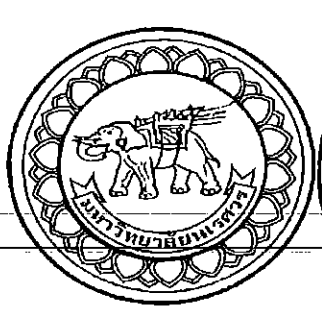


โปรแกรมตรวจหาดีไซน์แพทเทอร์น

Mining Design Pattern Program

นายเฉลิมพล สุภาษิ รหัส 46360228

นายศุภชาติ เสนวิรัช รหัส 46362125

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมไฟฟ้าและคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร

ปีการศึกษา 2549

ใบรับรองโครงงานวิศวกรรม

หัวข้อโครงงาน โปรแกรมตรวจหาดีไซน์แพทเทอร์น Mining Design Pattern Program

ผู้ดำเนินโครงงาน นายเฉลิมพล สุภาษิ รหัส 46360228

นายศุภชาติ เสนวิรัช รหัส 46362125

อาจารย์ที่ปรึกษา ดร.สุรเดช จิตประไพกุลศาล

สาขาวิชา วิศวกรรมคอมพิวเตอร์

ภาควิชา วิศวกรรมไฟฟ้าและคอมพิวเตอร์

ปีการศึกษา 2549

.....

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร อนุมัติให้โครงงานฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิศวกรรมศาสตรบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์

คณะกรรมการสอบโครงงานวิศวกรรม

.....ประธานกรรมการ

(ดร.สุรเดช จิตประไพกุลศาล)

.....กรรมการ

(ดร.พนมขวัญ ริยะมงคล)

.....กรรมการ

(อาจารย์จิราพร พุกสุข)

หัวข้อโครงงาน	โปรแกรมตรวจหาดีไซน์แพทเทอร์น Mining Design Pattern Program			
ผู้ดำเนินโครงงาน	นายเฉลิมพล	สุภาษิ	รหัส	46360228
	นายศุภชาติ	เสนวิรัช	รหัส	46362125
อาจารย์ที่ปรึกษา	ดร. สุรเดช	จิตประไพกุลศาล		
สาขาวิชา	วิศวกรรมคอมพิวเตอร์			
ภาควิชา	วิศวกรรมไฟฟ้าและคอมพิวเตอร์			
ปีการศึกษา	2549			

บทคัดย่อ

โครงงานนี้จะเป็นการประยุกต์ให้ตัวช่วยวิเคราะห์ไวยากรณ์และตัวจำดีไซน์แพทเทอร์น หรือ โปรแกรมโครโคแพท เพื่อนำมาหา ดีไซน์แพทเทอร์นในโค้ดของภาษาซีชาร์ป และการทำงานของโปรแกรม ในการตรวจหาแพทเทอร์น จะขึ้นอยู่กับ ภาษาเฉพาะของตัวตรวจหาแพทเทอร์น ซึ่งเราสามารถสร้างเองโดยใช้ความสัมพันธ์ทาง โครงสร้างของภาษาที่เป็นภาษาอาเอ็มแอล ซึ่งภาษาอาเอ็มแอลจะเปลี่ยนรูปจากแผนภาพยูเอ็มแอล มาเป็นความสัมพันธ์ของโครงสร้างดีไซน์แพทเทอร์นทางภาษา แล้วนำไปใช้เปรียบเทียบกับภาษาอาเอสเอฟ ว่ามีแพทเทอร์นที่เหมือนกันหรือไม่ และในโปรแกรมก็สามารถแพทเทอร์นใหม่ในโค้ดของภาษาซีชาร์ป ออกมาเป็นไฟล์อาร์เอมแอลได้ ผลลัพธ์ของโปรแกรมจะแสดงจำนวนแพทเทอร์นว่ามีเท่าไรในโค้ดที่นำมาทดสอบ ในตัวโปรแกรมจะสร้างไฟล์อาเอสเอฟ ซึ่งจะถูดสร้างจากตัวช่วยวิเคราะห์ภาษาอีกทีหนึ่ง คือ โคโคอา และในอาเอสเอฟก็จะแสดงความสัมพันธ์เชิงวัตถุของ โค้ดที่นำมาทดสอบ โปรแกรมโคโคอา เรานำมาช่วยวิเคราะห์ หรือว่าแปลภาษา ซีชาร์ป ให้ได้โค้ดในรูปแบบที่เราต้องการได้

Project Title	Mining Design Pattern Program	
Name	Mr.Chalermpon Supasi	ID. 46360228
	Mr.Supachat Sanwirach	ID. 46362125
Project Advisor	Dr.Suradet Jitprapaikulsarn	
Major	Computer Engineering	
Department	Electrical and Computer Engineering	
Academic Year	2006	

ABSTRACT

This project effort to apply the parser and recognizer tool of design pattern for detecting. Design pattern from source code (C#). Its performance runs to not depend on form of recognizer's language, but we must generate form by using Relational Manipulation Language (RML). The RML will change UML to relational language which uses to compare pattern with RSF file (Rigi Standard Format) by crocopat tool. The task of project will call crocopat tool for compare RML and RSF file that are them match or not. Its consequence will show number of pattern which detected in RSF file (RSF file will generate by parser called COCO/R) depict all class name which have relation be its pattern. In program we use COCO/R parser which will parse code C# where be input of program.

กิตติกรรมประกาศ

ขอขอบพระคุณ ดร.สุรเดช จิตประไพกุลศาล ซึ่งเป็นอาจารย์ที่ปรึกษาโครงงานนี้ ที่คอยให้คำแนะนำและปรึกษาเมื่อเกิดปัญหาในโครงงานและดูแลความเรียบร้อยของงานในส่วนต่างๆ และขอขอบพระคุณอาจารย์กรรมการทุกท่านที่ได้มาตรวจสอบความสมบูรณ์ของโครงงาน ให้โครงงานนี้สำเร็จได้เป็นอย่างดี และขอขอบคุณ คุณปุณณวัฒน์ ธาดาภาคย์ เพื่อนที่ให้คำแนะนำในส่วนของโปรแกรมที่เป็นเครื่องมือช่วยเหลือที่นำมาใช้ในโปรแกรมที่ทำขึ้นและขอขอบคุณผู้จัดทำโปรแกรมที่เป็นเครื่องมือช่วยเหลือต่างๆ ไม่ว่าจะเป็นโปรแกรมช่วยวิเคราะห์ภาษา Parser Generater โปรแกรมตรวจหาดีไซน์แพทเทอร์น Crocopat มา ณ โอกาสนี้ด้วย

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

ตารางที่ หน้า

สารบัญรูปภาพ

รูปที่ หน้า

สารบัญรูปภาพ (ต่อ)

รูปที่ หน้า

บทที่ 1
บทนำ

1.1 ที่มาและความสำคัญ

ในปัจจุบันนี้เทคโนโลยีด้านคอมพิวเตอร์ได้มีการพัฒนาไปอย่างรวดเร็ว และทำให้มีการใช้คอมพิวเตอร์เพื่อช่วยในการทำงานกันอย่างแพร่หลายซึ่งก็ทำให้เกิดการเขียนโปรแกรมคอมพิวเตอร์เพื่อใช้งานในด้านต่างๆมากมาย ดังนั้นจึงมีความพยายามที่จะพัฒนาเพื่อการเขียนโปรแกรมที่ดีและมีประสิทธิภาพโดยพิจารณาจากลักษณะของการเขียนโปรแกรมที่ผ่านมาส่วนใหญ่จะเป็นการนำเอารูปแบบลักษณะที่คล้ายๆกันมาใช้ และใช้ซ้ำๆกัน เป็นการนำรูปแบบต่างๆมาใช้โดยธรรมชาติไม่มีการจัดประเภทหมวดหมู่หรือบันทึกเอาไว้ แต่มีคนกลุ่มหนึ่ง (Gang of Four) ที่เล็งเห็นประโยชน์ของ Design Pattern ก็รวบรวมและจัดหมวดหมู่เพื่อให้เป็นประโยชน์ต่อส่วนรวม และง่ายต่อการอ้างอิง ช่วยให้การเขียนโปรแกรมนั้นง่ายขึ้น

Design Pattern ช่วยให้การพัฒนาโปรแกรมนั้นง่ายขึ้น โดยการใช้เทคนิคที่พิสูจน์แล้วว่าใช้งานได้จริง คือ Design Pattern แต่ละตัวต้องมาจากสิ่งที่โปรแกรมอื่นเคยใช้จริงมาแล้วมาเก็บรวบรวมและจัดหมวดหมู่เพื่อนำมาใช้เป็นเครื่องมือช่วยในการเขียนโปรแกรม โดยถ้านักพัฒนาโปรแกรมที่รู้จัก Design Pattern แต่ละตัว ก็สามารถสื่อสารกันง่ายขึ้น เช่น ถ้าต้องการสร้างระบบติดตามราคาหุ้น อาจออกแบบส่วนติดต่อตลาดหุ้นด้วย Observer Pattern ส่วน User Interface ใช้ MVC pattern ส่วนรับอินพุทจากผู้ใช้ ใช้ Command Pattern คนที่รู้รายละเอียดของแต่ละรูปแบบก็จะรู้ว่าต้องเขียนโปรแกรมออกมาอย่างไร

การตรวจหา Pattern ที่มีอยู่ใน Source Code นั้น เป็นการทำเพื่อค้นหารูปแบบที่มีการใช้งานในการเขียนโปรแกรมเพื่อจะทำให้สามารถรู้ได้ว่ามีรูปแบบใดบ้างที่มีการใช้งาน ในการเขียนโปรแกรมที่มีการทำงานในลักษณะต่างๆซึ่งมีการใช้กันอยู่แล้ว โดยจะทำให้ได้รูปแบบที่ใช้ในการเขียนโปรแกรมที่ทำงานในลักษณะนั้น ซึ่งมีการใช้กันอยู่แล้วเพื่อนำมาเก็บรวบรวมเป็นข้อมูลที่สามารถทำความเข้าใจได้โดยใช้ภาษาสำหรับ Design Pattern เพื่อสามารถเป็นประโยชน์ในการนำมาวิเคราะห์และตรวจสอบหาว่ารูปแบบใดเป็นรูปแบบที่ดี และมีประสิทธิภาพในการเขียนโปรแกรมในลักษณะต่างๆ ต่อไปและอาจทำการเพื่อตรวจสอบว่ารูปแบบดังกล่าวที่ได้มานั้นเป็นรูปแบบที่มีอยู่แล้วใน Design Pattern ที่มีการเก็บรวบรวมไว้แล้วหรือไม่

1.2 วัตถุประสงค์ของโครงงาน

1. ศึกษาทฤษฎีและหลักการเขียนโปรแกรมโดยใช้ Design Pattern ช่วยในการเขียนโปรแกรม
2. สามารถตรวจหาและเก็บรวบรวม Pattern ในการเขียนโปรแกรม เป็นภาษาสำหรับ Design Pattern

1.3 ขอบข่ายของโครงงาน

1. ศึกษาการใช้ Design Pattern เพื่อช่วยในการเขียนโปรแกรม
2. ทำการตรวจหา Pattern จาก Source Code ของโปรแกรม Open Source ต่างๆ

1.4 ขั้นตอนดำเนินงาน

1. ศึกษาเกี่ยวกับทฤษฎีและหลักการเกี่ยวกับ Design Patterns
2. ออกแบบและพัฒนาโปรแกรม
3. ทดสอบโปรแกรม
4. ทำการปรับปรุงและแก้ไขโปรแกรม
5. วิเคราะห์ผลการทดสอบ
6. สรุปผล
7. จัดทำรูปเล่มและการนำเสนอ

1.5 แผนการดำเนินงาน

กิจกรรม	ปี 2548		ปี 2549									
	พ.ย.	ธ.ค.	ม.ค.	ก.พ.	มี.ค.	เม.ย.	พ.ค.	มิ.ย.	ก.ค.	ส.ค.	ก.ย.	ต.ค.
1. ศึกษาหลักการและทฤษฎี	√	√	√									
2. ออกแบบและพัฒนาโปรแกรม			√	√								
3. ทดสอบโปรแกรม				√	√							
4. ทำการปรับปรุงและแก้ไขโปรแกรม					√	√						
5. วิเคราะห์การทดสอบพร้อมทั้งสรุปผล					√	√						
6. จัดทำรูปเล่มโครงงาน						√	√					

1.6 ผลที่คาดว่าจะได้รับ

1. เข้าใจในหลักการเขียนโปรแกรมโดยใช้ Design Patterns ช่วยในการเขียนโปรแกรม
2. สามารถตรวจสอบได้ว่าใน Source Code มีการใช้มี Design Patterns อะไรบ้าง

1.7 งบประมาณ

1. ค่าวัสดุสำนักงาน	เป็นเงิน	400 บาท
2. ค่าวัสดุคอมพิวเตอร์	เป็นเงิน	500 บาท
3. ค่าถ่ายเอกสาร	เป็นเงิน	300 บาท
4. ค่าวัสดุอื่นๆ	เป็นเงิน	800 บาท
	รวม	**2,000 บาท (สองพันบาทถ้วน)**

