValue : IComparator -> int
     */
    abstract public int extremeValue(Comparator c);

    /**
     * Find the extreme value from the tree
     * @param v the extreme value so far
     * @return the extreme value from the tree
     * @contract extremeValueHelper : int IComparator -> int
     */
    abstract public int extremeValueHelper(int v, Comparator c);
}
```

ไฟล์ NodeBinaryTree.java

```
/**
 * Data Definition
 * ---------------
 * A BinaryTree is either
 *   - Empty
 * or a structure consisting of
 *   - data as int
 *   - left as BinaryTree
 *   - right as BinaryTree
```

```
*/
public class NodeBinaryTree extends BinaryTree
{
 private int data;
 private BinaryTree left;
 private BinaryTree right;

 /* Code Pattern
  * ...M(...)
  * {
  *    ... this.data ...
  *    ... this.left ...
  *    ... this.right ...
  * }
  *
  * ...M(...)
  * {
  *    ... this.data ...
  *    ... this.left.M(...) ...
  *    ... this.right.M(...) ...
  * }
  */

 /**
  * Constructor
  * @param d the initial data
  * @param L the initial left branch
  * @param R the initial right branch
  * @contract NodeBinaryTree : int BinaryTree BinaryTree ->
  * @example NodeBinaryTree(2, new EmptyBinaryTree(), new EmptyBinaryTree())
  * @example NodeBinaryTree(3, new EmptyBinaryTree(), new EmptyBinaryTree())
  * @example NodeBinaryTree(1,
```

```
     *       new NodeBinaryTree(2, new EmptyBinaryTree(), new EmptyBinaryTree()),
     *      new NodeBinaryTree(3, new EmptyBinaryTree(), new EmptyBinaryTree()))
     */
    public NodeBinaryTree(int d, BinaryTree L, BinaryTree R)
    {
     this.data = d;
     this.left = L;
     this.right = R;
    }

    /**
     * Convert to a string representation
     * @return the string representation
     * @contract toString : -> String
     * @example NodeBinaryTree(2, new EmptyBinaryTree(), new EmptyBinaryTree())
    == (2, empty, empty)
     * @example NodeBinaryTree(3, new EmptyBinaryTree(), new EmptyBinaryTree())
    == (3, empty, empty)
     * @example NodeBinaryTree(1,
     *       new NodeBinaryTree(2, new EmptyBinaryTree(), new EmptyBinaryTree()),
     *       new NodeBinaryTree(3, new EmptyBinaryTree(), new EmptyBinaryTree()))
     * == (1, (2, empty, empty), (3, empty, empty))
     */
    public String toString()
    {
     return "(" + this.data + ", " + this.left.toString() + ", " + this.right.toString() + ")";
    }

    /**
     * Find the extreme value from the tree
     * @param C the comparison operator
     * @return the extreme value from the tree
```

```
 * @contract extremeValue : IComparator -> int
 * @example NodeBinaryTree(2,
 *                  new EmptyBinaryTree(),
 *                  new EmptyBinaryTree()).extremeValue(new GreaterThan()) == 2
 * @example NodeBinaryTree(3,
 *                  new EmptyBinaryTree(),
 *                  new EmptyBinaryTree()).extremeValue(new LessThan()) == 3
 * @example NodeBinaryTree(1,
 *                  new NodeBinaryTree(2,
 *                  new EmptyBinaryTree(),
 *                  new EmptyBinaryTree()),
 *                  new NodeBinaryTree(3,
 *                  new EmptyBinaryTree(),
 *                  new EmptyBinaryTree())).extremeValue(new LessThan()) == 1
 */
public int extremeValue(Comparator c)
{
 return c.extremeValue(this.left.extremeValueHelper(this.data, c),
                this.right.extremeValueHelper(this.data, c));
}

/**
 * Find the extreme value from the tree
 * @param v the extreme value so far
 * @return the extreme value from the tree
 * @contract extremeValueHelper : int IComparator -> int
 * @example NodeBinaryTree(2,
 *                  new EmptyBinaryTree(),
 *                   new EmptyBinaryTree()).extremeValueHelper(3, new GreaterThan())
== 3
 * @example NodeBinaryTree(3,
 *                  new EmptyBinaryTree(),
```

```
     *                  new EmptyBinaryTree()).extremeValueHelper(4, new LessThan()) ==
  3
     * @example NodeBinaryTree(1,
     *                  new NodeBinaryTree(2,
     *                          new EmptyBinaryTree(),
     *                          new EmptyBinaryTree()),
     *                  new NodeBinaryTree(3,
     *                          new EmptyBinaryTree(),
     *                              new EmptyBinaryTree())).extremeValueHelper(4, new
  LessThan()) == 1
     */
    public int extremeValueHelper(int v, Comparator c)
    {
      return c.extremeValue( this.left.extremeValueHelper(c.extremeValue(v, this.data), c),
                      this.right.extremeValueHelper(c.extremeValue(v, this.data), c) );
    }
  }
```

ไฟล์ EmptyBinaryTree.java