บทที่ 2
ทฤษฎีและหลักการ

ในทฤษฎีและหลักการของ Design Pattern เป็นศาสตร์ที่ค่อนข้างใหม่ และในการศึกษาถึงแม้จะมีรูปแบบที่เห็นเป็นรูปธรรมอยู่บ้างแล้วก็ตามแต่ก็ยังใช้หลักการของ Design Pattern ในงานใหม่ๆได้น้อยมาก จากเดิมที่โปรแกรมส่วนมากที่ทำไว้จะพัฒนาโดยใช้ภาษา Java แต่การประยุกต์สามารถใช้ได้กับภาษา C++, C# และอื่นๆอีกโดยอาศัยโครงสร้างของโปรแกรม (Pattern of Code) เดียวกัน และประยุกต์ใช้ในแต่ละภาษาได้

Design Pattern นั้นเปรียบได้เป็นสถาปัตยกรรมที่เป็นรูปแบบที่พิสูจน์แล้วว่าใช้ได้จริง และเป็นเครื่องมือที่มีประสิทธิภาพในการเขียนโปรแกรม ดังนั้นในตัวโปรแกรมต่างๆ ก็จะมีลักษณะที่ซ้ำกันได้ในการเขียน (Same Characteristic) หากใช้หลักการนี้เขียนโปรแกรมขนาดใหญ่ก็จะทำได้ง่ายและมีข้อผิดพลาดน้อย

Gang of Four ซึ่งได้รวบรวมและวิเคราะห์ Design Pattern ที่เป็นไปได้ในการ Reusable 23 Patterns ด้วยกัน

มีองค์ประกอบอยู่ 3 อย่างที่สำคัญสำหรับ Design Pattern ที่แบ่งตามหลักการของ GoF คือ

- Creational: Patternในส่วนนี้จะพิจารณาถึงการสร้างรูปแบบเชิงวัตถุของโปรแกรม (Object Program) ขึ้นมา เราสามารถออกแบบชนิดของตัววัตถุที่เขียนได้ แต่ก็ยังไม่เพียงพอที่จะนำมาผสมกับโครงสร้างของ Pattern ได้ เพราะในความเป็นจริงยังจะต้องมีเครื่องมือที่มากกว่านี้ในการทำให้เกิดผลลัพธ์ที่ต้องการซึ่งอาจจะเป็นส่วนที่ซ่อนอยู่ (Function Implementation) โดยการสร้าง Creational Pattern เป็นเพียงการบอกลักษณะของ Source Code ที่จะมี Object อย่างไรและใช้ Object เหล่านั้นอย่างไรบ้างเท่านั้น ซึ่งในการแก้ปัญหาเกี่ยวกับ Object จะประกอบด้วย 5 Patterns คือ Factory, Abstract Factory, Builder, Prototype และ Singleton
- Structural: Pattern ในส่วนนี้จะเป็นการบอกถึงโครงสร้างความสัมพันธ์ของ Class ซึ่งจะกำหนดลำดับของ Class หรือความสัมพันธ์ของ Class และการรวมกันของ Class และ Object ในโครงสร้างที่ใหญ่ขึ้น ในการแก้ปัญหาของ Structural นี้จะประกอบด้วย 7 Patterns คือ Adapter, Bridge, Composite, Decorator, Façade, Flyweight และ Proxy
- Behavioral: Pattern ส่วนนี้จะบอกถึงการติดต่อระหว่างกัน (Communication) ของ Object โดยกำหนดรูปแบบการกระทำไว้ในตัวกระทำ (Operation) ซึ่ง Pattern รูปแบบนี้จะเป็นรูปแบบที่รู้จำ (Recognition) ที่ยากที่สุดในจำนวนทั้ง 3 รูปแบบ ในการแก้ปัญหาพฤติกรรมของ Object จะประกอบไปด้วย 11 Patterns คือ Chain of Responsibility, Command, Interpreter, Interator, Mediator, Memento, Observer, State, Strategy, Template Method และ Visitor

2.1 Design Patterns

ในหัวข้อนี้จะอธิบายถึงราบละเอียดของ แนวความคิดที่เป็นที่มาของ Pattern โครงสร้าง การใช้งานและข้อจำกัดของการใช้งานในแต่ละ Pattern

ตัวอย่างของ Design Pattern

ในPattern ส่วนนี้จะอธิบายถึงการสร้าง Object อย่างมีหลักการและทำให้เกิดประโยชน์มากที่สุด และใช้งานได้อย่างมีประสิทธิภาพ ไม่เกิดความซ้ำซ้อนของหน้าที่หรือสร้าง Object มาเกินความจำเป็น ซึ่งแนวคิดของการสร้าง Object จะพิจารณาในส่วนของ Creational Pattern นี้ซึ่งจะมีด้วยกัน 5 patterns ได้แก่

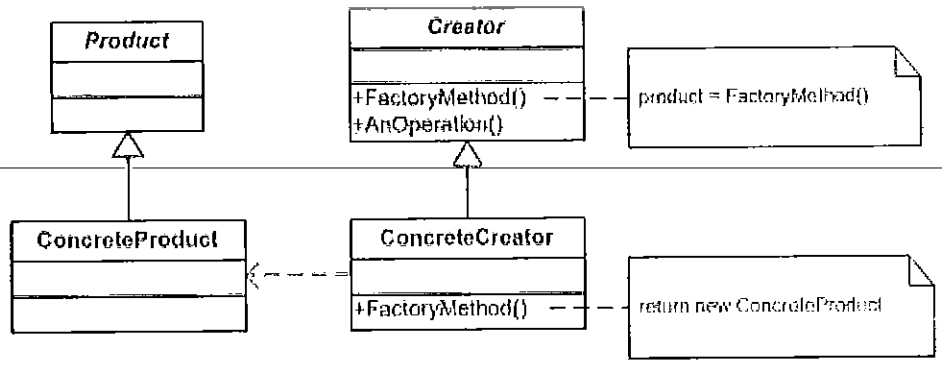
1. Factory Method

1.1 แนวความคิด.

สมมุติว่ามี Object อยู่ 1 ตัวที่ทำหน้าที่เป็นเสมือนโรงงานโดยจะใช้เพื่อสร้าง Object ของSub class ที่มี Super Class เดียวกันเพื่อทำให้เกิดผลตามที่ต้องการในแต่ละ Subclass ซึ่งเปรียบเทียบเหมือนโรงงาน (Factory) ที่ผลิตสินค้าประเภทหนึ่ง แต่โรงงานสามารถที่จะผลิตสินค้าประเภทนั้นได้ในหลายรูปแบบ ขึ้นอยู่กับว่าจะมีคำสั่ง (Oder) อะไรเข้ามา

ในรูปที่ 2-1 จะแสดงถึงการสร้าง Object ของ Sub Class ว่าต้องสร้างตอนไหนและมีเงื่อนไขอะไรที่จะต้องสร้าง

1.2 องค์ประกอบ



รูปที่ 2.1 Factory Method Pattern [11]

- **Product**
  - เป็น Class หรือ Interface ต้นแบบของ Factory
  - เป็นตัวกำหนดคุณสมบัติร่วมของ object ที่จะสร้างจาก factory method
- **ConcreteProduct**
  - เป็น Subclass ของ Product ใช้ถ่ายทอดคุณสมบัติที่จำเป็นของ Product ให้กับตัวเองแล้วก็จะถูกร้องขอให้ทำงานจากตัวอื่นอีกทีหนึ่ง
- **Creator**
  - เป็นตัวกำหนดคุณสมบัติร่วมของ Object ที่ทำหน้าที่สร้าง Sub Class หรือว่าสร้าง Implement Class ของ Class Product ขึ้นมาตามผลที่ต้องการ
  - อาจจะทำการเรียก Factory Method เพื่อสร้าง Object ของ Product ได้
- **ConcreteCreator**
  - เป็น Class ที่ทำการ Override Method FactoryMethod() ของ Creator
- **Client**
  - เป็น Class ที่ร้องขอให้ Factory ทำงานตามที่ต้องการ

1.3 การทำงาน

เวลาที่ต้องการผลลัพธ์ของหลายๆสิ่งที่บางครั้งสิ่งเหล่านั้นมีคุณสมบัติพื้นฐานที่คล้ายคลึงกันเช่น ถ้าต้องการโต๊ะ เก้าอี้ หรือ ชุดรับแขก จะต้องเข้าไปใช้โรงงานที่ผลิตเฟอร์นิเจอร์ ที่ขีดความสามารถของโรงงานสามารถผลิตสินค้าได้หลายรูปแบบนั่งเอง

หากเป็นทาง Software ก็เช่นเดียวกัน เมื่อเรามี Object เริ่มต้นของ Class ใดๆ (Client Class) โดยทำการเรียก factoryMethod () ของ Class ConcreteCreator พร้อมกับมีเงื่อนไขบางอย่างส่งมาด้วย เสร็จแล้วจะเป็นหน้าที่ของ ConcreteCreator ที่จะทำงานสร้าง Object ของ ConcreteProduct เพื่อทำให้เกิดผลตามเงื่อนไขที่ส่งเข้าระบบมาตั้งแต่ตอนแรกนั่นเอง

1.4 ข้อดี

ใน Pattern นี้จะไม่สามารถรู้ถึงการทำงานภายในของ Factory เลยเพราะว่ามีตัวแทนคือ Class ConcreteCreator ทำหน้าที่แทนในการสั่งงาน Factory ดังนั้นจึงไม่จำเป็นต้องรู้ถึงโครงสร้างทั้งหมดของ Factory จะต้องรู้เฉพาะว่าจะใช้งานอย่างไรบ้างเท่านั้นเอง

1.5 ข้อจำกัด

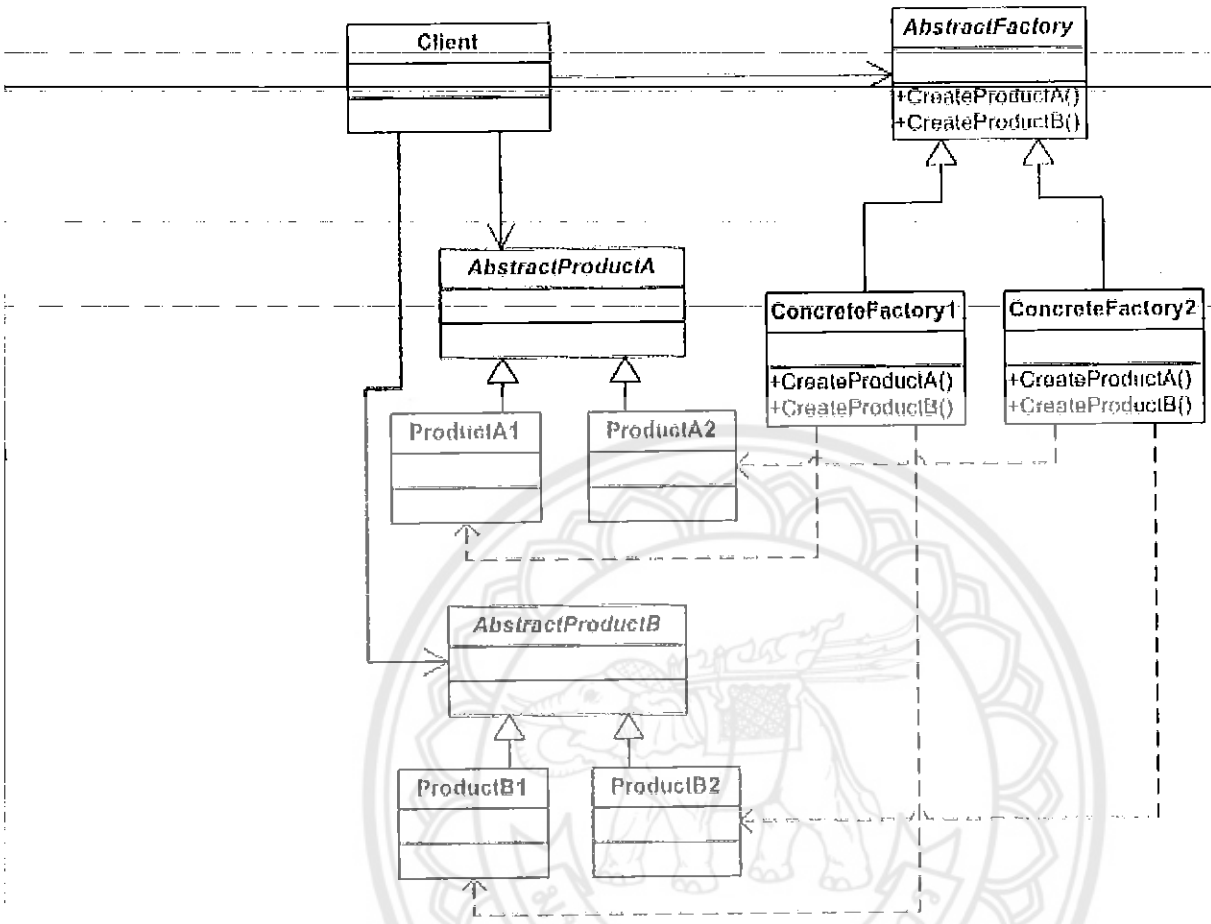
เมื่อไม่มีความจำเป็นต้องรู้โครงสร้างภายในของ Factory เวลาจะเปลี่ยนแปลงโครงสร้างภายในก็ทำได้ยากมากขึ้น เช่นเราต้องการ Product ใหม่ที่นอกเหนือจากที่มีใน Factory หรือว่าเปลี่ยนประเภทของผลลัพธ์ที่ต้องการ เราก็ต้องไปแก้ไข Factory ใหม่เกือบทั้งหมด หรือเปลี่ยนตัว Factory ใหม่ทั้งหมดเลยก็เป็นไปได้

2. Abstract Factory Method

2.1 แนวความคิด

เป็นแนวคิดที่นำเอา Factory มาใช้งานที่มีความซับซ้อนมากขึ้น หรือกล่าวได้ว่าเป็น Factory ของ Factory เช่น เราต้องการ Product ประเภทใดประเภทหนึ่ง ณ เวลาใดเวลาหนึ่ง แต่ ระบบสามารถตอบสนองการเปลี่ยนแปลงเมื่อเราต้องการเปลี่ยน Product ณ เวลาใดเวลาหนึ่งได้เช่นกัน การทำงานแบบนี้ต้องความยืดหยุ่นและมีกลไกอย่างหนึ่งที่จะเลือกว่าให้ระบบทำงานอะไร

2.2 องค์ประกอบ



รูปที่ 2.2 Abstract Factory Method Pattern [11]

- **AbstractFactory**
  - เป็น Interface ใช้สำหรับประกาศการทำงานของผลลัพธ์ที่ต้องมีทั้งหมดเช่น ในรูปจะมี method CreateProductA() และ CreateProductB()
- **ConcreteFactory**
  - เป็น Class ที่สืบทอดมาจาก AbstractFactory โดยจะเป็นตัวเลือกที่จะถูกเลือกให้ทำงาน ณ เวลาใดเวลาหนึ่ง และจะต้องเลือกเพียงตัวเดียวเท่านั้น ในรูปจะมี Class ConcreteFactory1 และ ConcreteFactory2
- **AbstractProduct**
  - เป็น Interface ที่จะประกาศ method ของ Product สำหรับ Object ที่จะสร้างใหม่แต่ละชนิดตามประเภทของ ConcreteFactory แต่ละตัว

- **Product**
  - เป็น Class ที่ทำการ Inheritance หรือ Implement มาจากAbstractProduct แต่ละตัว และมีตามจำนวนของชนิดของ Product และตามจำนวนชนิดของ ConcreteFactory ด้วย
  - จะทำการประกาศ Product Object ของแต่ละ ConcreteFactory
- **Client**
  - เป็น Class ที่ทำการร้องขอการใช้ AbstractFactory และ AbstractProduct classes

2.3 การทำงาน

จะมี Client เข้ามาสร้าง Object เรียก method ของ AbstrctFactory ขึ้น และจะติดต่อผ่าน Interfacing ของระบบเพื่อเข้าถึงการให้ผลลัพธ์ โดยไม่จำเป็นต้องรู้การทำงานภายใน แต่การทำงานภายในก็จะเป็น Factory เหมือน Pattern ที่ 1 แล้วแต่เงื่อนไขของการเลือกว่าจะเลือก ConcretFactory ตัวไหน แล้ว ConcreteFactory ตัวนั้นก็จะทำงานต่อไปจนได้ผลลัพธ์ตามประเภทของ Product ของ Factory นั้นๆ

2.4 ข้อดี

สามารถเลือกได้ว่า ณ เวลาใดเวลาหนึ่งเราต้องการให้ระบบแสดงผลของ Factory ตัวไหน สามารถนำไปใช้กับ User Interface ของ ระบบปฏิบัติการที่แตกต่างกันได้ โดยระบบ Abstract Factory จะมีส่วนของ Product ที่จำเป็นต้องมีในทุกระบบปฏิบัติการที่สามารถเลือกได้

และ AbstrctFactory จะคล้ายกับ Factory ที่ลดความยุ่งยากของการสร้าง Object โดย Client เพียงแต่ติดต่อ Interface ที่มีให้ แล้วตัวของ Factory จะจัดการเรื่อง Object ให้เองทั้งหมด

2.5 ข้อจำกัด

ในการเพิ่มหรือลด Product จะมีผลกระทบต่อ ConcreteFactory ทุกตัวโดยจะมีความยุงยากในการเปลี่ยนแปลง Product เพราะ ConcreteFactory เป็นโครงสร้างสำคัญที่ต้องติดต่อกับ Product ทุกๆ ตัว ดังนั้น AbstractFactory จึงไม่เหมาะที่จะนำไปใช้กับงานที่ต้องเปลี่ยนแปลง Product บ่อยๆ

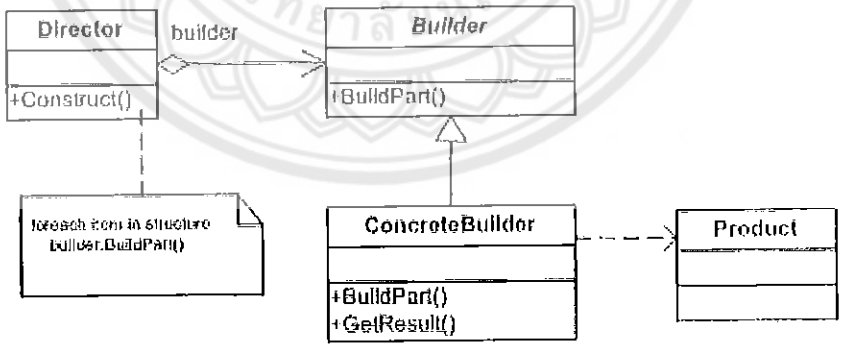
3. Builder

3.1 แนวความคิด

ข้อจำกัดของ Factory Pattern คือ Sub Class ทุก Class มี Super Class เพียง Class เดียว บางครั้งเราต้องการข้อมูลที่แตกต่างกันโครงสร้างโดยรวมของ ระบบอาจจะเปลี่ยนแปลงไป เช่นเราต้องการ Product ใหม่ใน Factory หรือ AbstractFactory จะต้องมีการเปลี่ยนแปลง Object ใหม่ทั้งโครงสร้าง และ ใน Builder เป็นการรวมกันของหลายๆ Object ซึ่งจะเหมาะกับงานที่ข้อมูลมีการเปลี่ยนแปลงบ่อย

ใน Factory Pattern เราจะเห็นว่าผล (Product) ที่ได้จะมีเพียง 1 อย่างจากการทำงานของ Subclass หลายๆอัน ถ้าเราต้องการผล (Product) หลายอย่างจากระบบที่ค่อนข้างจะคงที่และไม่เปลี่ยนแปลง เราจะต้องใช้ Builder Pattern เพราะว่าจะให้ผลได้หลายอย่างซึ่งต้องขึ้นอยู่กับข้อมูลที่เข้ามาในระบบ (Data) เช่น User Interface ที่รูปแบบขึ้นกับ Data ที่รับมา เช่น E-mail Address Box จะมีทั้ง คนที่ติดต่อ และกลุ่มของคนเหล่านั้น เมื่อเราต้องการแสดง (Display) เฉพาะคนที่ติดต่อหน้าตาของ Screen ก็จะมี ชื่อ, สกุล, ที่อยู่, เบอร์โทร, E-mail เป็นต้น แต่ถ้าเราต้องการแสดงกลุ่มของคนเหล่านั้นจะต้องแสดง ชื่อของกลุ่ม, ลักษณะของกลุ่ม, รายชื่อสมาชิกในกลุ่ม, E-mail Address และอื่นๆ โดยการแสดงผลมี Form ของ GUI ที่แตกต่างกันทั้งคู่

องค์ประกอบ



รูปที่ 2.3 Builder Pattern [11]

- **Builder**
  - เป็น Interface ของส่วนหรือ Class ที่จะสร้างองค์ประกอบต่างๆ ของ Product Object
- **ConcreteBuilder**
  - เป็น Class ที่สร้างและทำการรวมส่วนประกอบต่างๆของ Product โดยการ Implement BuilderPart Method ของ Builder ก่อน
  - เป็นตัวกำหนดและคงรูปแบบของการทำงานในระบบของ Builder Pattern
- **Director**
  - เป็น Class ที่ใช้สร้าง Object ที่จะใช้ใน Builder Interface
- **Product**
  - เป็นตัวรวมของ Object ภายใต้โครงสร้าง Product อันเดียวซึ่งมีความซับซ้อน
  - และรวม Class ที่กำหนดส่วนประกอบต่าง, รวม Interface สำหรับตัวทำงานที่จะให้ผลลัพธ์สุทธิออกมา

3.2 การทำงาน

มี Object ของ Class ใดๆ มาทำการสร้าง Object ของ ConcreteBuilder และสร้าง Object ของ Class Director ขึ้นมาโดยการใส่ Object ของ Class ConcreteBuilder ที่สร้างขึ้นมาแต่แรกมาเป็น Parameter เพื่อบ่งบอกว่าจะต้องการให้ Product นั้นออกมาเป็นรูปแบบใดโดยการให้ Object ของ director เข้ามากำกับดูแลการทำงานของ Object ที่สร้างตัวแรกจาก Class ConcreteBuilder แล้วก็จะทำงานโดยการใช้ Method buildPart ของ Class ConcreteBuilder เพื่อสร้าง Production ที่ต้องการ เมื่อสร้างเสร็จก็จะใช้ Method getResult นำผลลัพธ์นั้นออกมา

3.3 ข้อดี

เพียงแต่เรามีตัว Builder Pattern ก็ได้ Product หรือผลที่ต้องการแล้ว ถึงแม้ว่าการสร้าง Object ของ Builder Pattern จะเป็นเรื่องยุ่งยาก แต่เราก็ไม่ได้เข้าไปยุ่ง เราเพียงแต่รู้ว่ามี Service หรือการใช้ Method ใดที่ Builder Pattern ได้จัดไว้ให้ ซึ่งจะทำงานอิสระจากกันระหว่าง การเรียกใช้งานเพื่อต้องการผลลัพธ์ และการทำงานภายในระบบ

Object ที่ถูกสร้างขึ้นจะมาจาก Class เพียง Class เดียว แต่องค์ประกอบและการทำงานสามารถทำได้ในหลายๆ Product แล้วแต่เงื่อนไขที่เข้ามาเรียกตั้งแต่เริ่มต้น จะแตกต่างกับ Factory ตรงที่ Factory มีประเภทของ Object ที่แตกต่างกันเพราะขึ้นอยู่กับ Subclass ที่มีและยังมีความยืดหยุ่นของการสร้าง Product น้อยกว่า Builder

3.4 ข้อจำกัด

ในส่วนของ Product ถ้าหากทำการออกแบบจะมีความซับซ้อน และยุ่งยากมาก เพราะเป็นที่รวมของ Object ของระบบทั้งหมดโดยผลลพัธ์จะเกิดจากการเรียกใช้ภายนอกตามจำนวน Product ที่มีและจะเรียกใช้งานนอกเหนือจาก Product ที่ระบบมีไม่ได้ ถ้าเรียกเราต้องทำการสร้าง Object เพิ่มเติมจากที่มีอยู่เดิม และอาจจะต้องสร้างโครงวสร้างและความสัมพันธ์ใหม่ที่ซับซ้อนมากขึ้น

3.5 ความแตกต่างระหว่าง Builder Pattern และ Factory Pattern

Factory Pattern จะแสดงถึงสิ่งต่างๆ ที่สามารถเป็นทางเลือก และแสดงออกมาเป็นรูปธรรมในขณะที่มีการทำงานได้ (Boject to make until run time) เช่น การรอให้ Interface สั่งให้ Factory ทำการ Generic Meassage "Get me the special of the daay" และก็ return ค่าออกมา เป็น Product ที่เป็นรูปธรรม

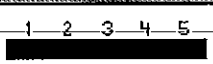
Builder Pattern จะถูกห่อหุ้มใน Logic การทำงาน ที่จะทำการรวม Object ให้มีความซับซ้อนมากยิ่งขึ้น เช่น Client ต้องการ Configuration ของระบบ และ Builder จะทำการสร้าง Logic ขึ้นมารองรับโดยตรง

ใน Factory จะเกี่ยวข้องอะไรก็ได้ที่ใช้ในการสร้าง (made) ส่วน Builder จะเกี่ยวกับวิธีการที่จะทำสิ่งต่างๆเหล่านั้น ใน Design Pattern Aabstract จะคล้ายกับ Builder ตรงที่มีความซับซ้อนของวัตถุ (Complex Object) มากเหมือนกัน แต่ความแตกต่างของความซับซ้อนนั้น คือ Builder Pattern จะเฉพาะเจาะจงในโครงสร้างของ Object เหล่านั้นเป็นลำดับ และเป็นขั้นเป็นตอนไปเรื่อยๆ (Complex Object Step by Step) ส่วน Abstact Factory จะเน้นที่การรวมกลุ่มของผลที่จะได้รับ (Product) เป็นแต่ละกลุ่ม การ return ค่านั้น Builder จะ return ค่าออกมาจากขั้นสุดท้ายของการทำงาน (Final Step) แต่ Abstract จะคืนค่าออกมาทันทีที่ผ่านแต่ละขั้นตอนโดยไม่ต้องรอถึงขั้นตอนการทำงานสุดท้าย