```
  package binaryTree;
  /**
   * Data Definition
   * ---------------
   * A BinaryTree is either
   *   - Empty
   * or a structure consisting of
   *   - data as int
   *   - left as BinaryTree
   *   - right as BinaryTree
   */
  public class EmptyBinaryTree extends BinaryTree
  {
```

```
/* Code Pattern
 * ...M(...)
 * {
 *   ...
 * }
 */

/**
 * Convert to a string representation
 * @return the string representation
 * @contract toString : -> String
 * @example EmptyBinaryTree.toString() == "empty"
 */
public String toString()
{
 return "empty";
}

/**
 * Find the extreme value from the tree
 * @param C the comparison operator
 * @return the extreme value from the tree
 * @contract extremeValue : IComparator -> int
 * @example EmptyBinaryTree().extremeValue(new GreaterThan()) => exception
 */
public int extremeValue(Comparator c)
{
 // Exception
 throw new RuntimeException();
}

/**
```

```
    * Find the extreme value from the tree
    * @param v the extreme value so far
    * @return the extreme value from the tree
    * @contract extremeValueHelper : int IComparator -> int
    * @example EmptyBinaryTree().extremeValue(3, new GreaterThan()) == 3
    */
   public int extremeValueHelper(int v, Comparator c)
   {
    return v;
   }
}
```

ไฟล์ BinaryPlus.java

```
/**
 * An expression representing an binary addition expression
 * @author Suradet Jitprapaikulsarn
 * @data 2005 December 15
 * @note JDK 1.5.0_06 (Windows XP SP2)
 */

/**
 * Data Definition
 * ---------------
 * An Expression is either
 *   - a number
 * or a structure consisting of
 *   - an operator '-'
 *   - an operand
 * or a structure consisting of
 *   - an operator '+'
 *   - an operand
 *   - an operand
```

```
* or a structure consisting of
*   - an operator '-'
*   - an operand
*   - an operand
* or a structure consisting of
*   - an operator '*'
*   - an operand
*   - an operand
* or a structure consisting of
*   - an operator '/'
*   - an operand
*   - an operand
*/
public class BinaryPlus implements IExpression
{
 private char op;
 private IExpression left;
 private IExpression right;

 /* Code Pattern
  * ... M(...)
  * {
  *    ... this.op ...
  *    ... this.left ...
  *    ... this.right ...
  * }
  *
  * ... M(...)
  * {
  *    ... this.op ...
  *    ... this.left.M2(...) ...
  *    ... this.right.M2(...) ...
```

```
 * }
 */

/**
 * Constructor
 * @param L the initial left expression
 * @param R the initial right expression
 * @contract BinaryPlus : IExpression IExpression ->
 * @example BinaryPlus(new Number(2.0), new Number(3.0))
 * @example BinaryPlus(new BinaryPlus(new Number(5.0), new Number(8.0)), new
Number(12.0))
 * @example BinaryPlus(new BinaryPlus(new Number(4.0), new Number(7.0)), new
UnaryMinus(new Number(2.0)))
 */
public BinaryPlus(IExpression L, IExpression R)
{
 this.op = '+';
 this.left = L;
 this.right = R;
}

/**
 * Evaluate the value of the expression
 * @return the value of the expression
 * @contract evaluate : -> double
 * @example BinaryPlus(new Number(2.0), new Number(3.0)).evaluate() == 5.0
 * @example BinaryPlus(new BinaryPlus(new Number(5.0), new Number(8.0)), new
Number(12.0)).evaluate() == 25.0
 * @example BinaryPlus(new BinaryPlus(new Number(4.0), new Number(7.0)), new
UnaryMinus(new Number(2.0))).evaluate() == 9.0
 */
public double evaluate()
```

```
{
 return this.left.evaluate() + this.right.evaluate();
}

/**
 * Convert to a representative string
 * @return a representative string
 * @contract toString : -> String
 * @example BinaryPlus(new Number(2.0), new Number(3.0)).toString() == "(2.0+
3.0)"
 * @example BinaryPlus(new BinaryPlus(new Number(5.0), new Number(8.0)), new
Number(12.0)).toString() == "((5.0 + 8.0) + 12.0)"
 * @example BinaryPlus(new BinaryPlus(new Number(4.0), new Number(7.0)), new
UnaryMinus(new Number(2.0))).toString() == "((4.0 + 7.0) + -2.0)"
 */
public String toString()
{
 return "(" + this.left.toString() + " + " + this.right.toString() + ")";
}
```

ภาคผนวก ข.

ภาษา RML ของแต่ละ Design Pattern

ไฟล์ DesignPattern.rml