2.2 การเปรียบเทียบการใช้งาน และความสัมพันธ์ของแต่ละ Pattern

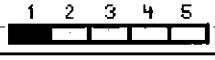
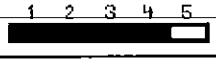
ความถี่ในการใช้งาน

Creational Pattern

No.	Pattern	Level	Frequency of use
1	Abstract		high
2	Builder		medium low
3	Factory		high
4	Prototype		medium
5	Singleton		medium high

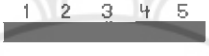
ตารางที่ 2.1 Creational Pattern

Structural Pattern

No.	Pattern	Level	Frequency of use
1	Adapter		medium high
2	Bridge		medium
3	Composite		medium high
4	Decorator		medium
5	Façade		high
6	Flyweight		low
7	Proxy		medium high

ตารางที่ 2.2 Structural Pattern

Behavioral Pattern

No.	Pattern	Level	Frequency of use
1	Chain		medium low
2	Command		medium high
3	Interpreter		low
4	Iterator		high
5	Mediator		medium low
6	Memento		low
7	Observer		high
8	State		medium
9	Strategy		medium high
10	Template		medium
11	Visitor		low

ตารางที่ 2.3 Behavioral Pattern

ในตารางที่ 2.1, 2.2 และ 2.3 จะแสดงถึงความถี่ในการใช้งานของแต่ละ Pattern ซึ่งจะมีตัวเลขไว้ 5 ลำดับของความถี่

2.3 Tool และ ภาษาที่ใช้

2.3.1 ภาษา C#

การเลือกใช้ภาษา C# ในการหา (Detected Pattern) แพทเทอร์น ในโปรแกรมนั้น เนื่องจากว่า ตัวภาษาเองมีความยืดหยุ่น และเหมาะแก่การเขียนโปรแกรมเชิงวัตถุ หรือว่าสามารถถอดแนวความคิด เชิงวัตถุที่เป็น Design Pattern ที่อยู่ในรูป UML มาเป็นเขียนภาษาได้อย่างง่าย เพราะว่า C# ถูกออกแบบมาเพื่อการเขียนโปรแกรมในเชิงวัตถุโดยเฉพาะอยู่แล้ว และการสื่อความหมายต่างๆ ของภาษา ก็มีองค์ประกอบของการทำงานที่มองเห็นเป็นภาพที่สามารถใช้ Model ใดๆ เข้าไปจับและถอดออกมาเป็น Model ต้นแบบได้ ดั้งนั้น Design Pattern ก็เป็นการจำลองถึงภาพเหล่านั้นด้วย จึงเป็นการสะดวกที่เราจะนำภาษา C# มาใช้ในการวิเคราะห์ Code ใดๆ ที่น่าจะมีตัว Design Pattern แฝงอยู่ก็ได้

และตัวโปรแกรมที่เขียน กับ Tool ที่ใช้ในตัวโปรแกรม ก็ใช้ภาษา C# มาสร้าง หากใช้โปรแกรมมาวิเคราะห์ภาษา C# แล้ว ก็น่าจะทำได้สะดวกกว่าการวิเคราะห์ภาษา C# จากโปรแกรมที่เขียนจากภาษาอื่นๆ

2.3.2 เครื่องมือ (Tools) ที่ใช้งาน

Pattern Recognizer

CRORO PAT เครื่องมือที่ช่วยในการจับตัว Design Pattern ตัวหนึ่งที่หลายโปรเจกที่เกี่ยวข้องกับการหา Pattern ต่างๆนิยมใช้กัน แต่ตัวที่ทำหน้าที่สำคัญในการกำหนด Pattern ว่ามีความตรงกันหรือไม่มี 2 อย่างคือ ภาษา RML และ ภาษา RSF ส่วนตัว CROCO PAT ใช้เป็นตัวเปรียบเทียบว่า RML และ RSF มี Pattern ที่ตรงกันหรือไม่

Parser Generator

Parser Generator เป็นเครื่องมือที่จะช่วยในการวิเคราะห์ Code หรือ สร้าง Code ตามที่เราต้องการ เช่นการหา Token การทำการหาชื่อของ Class, Object หรือการสร้างรูปแบบของ Code ที่เราต้องการ

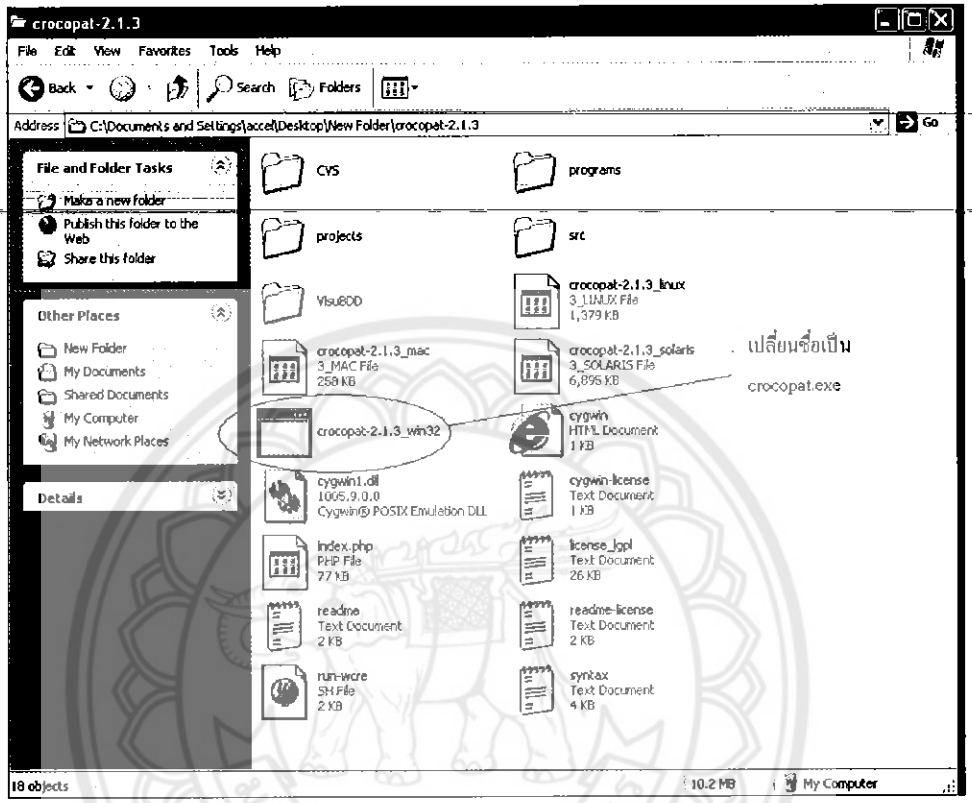
2.4 Pattern Recognizer และการใช้งาน

ในหัวข้อนี้จะอธิบายถึงกรณีศึกษาของ Pattern Composite ที่ใช้เป็นตัวเปรียบเทียบ และการวิเคราะห์ Code ในตัว Program ซึ่งจะจัดอยู่ในประเภทของ Creational Design Pattern

2.4.1 CROCOPAT Tool (Pattern Recognizer Tool)

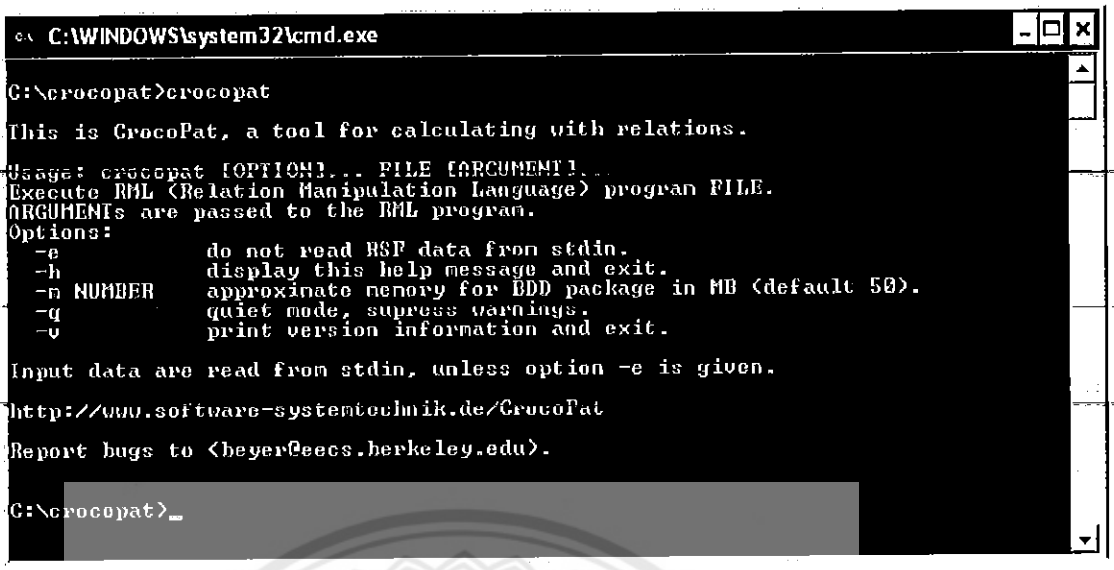
CROCOPAT จะเป็น Tool ชนิด command-line ซึ่งจะมีความคล้ายคลึงกับพวก Unix Tool และ CROCOPAT ยังเป็น Freeware ซึ่งเป็น file Zip มา และสามารถ download

มา จาก http://mtc.epfl.ch/~beyer/CrocoPat/download.html เมื่อ download เสร็จ และแตก file zip ก็จะได้ ดังรูป



รูปที่ 2.4 Program CROCOPAT

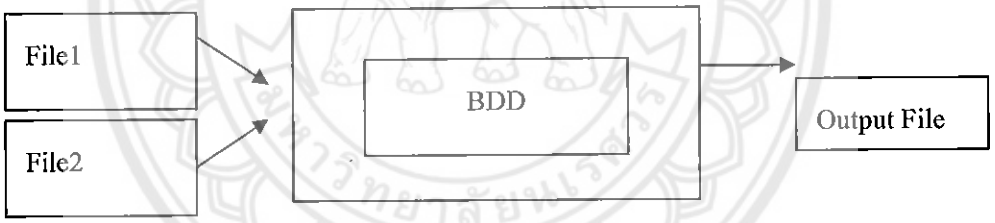
ทำการเปลี่ยนชื่อโปรแกรม crocopat-2.1.3_win32.exe ให้เป็น crocopat.exe เพื่อจะได้ทดสอบได้ง่ายเวลาเรียกทำงานใน Command Mode เมื่อ เรียกใช้ CROCOPAT ใน Command Mode แต่ไม่ใส่ Parameter ใดๆ จะแสดงลักษณะการใช้งาน และข้อมูลอื่นๆที่จำเป็น ดังรูปที่ 3.6



รูปที่ 2.5 การเรียก Program CORCOPAT โดยไม่ใส่ Parameter

Objectives

- Relational computation.