```
Adapter (Target, Adapter, Adaptee, Client) :=
                        Inherit (Adapter, Target)
                      & Use (Adapter, Adaptee)
                      & Use (Client, Target);
PRINT "Adapter: " , #(Adapter(Target,_,_,_)) , ";" , ENDL;

Bridge (Abstraction, Implementor, RefineAbstraction, ConcreteImplementor) :=
                        Inherit (RefineAbstraction, Abstraction)
                      & Inherit (ConcreteImplementor, Implementor)
                      & Contain (Abstraction, Implementor);
PRINT "Bridge: " , #(Bridge(Abstraction,_,_,_)) , ";" , ENDL;

BuilderP(Builder, Director, ConcreteBuilder) :=
                        Contain (Director, Builder)
                      & Inherit (ConcreteBuilder, Builder);
PRINT "Builder: " , #(BuilderP(Builder,_,_)) , ";" , ENDL;


Chain(CommandHeader, AbstractCommand, Concrete) :=
                        Inherit (AbstractCommand , CommandHeader)
                      & Inherit (Concrete, AbstractCommand)
                      & Use (AbstractCommand, CommandHeader);
PRINT "Chain of Responsibility: " , #(Chain(CommandHeader,_,_)) , ";" , ENDL;

CommandP1(Command, Invoker, ConreteCommand, Receiver, Client) :=
                        Contain (Invoker, Command)
                      & Inherit (ConreteCommand, Command)
```

```
                    & Use (ConreteCommand, Receiver)
                    & Use (Client, Receiver);
nCommand :=  #(CommandP1(Command,_,_,_,_) );

CommandP2(Command, CommandManager, ConreteCommand) :=
                    Inherit (ConreteCommand, Command)
                    & Use (CommandManager, Command);
nCommand2 := #(CommandP2(Command,_,_));
nCommand0 := nCommand + nCommand2;
PRINT "Command: " , nCommand0 , ";" , ENDL;

Composite1 (Component, Leaf, Composite, Client) :=
                      Inherit (Composite, Component)
                    & Contain (Composite, Component)
                    & Inherit (Leaf, Component)
                    & ! Contain (Leaf, Component)
                    & Use (Client, Component);
PRINT "Composite Type1: " , #(Composite1(Component,_,_,_)) , ";" , ENDL;

Composite2 (Component, Leaf, Composite) :=
                      Inherit (Composite, Component)
                    & Contain (Composite, Component)
                    & Inherit (Leaf, Component)
                    & ! Contain (Leaf, Component);
PRINT "Composite Type2: " , #(Composite2(Component,_,_,_)) , ";" , ENDL;

Composite3 (Component, Composite) :=
                      Inherit (Composite, Component)
                    & Contain (Composite, Component);
PRINT "Composite Type3: " , #(Composite3(Component,_,_,_)) , ";" , ENDL;
```

```
Composite4 (Component) :=
                    Contain (Component, Component);
PRINT "Composite Type4: " , #(Composite4(Component,_,_,_)) , ";" , ENDL;

Decorator (Component, ConcreteComponent, Decorator) :=
                    Inherit (Decorator, Component)
                    &(Contain(Decorator, Component)
                    | Use (Decorator, Component))
                    & Inherit (ConcreteComponent, Component);
PRINT "Decorator: " , #(Decorator(Component,_,_)) , ";" , ENDL;

Facade (Product1, Product2, Product3,Product4) :=
                    Use(Product2,Product1)
                    & Use(Product3,Product1)
                    & Use(Product4,Product1);
PRINT "Facade: " , #(Facade(Product1,_,_,_)) , ";" , ENDL;

Flyweight1(FlyweightFactory, Flyweight, ConcreteFlyweight,
                    UnshardConcreteFlyweight, Client) :=
                      Inherit (UnshardConcreteFlyweight, Flyweight)
                    & Inherit (ConcreteFlyweight, Flyweight)
                    & (Contain (FlyweightFactory, Flyweight)
                    | Contain (FlyweightFactory, ConcreteFlyweight))
                    & Use (Client, UnshardConcreteFlyweight)
                    & Use (Client, ConcreteFlyweight)
                    & Use (Client, FlyweightFactory);
nFly1 := #(Flyweight1(FlyweightFactory,_,_,_,_));

Flyweight2 (FlyweightFactory, Shared, Unshard, AbstractFlyweight) :=
                      Inherit (Unshard, AbstractFlyweight)
                    & Inherit (Shared, AbstractFlyweight)
                    & Contain (FlyweightFactory, Shared) ;
```