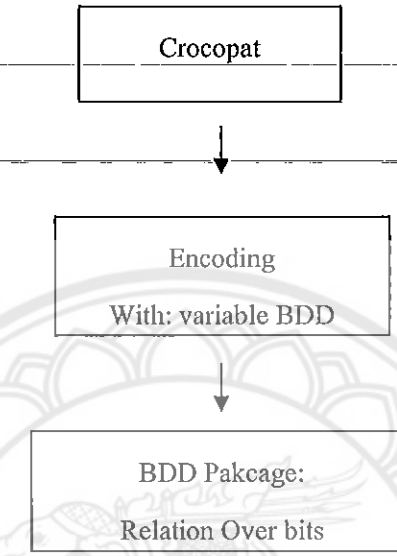


รูปที่ 2.6 การไหลของข้อมูลเบื้องต้นใน Crocopat

ทำงานภายใน Crocopat จะเป็นลักษณะของกราฟที่ขึ้นอยู่กับตัว Input หรือเรียกว่า Arity Binary Relation Graph หรือกราฟที่ไม่ได้ขึ้นกับโครงสร้างใดๆ และสามารถกำหนดความสัมพันธ์ของกราฟเองได้ โดยการทำงานภายในจะใช้การคำนวณแบบ first-order predicate calculus ซึ่งจะประมวลผลข้อมูล Input โดยใช้ โครางสร้างของข้อมูลแบบ Binary Decision Diagram (BDD) ซึ่งโครงสร้างประเภทนี้จะสามารถรวมและตัดความซับซ้อน หรือการทำงานที่ซ้ำๆ กันบางงานออกมาจากความสัมพันธ์ของข้อมูลขนาดใหญ่ได้ Crocopat เป็น Tool ที่สามารถอ่านและเขียนข้อมูลได้ตามต้องการในรูปแบบของภาษา

ที่เขียนในเชิงความสัมพันธ์ของกราฟได้ โดยจะใช้ File 2 File มาเปรียบเทียบกันคือ RML และ RSF File

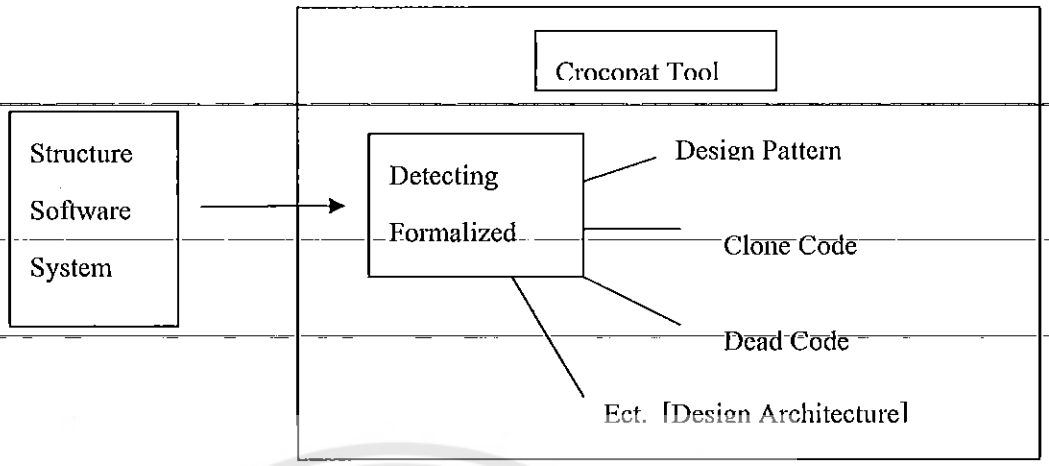
- **High level BDD package**



รูปที่ 2.7 High level BDD package

ใน Crocopat จะใช้ ความสัมพันธ์ผ่าน String ในขณะที่ BDD จะให้ผลในเชิงความสัมพันธ์ของ กราฟ ดังนั้นจะต้องมีการแปลงจาก String มาเป็น Bits โดยใช้ encoder ที่ใช้ตัวแปรทาง BDD

- **Comprehension and assessment of large software systems.**



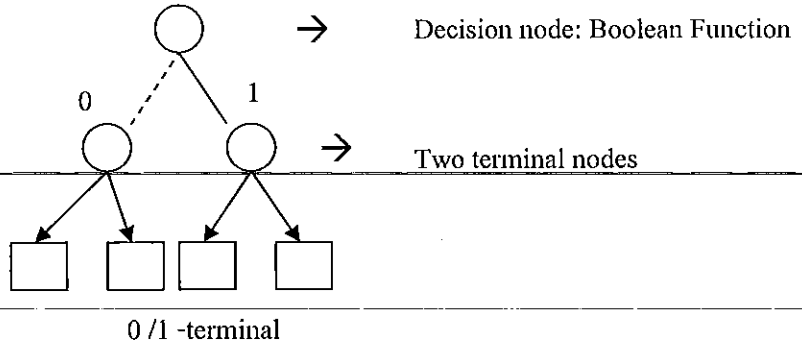
รูปที่ 2.8 รูปแบบการตรวจหาโครงสร้างใดๆของการออกแบบต่างๆ

การวิเคราะห์โครงสร้างโดยทั่วไปของระบบซอฟต์แวร์ จะสามารถวิเคราะห์ผลออกมาในรูปแบบของความสัมพันธ์ทางภาษาที่เป็นในเชิงกราฟ หรือรูปแบบที่กำหนดขึ้นมาเอง เหมือนในรูปที่ 3.9 จะมีทั้งรูปแบบที่เป็น Design Pattern, รูปแบบของการคัดลอกส่วนสำคัญของ Source Code หรือ Code ที่เป็นรูปแบบที่ตายตัวและนำไปใช้งานใหม่ได้ หรือแม้แต่การตรวจหาโครงสร้างใดๆ ที่ต้องการก็ตาม ซึ่ง Tool Crocopat จะตรวจหา Real-Word System หรือ Input Graph Language (RML, RSF)

2.4.2 Binary Decision Diagram (BDD)

BDD จะเหมือนกับ Negative Normal Form (NNF) หรือ คล้ายกับ Propositional Direction Acyclic Graph (PDAG) เป็น Data Structure ที่แสดงBoolean Function ซึ่ง Boolean Function จะแสดง root, directed, acyclic graph ได้ และจะบรรจุ decision node และ 2 terminal nodes (ซึ่งเรียกว่า 0-terminal / 1-terminal)

ในแต่ละ decision node จะบอกถึงตัวแปร Boolean และจะมี 2 child nodes ที่เรียกว่า low / high node ส่วน edge ที่ลากจาก node ถึง low / high child จะถูกกำหนดค่าให้เป็น 0 หรือ 1 แล้วแต่เส้นทางที่กำหนด (edge ที่แทนด้วยค่า จะแสดงด้วยเส้นประ และ ค่า 1 จะแสดงด้วยเส้นทึบ)



รูปที่ 2.9 โครงสร้างอย่างง่ายของ BDD

BDD จะถูกเรียกว่า 'Order' ก็ต่อเมื่อมีตัวแปรที่เกิด Path ที่แตกต่างจากตัวอื่นจาก root และจะเรียก BDD ว่า 'Reduced' เมื่อเป็นไปตามเงื่อนไข 2 ข้อคือ

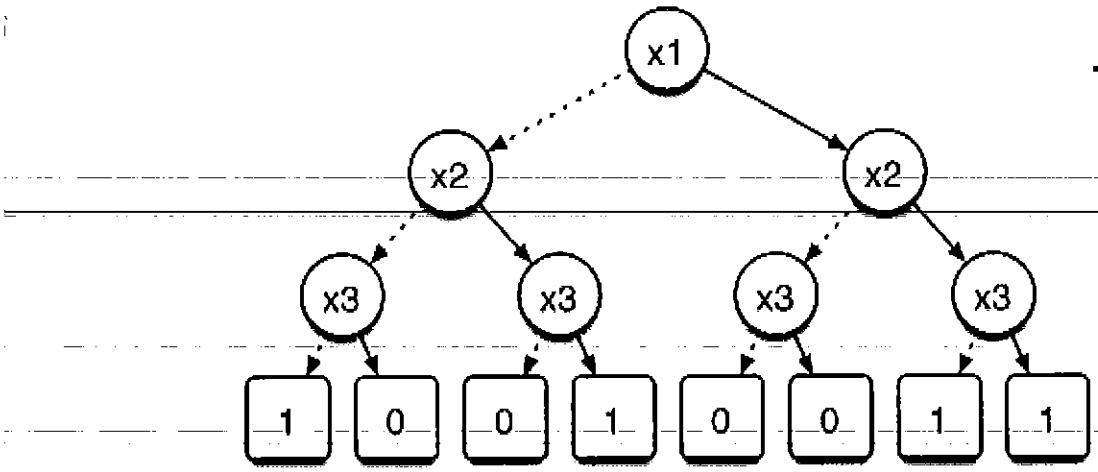
1. มีการรวม (Merge) Sub graph ที่เป็น isomorphic กัน
2. การลด node ซึ่งมี children ที่เป็น isomorphic กัน

โดยทั่วไปการใช้ทฤษฎี BDD จะอ้างอิงถึงการใช้แบบ Reduce Ordered Binary Decision Diagram (ROBDD) การใช้ ROBDD จะสามารถทำ function ที่มีความเป็นเอกเทศ (Unique Function) ได้ ซึ่งคุณสมบัตินี้สามารถนำไปประยุกต์ใช้เป็น function ตรวจสอบกราฟที่มีความคล้ายคลึงกันได้ (isomorphism) หรือใช้ทำ function mapping ได้

ตัวอย่าง

ในรูปที่ 3.11 จะแสดง Binary Decision Tree (ซึ่งจะใช้ reduction rule ไม่ได้) และตารางค่าความจริงของแต่ละค่าของ function f(x1, x2, x3) เส้นประจะแสดงทางเดินที่แทนด้วยค่า 0 และเส้นทึบแสดงทางเดินที่แทนด้วยค่า 1 terminal ที่มีค่า 0 หรือ 1 จะเป็นปลายทางของ Path จาก root ถึง child node สุดท้าย

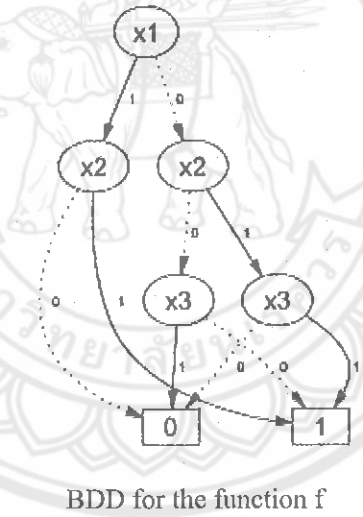
Binary Decision Tree ของ Funrion f นี้สามารถลดทอนได้ดังรูปที่ 3.12



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Binary decision tree and truth table for the function f(x1, x2, x3) = -x1 * -x2 * -x3 + x1 * x2 + x2 * x3

รูปที่ 2.10 BDD ของ function f [10]



BDD for the function f

รูปที่ 2.11 BDD ของ function f จากการแปลงฟแล้ว [10]

Variable ordering

ขนาดของ BDD จะขึ้นอยู่กับ function ที่มีอยู่ในระบบ และการเลือก ordering ของตัวแปร

พิจารณาตัวอย่างของ Boolean function ดังนี้

$f(x_1,...,x_{2n}) = x_1x_2 + x_3x_4 + ... + x_{2n-1}x_{2n}$, ใช้ variable ordering

$x_1 < x_3 < ... < x_{2n-1} < x_2 < x_4 < ... < x2n$, BDD ต้องการ 2^{n+1} nodes ในการแสดง function

ใช้ ordering

$x_1 < x_2 < x_3 < x_4 < \ldots < x_{2n-1} < x_{2n}$, BDD มี 2n nodes.

2.4.3 ภาษาที่ใช้ใน CROCOPAT Tool [Pattern Recognizer]

Tool CROCOPAT นี้คือ ตัว Pattern Recognizer ที่สามารถเปรียบเทียบ Design Pattern ว่าใน Input File (RSF) จะมีตัว Pattern ตัวไหนที่ตรงกับ Compare File (RML) บ้าง โดยในโปรแกรมจะแสดง Composite Pattern เป็นกรณีตัวอย่างจากทั้ง 23 patterns ที่มีอยู่ ในการเปรียบเทียบ Design Pattern จะใช้ Input อยู่ 2 File คือ RML File และ RSF File ในการประมวลผลของ CROCOPAT ซึ่ง RML และ RSF จะมีรายละเอียดดังนี้

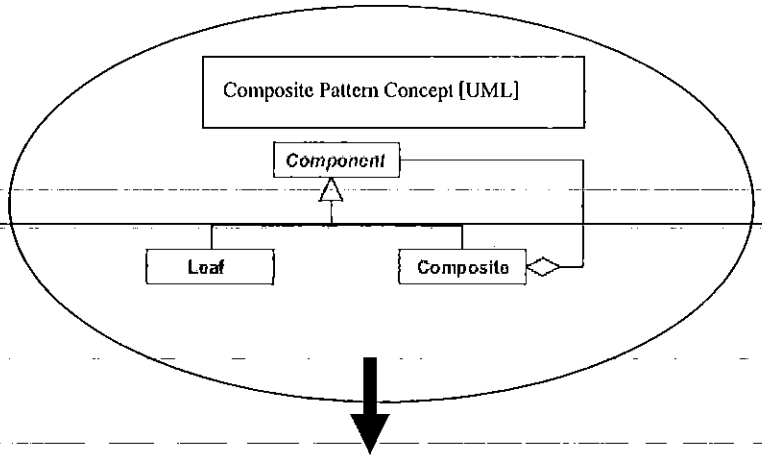
กรณีตัวอย่างใน Program ที่ใช้ Composite Pattern ใน RML และ SRF

Composite Pattern RML

ใน Composite Pattern ที่เขียนขึ้นโดย RML เราจะต้องเข้าใจในส่วนของโครงสร้างของภาษา และการเขียนภาษาของ RML ก่อน และนอกจากนั้นจะต้องมีความเข้าใจในโครงสร้างของ Composite Pattern ด้วยจึงจะสามารถนำแนวความคิดของ Composite Pattern และ ไวยากรณ์ของภาษา RML มาสร้างตัวต้นแบบเป็น RML Composite Pattern ได้

เหตุผลที่จะต้องใช้ RML มาเป็นตัวต้นแบบในการเปรียบเทียบ Pattern ก็คือ ในการทำงานของ CROCOPAT จะใช้ file RML มาเป็นตัวหลักเพื่อนำ file RSF มาเปรียบเทียบว่า มี Pattern ตรงกันหรือไม่ ซึ่งจะแสดงได้ว่าตัว file RML นั้นเราเป็นคนกำหนดขึ้นมาเอง และอ้างอิงโครงสร้างของ Composite เอง ถ้าหากเป็น Pattern อื่นๆ ก็มีลักษณะการสร้างแบบเดียวกัน

หลักในการเขียน RML ออกมาได้อธิบายในบทที่ 2 แล้ว ส่วนในหัวข้อนี้จะมาพิจารณาว่าจะมีการใช้ภาษา RML มาแสดงโครงสร้างของ Composite Pattern ได้อย่างไร



RML

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

รูปที่ 2.12 การคัดลอกรูปแบบของภาษา RML จาก Composite Pattern UML

ในรูปที่ 3.3 จะใช้ Concept UML ของ Composite Pattern มาใช้โดยจะพิจารณาอยู่ 3 Classes คือ Leaf, Component, Composite และความสัมพันธ์ของ Leaf และ Composite Class จะได้รับการสืบทอด (Inheritance) มาจาก class Component จะอยู่ในคำสั่ง

[Inherit(composite, component)] หมายถึง composite สืบทอดจากมาก component

[Inherit(leaf, component)] หมายถึง leaf สืบทอดจากมาก component และในส่วนที่สำคัญอีกส่วน และเป็นส่วนที่บ่งบอกว่าความสัมพันธ์เช่นนี้จะเป็น ความสัมพันธ์ของ Composite Pattern ก็คือ Class Component ถูกบรรจุอยู่ใน Class Composite อีกที

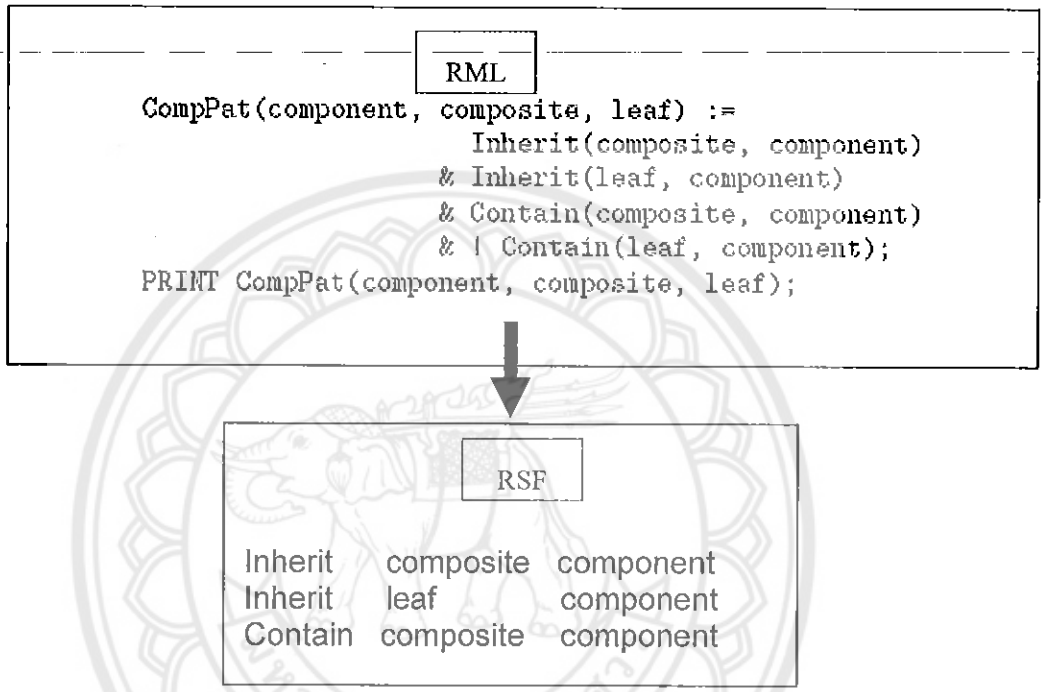
[Contain(composite, component)] หมายถึง composite มี component อยู่ข้างใน

แต่สิ่งที่จะต้องระวังและต้องกำหนดไว้ใน RML อีกอย่างก็คือ Class Leaf ไม่ได้บรรจุ Class Component เหมือนกับ Class Composite จะแสดงได้ในคำสั่ง

[! Contain(leaf, component)] หมายถึง leaf ไม่มี component อยู่ข้างใน

Composite Pattern RSF

RSF File จะเป็น File ที่ได้จากการ Generate มาจากตัวโปรแกรมที่ทำขึ้นมา โดยจะอาศัย Tool ที่ชื่อว่า COCO\R มาช่วย ซึ่งจะอธิบายในหัวข้อถัดไป และภาษา RSF ที่เป็น Composite Pattern จะต้องสร้างโดยอาศัย RML File เป็นฐานที่จะทำการเขียนขึ้นมา กล่าวคือ เราจะมี File RML ที่ทำไว้เป็น Pattern อยู่แล้วตามภาษา RML



รูปที่ 2.13 การคัดลอกรูปแบบของภาษา RML เป็นรูปแบบของภาษา RSF

ในรูปที่ 3.10 ในส่วนของ RSF เราสมมุติให้ความสัมพันธ์ของ 3 Class (A, B, C) เป็นแบบ Composite Pattern และสามารถอธิบายตามโครงสร้างของ RML ต้นแบบคือ

Component แทนด้วย Class C

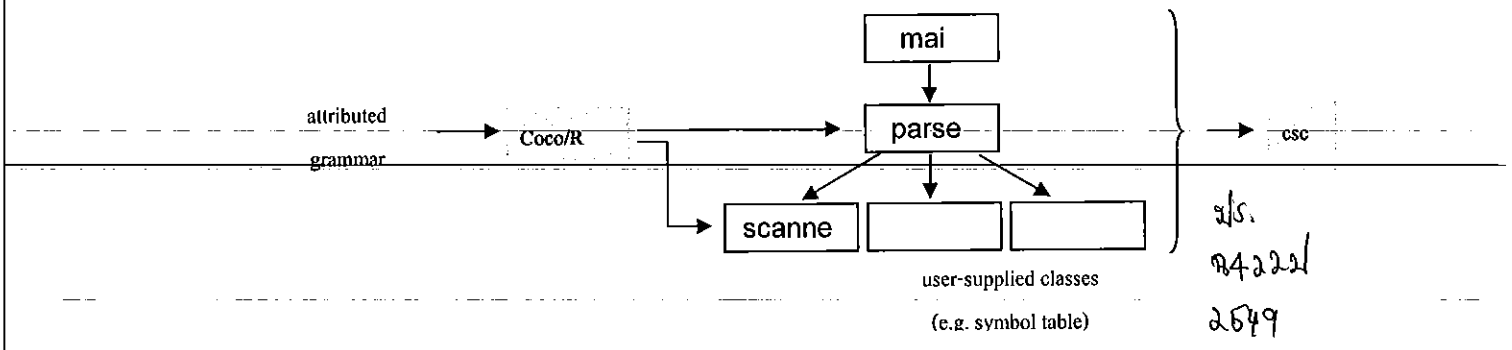
Leaf แทนด้วย Class B

Composite แทนด้วย Class A

แต่ไม่จำเป็นต้องบอกว่า C ไม่ได้อยู่ใน B เหมือน A

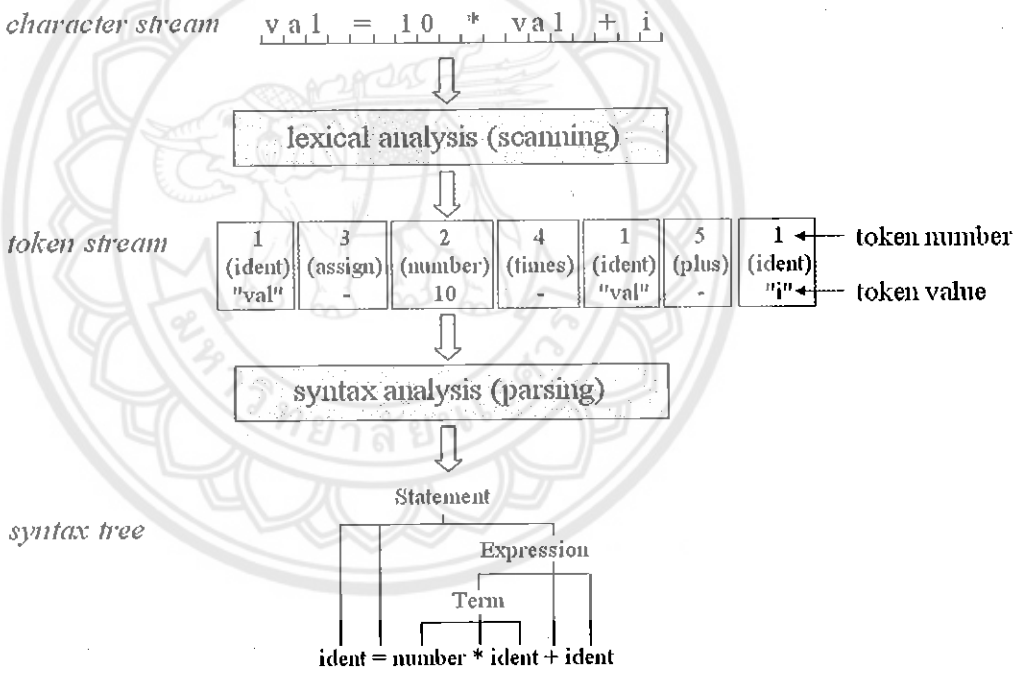
2.5 การศึกษา COCO\R (C# Parser Generator for C#) และการใช้งาน

2.5.1 Structure

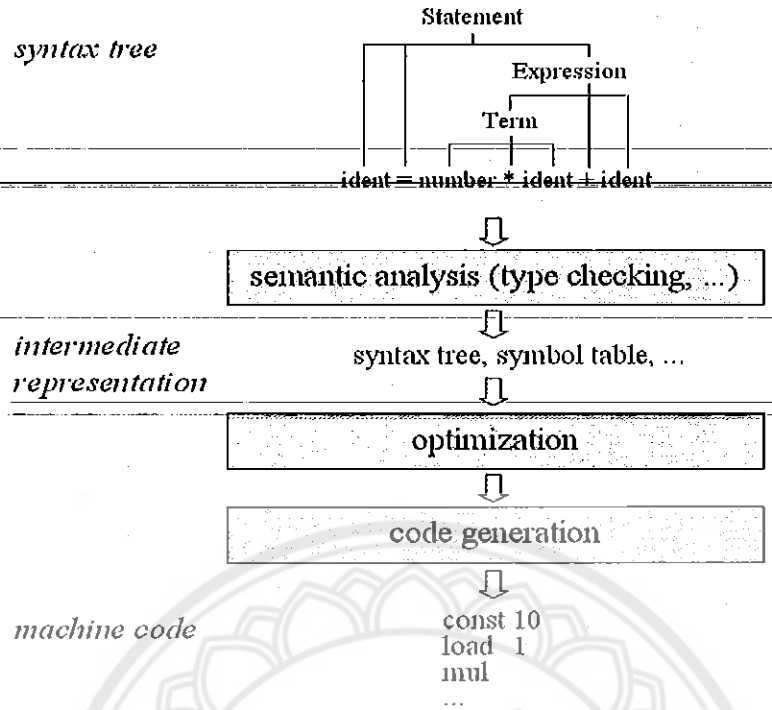


รูปที่ 2.14 การทำงานของ Coco/R [12]

2.5.2 Scanner and Parser



รูปที่ 2.15 การทำงานของ Scanner และ Parser [12]

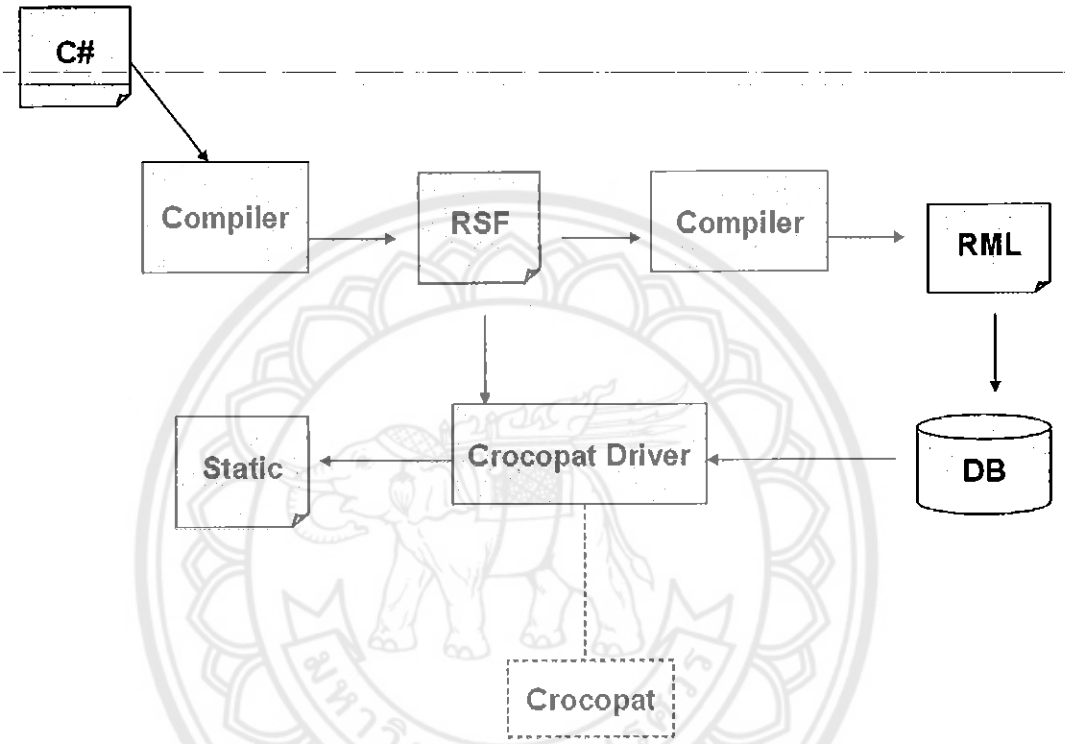


รูปที่ 2.16 การนำ Syntax Tree ออกมาเป็น Output [12]