```
nFly2 := #(Flyweight2(FlyweightFactory,_,_,_));
PRINT "Flyweight: " , nFly1 + nFly2 , ";" , ENDL;

IteratorP(Iterator, Aggregate, ConcreteIterator, ConcreteAggregate) :=
                        Inherit (ConcreteIterator, Iterator)
                      & Use (ConcreteIterator, ConcreteAggregate)

                      & Inherit (ConcreteAggregate, Aggregate);
PRINT "Iterator: " , #(IteratorP(Iterator,_,_,_)) , ";" , ENDL;

MediatorP1(Mediator, Colleague, ConcreteMediator, ConcreteColleague) :=
                        Inherit (ConcreteMediator, Mediator)
                      & Use (Colleague, Mediator)
                      & Inherit (ConcreteColleague, Colleague)
                      & Use (ConcreteMediator, ConcreteColleague);
nMediator1 := #(MediatorP(Mediator,_,_,_));

MediatorP2(Mediator, Event1, Event2, Event3, Colleague1, Colleague2) :=
                        Inherit (Mediator, Event1)
                      & Inherit (Mediator, Event2)
                      & Inherit (Mediator, Event3)
                      & Use (Mediator, Colleague1)
                      & Use (Mediator, Colleague2);
nMediator2 := #(MediatorP2(Mediator,_,_,_,_,_));
PRINT "Mediator: " , nMediator1 + nMediator2, ";" , ENDL;

MementoP(Memento, Caretaker) :=
                        Contain (Caretaker, Memento);
PRINT "Memento: " , #(MementoP(Memento,_)) , ";" , ENDL;
```

```
ObserverP(Observer, Subject, ConcreteObserver, ConcreteSubject) :=
                        Inherit (ConcreteObserver, Observer)
                      & Use (Subject, Observer)
                      & Inherit (ConcreteSubject, Subject)
                      & Use (ConcreteObserver, ConcreteSubject);
PRINT "Observer: " , #(ObserverP(Observer,_,_,_)) , ";" , ENDL;

PrototypeP(Prototype, ConcretePrototype, Client) :=
                        Inherit (ConcretePrototype, Prototype)
                      & Use (Client, Prototype);
PRINT "Prototype: " , #(PrototypeP(Prototype,_,_)) , ";" , ENDL;

PRINT "Singleton: " , "1;" , ENDL;

Proxy (Subject, RealSubject, Proxy) :=
                        Inherit (RealSubject, Subject)
                      & Use (Proxy, RealSubject);
PRINT "Proxy: " , #(Proxy(Subject,_,_)) , ";" , ENDL;

StateP(State, ConcreteState, Context) :=
                        Inherit (ConcreteState, State)
                      & Contain (Context, State);
PRINT "State: " , #(StateP(State,_,_)) , ";" , ENDL;

StrategyP(Strategy, ConcreteStrategy, Context) :=
                        Inherit (ConcreteStrategy, Strategy)
                      & Contain (Context, Strategy);
PRINT "Strategy: " , #(StrategyP(Strategy,_,_,_)) , ";" , ENDL;

TemplateMethod (AbstractClass, ConcreteClass) :=
                        Inherit (ConcreteClass, AbstractClass);
PRINT "Template Method: " , #(TemplateMethod(AbstractClass,_)) , ";" , ENDL;
```

```
Visitor (Element, ConcreteElement, ObjectStructure) :=
        Inherit (ConcreteElement, Element)
        & Use (ObjectStructure, Element);
PRINT "Visitor: " , #(Visitor(Element,_,_)) , ";" , ENDL;
```

ประวัติผู้เขียนโครงงาน

ชื่อ นายปุณณวัฒน์ ธาดาภาคย์

ภูมิลำเนา 72/26 หมู่ที่ 7 ตำบลวัดจันทร์ อำเภอเมือง จังหวัดพิษณุโลก 65000

ประวัติการศึกษา

- จบการศึกษาระดับมัธยมศึกษาจาก โรงเรียนพิษณุโลกพิทยาคม
- ปัจจุบันกำลังศึกษาอยู่ชั้นปีที่ 4 สาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร

E-mail punnawatt2@yahoo.com

ชื่อ นายพรพงษ์ ร้องเกาะเกิด

ภูมิลำเนา 32 ถนนสนามบิน อำเภอเมือง จังหวัดพิษณุโลก 65000

ประวัติการศึกษา

- จบการศึกษาระดับมัธยมศึกษาจาก โรงเรียนพิษณุโลกพิทยาคม
- ปัจจุบันกำลังศึกษาอยู่ชั้นปีที่ 4 สาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร

E-mail taeppr@hotmail.com