ตามรูปที่3.19 และ 3.20 แสดงขั้นตอนของการทำงานในการ Scan Character และ Parser Token Steam จนได้เป็นภาษาที่มีความสัมพันธ์ในเชิง Syntax Tree แล้วก็นำมา Check แล้ว Generate Output ออกมาตามต้องการ

บทที่ 3
วิธีการดำเนินงาน

3.1 โครงสร้างของตัวโปรแกรม



รูปที่ 3.1 ลำดับการทำงานโปรแกรม

ส่วนประกอบของโปรแกรม

3.1.1 C# to RSF Compiler

เป็นส่วนที่แปลงโค้ดโปรแกรมภาษา C# มาเป็นไฟล์ RSF ซึ่งโค้ดโปรแกรมในส่วนนี้ได้ใช้ Tool ที่ใช้ในการสร้าง Compiler คือ Coco-R ช่วยการสร้างโค้ดโดยโปรแกรม Coco-R รับอินพุตเป็นไฟล์นามสกุล ATG ซึ่งเป็นไฟล์ที่เก็บรูปแบบไวยกรณ์ของภาษาที่ต้องการจะสร้าง Compiler ขึ้นซึ่งเมื่อทำการรัน Coco-R จะทำให้ได้โค้ดโปรแกรมเป็นภาษา C# ทั้งหมด 2 ไฟล์ ประกอบด้วย

1.1 Scanner ทำหน้าที่ในการอ่านซ้อสโค้ด C# เข้ามาและทำการ จัดแบ่งกลุ่มตัวอักษรเป็น Token ซึ่ง Token หนึ่งจะเก็บค่ากลุ่มคำหรืออักขระที่มีความหมายในโค้ดภาษา C#

โดยก่อนที่จะทำการแบ่งเป็น Token ได้ทำการตัดส่วนที่เป็น Comment ของโค้ดโปรแกรมออกแล้วโดยข้อมูลที่ได้จากส่วนนี้จะเก็นไว้เป็น List ของ Token เพื่อนำไปใช้ในการวิเคราะห์ต่อไป

1.2 Parser เป็นส่วนที่ทำหน้าที่ในการตรวจไวยกรณ์ของภาษา C# ว่าถูกต้องตามไวยกรณ์โดยทำการวนลูปตรวจสอบไปทีละ Token และตรวจสอบว่าถูกต้องตามไวยกรณ์หรือไม่หากพบที่ไวยกรณ์ผิด ก็แจ้งบรรทัดที่ผิดออกมาถ้าตรวจสอบก็จะแจ้งว่าไวยกรณ์ถูกต้องโดยจะแจ้งจำนวนข้อผิดพลาดเท่ากับ 0

1.3 RSFDetector เป็นส่วนที่ทำหน้าที่ตรวจหาโครงสร้างของโปรแกรมเพื่อนำมาสร้างเป็นไฟล์ RSF ต่อไปการทำงานได้มาจากไฟล์ Parser.cs มาทำการดัดแปลงโดยส่วนนี้ก็จะทำการวนลูปตรวจสอบหาโครงสร้างองค์ประกอบต่างๆ ของโค้ดโปรแกรมเหมือนในการทำ Parser และจะทำการเก็บข้อมูลในส่วนที่เกี่ยวกับความสัมพันธ์ของของ Class ไว้โดยในโค้ดโปรแกรมภาษา C# จะมีความสัมพันธ์ที่ตรวจสอบ 3 อย่างคือ Inheritance, containment และ Implement โดยค่าที่ได้จะเก็บไว้ในลิสท์ของข้อมูลโดยโครงสร้างของลิสท์ประกอบด้วย Class Name, list Inheritance, list containment และ list Implement นับเป็นหนึ่ง Node และนำข้อมูลที่ได้ไปเขียนเป็นไฟล์ RSF ต่อไป

3.1.2 RSF to RML Compiler

เป็นส่วนที่ทำการแปลงความสัมพันธ์ทั้งหมดที่มีในโค้ดโปรแกรมภาษา C# หรือ RSF มาเป็น RML ซึ่งเป็นความสัมพันธ์ย่อยๆ ที่มีอยู่ในโปรแกรมซึ่ง Compiler ในส่วนนี้จะทำการอ่านข้อมูลมาจากลิสท์ข้อมูลของความสัมพันธ์ระหว่าง Class ทั้งหมดหรือคือข้อมูลที่ใช้ในการสร้างไฟล์ RSF นำมาทำการวิเคราะห์โครงสร้างของความสัมพันธ์ระหว่าง Class ทั้งหมดโดยแบ่งออกมาเป็นความสัมพันธ์ย่อยๆ นี้ทำการแบ่งโดยใช้ความสัมพันธ์แบบ Inheritance และ containment เป็นหลักโดยหา Class ที่มีความสัมพันธ์ดังกล่าวต่อกันทั้งหมดเป็นโครงสร้างความสัมพันธ์ย่อยๆ ต่อกันเพื่อนำมาสร้างมาเป็น RMLแล้วเก็บไว้ในฐานข้อมูลต่อไปดังนั้นในหนึ่ง RSF จึงสามารถหาได้หลายๆ RML

3.1.3 Database

ใช้เก็บฐานข้อมูลเป็นรูปแบบของไฟล์โดยจะเก็บรวบรวมไฟล์ RML ไว้เพื่อใช้ในการรันเปรียบเทียบกับไฟล์ RSF ต่อไปและเก็บเป็นฐานข้อมูลของโครงสร้างที่พบในการรันโปรแกรม

3.1.4 Crocopat Driver

เป็นส่วนที่ใช้ในการสั่งรันโปรแกรม Crocopat เพื่อใช้ในการตรวจสอบระหว่าง RSF กับ RML ได้ค่าเป็นสถิติจำนวนรูปแบบความสัมพันธ์ที่มีใน RML ที่มีใน RSF นั่นเอง โดยตัวที่ใช้ในการรันจะเขียนเป็น Bath file ที่มีรูปแบบคำสั่งดังนี้

รูปที่ 3.2 Command Window Running Crocopat tool

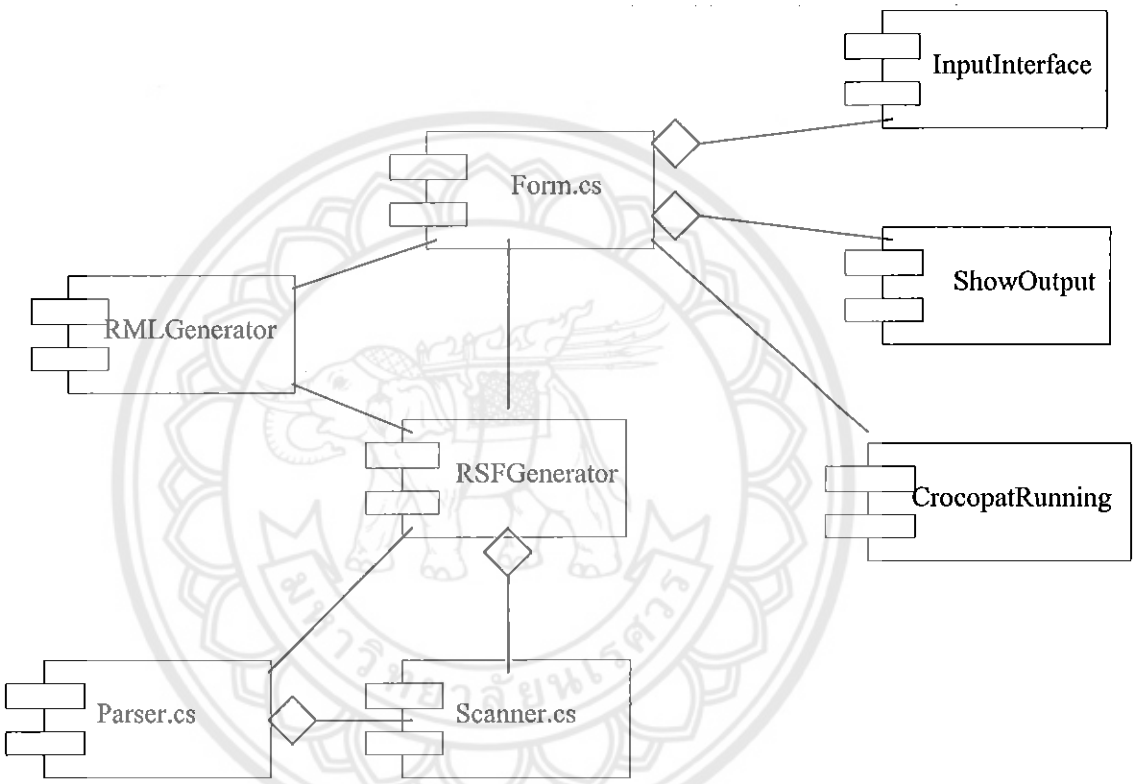
```
crocopat\crocopat Patern1.rml < test.rsf > output.txt
echo . >> output.txt
crocopat\crocopat Patern2.rml < test.rsf >> output.txt
```

ผลลัพท์ที่ได้จารการรันจะได้เป็นไฟล์ output.txt ซึ่งจะทำการเก็บเอาท์พุตที่ได้จารการรันโปรแกรมเอาไว้และในการนำเอาท์พุตมาแสดงก็จะทำการอ่านค่ามาจากไฟล์ output.txt

3.2 การออกแบบโปรแกรม

3.2.1 Component Diagram

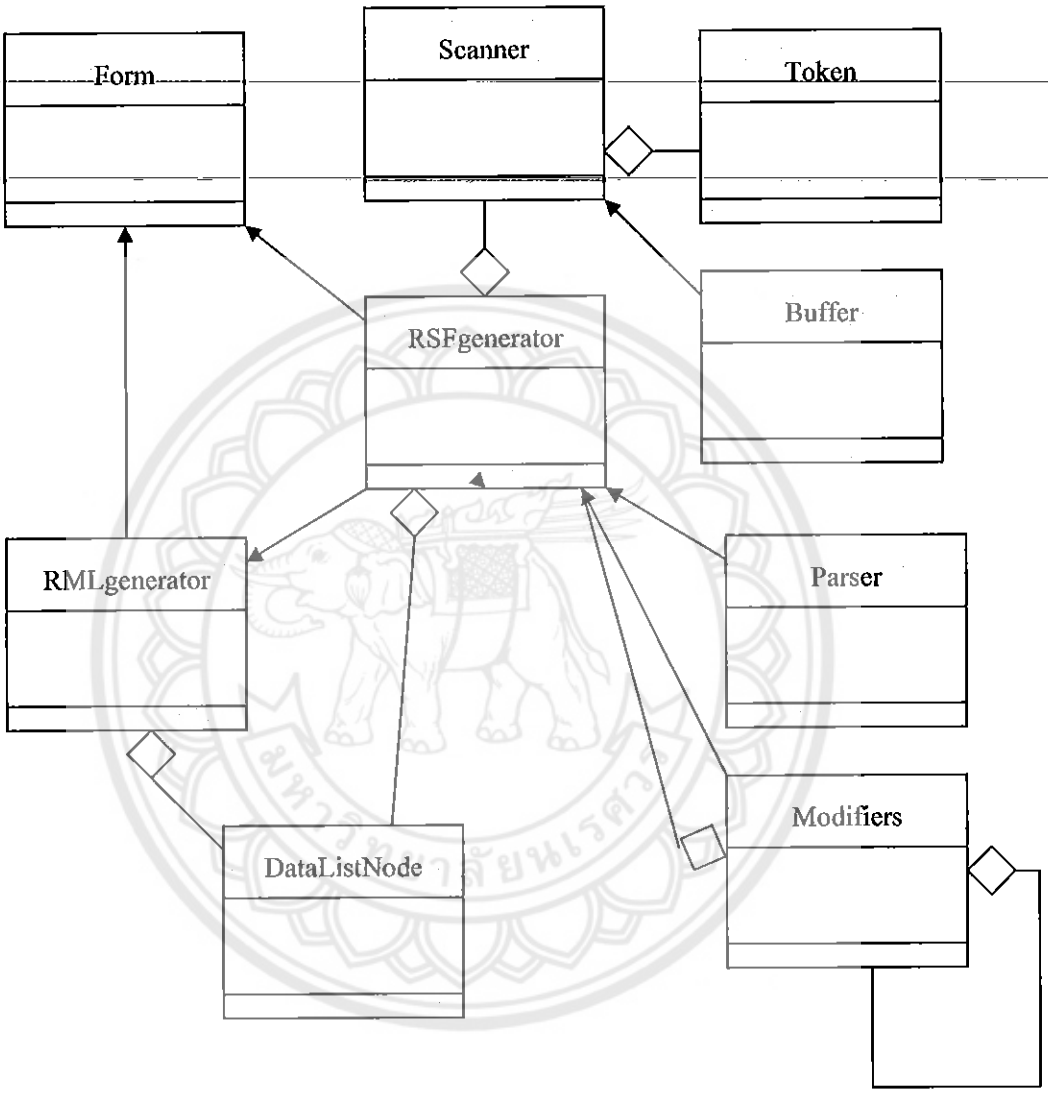
เขียน Component Diagram เพื่อใช้จำลองลักษณะทางกายภาพของ object-oriented system ของโปรแกรม Design pattern mining program โดยจะแสดงให้เห็นถึงส่วนประกอบทาง software component ต่าง ๆ ของระบบของโปรแกรม รวมถึงความสัมพันธ์ระหว่าง component ต่าง ๆ ภายในโปรแกรมได้ดังนี้



รูปที่ 3.3 Component Diagram

3.2.2 Class Diagram

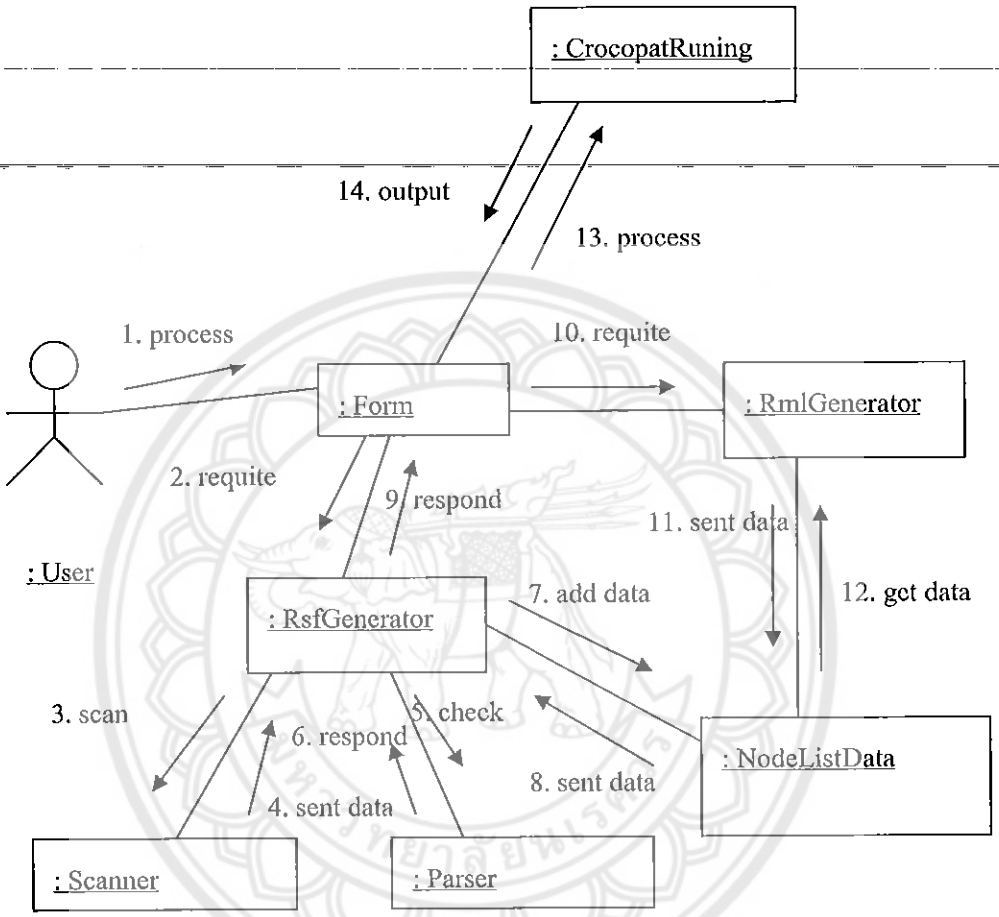
เขียน Class diagram เพื่อใช้แสดง class และความสัมพันธ์ (relationship) ระหว่าง class ของโปรแกรมได้ดังนี้



รูปที่ 3.4 Class Diagram

3.2.3 Collaboration Diagram

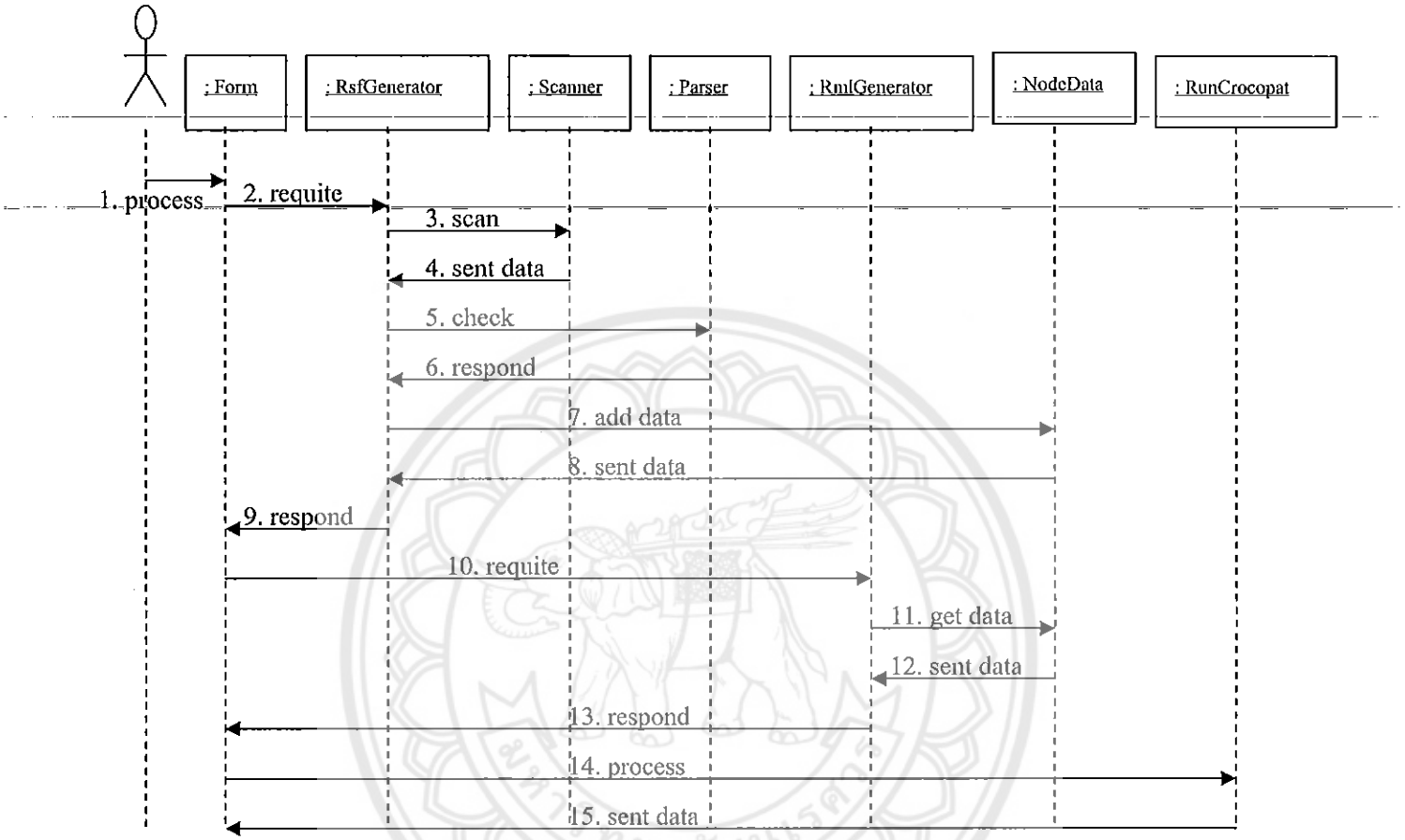
เขียน Collaboration Diagram เพื่อแสดงการติดต่อสื่อสารระหว่างออบเจ็กต์ต่างๆ และความสัมพันธ์ระหว่างที่แต่ละออบเจ็กต์ติดต่อสื่อสารกันของโปรแกรมได้ดังนี้



รูปที่ 3.5 Collaboration Diagram

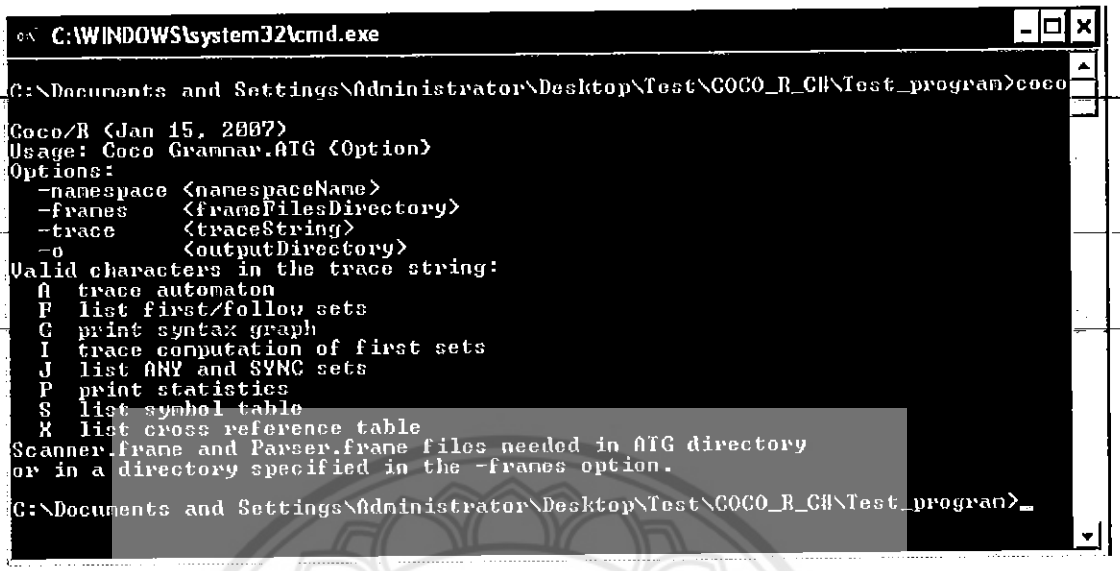
3.2.4 Sequence Diagram

เขียน Sequence Diagram เพื่อแสดงการทำงานของออบเจ็กต์ต่างๆ โดยลำดับการทำงานโดยการส่ง massage ระหว่าง ออบเจ็กต์เมื่อเกิดเหตุการณ์ต่างๆ ได้ดังนี้



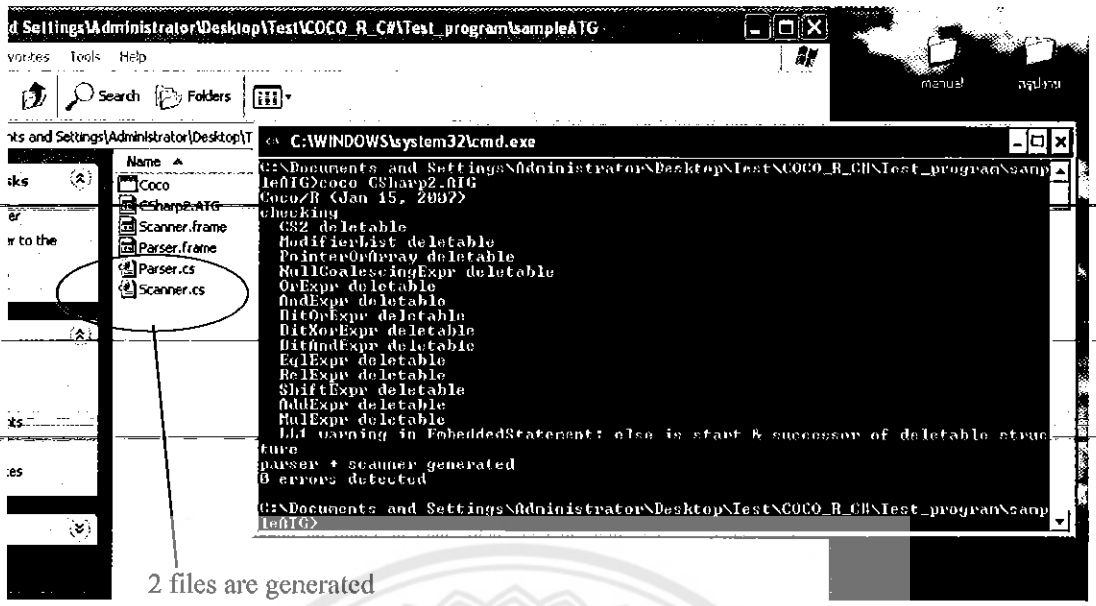
รูปที่ 3.6 Sequence Diagram

3.3 Generate Code ของ Paser และ Scanner



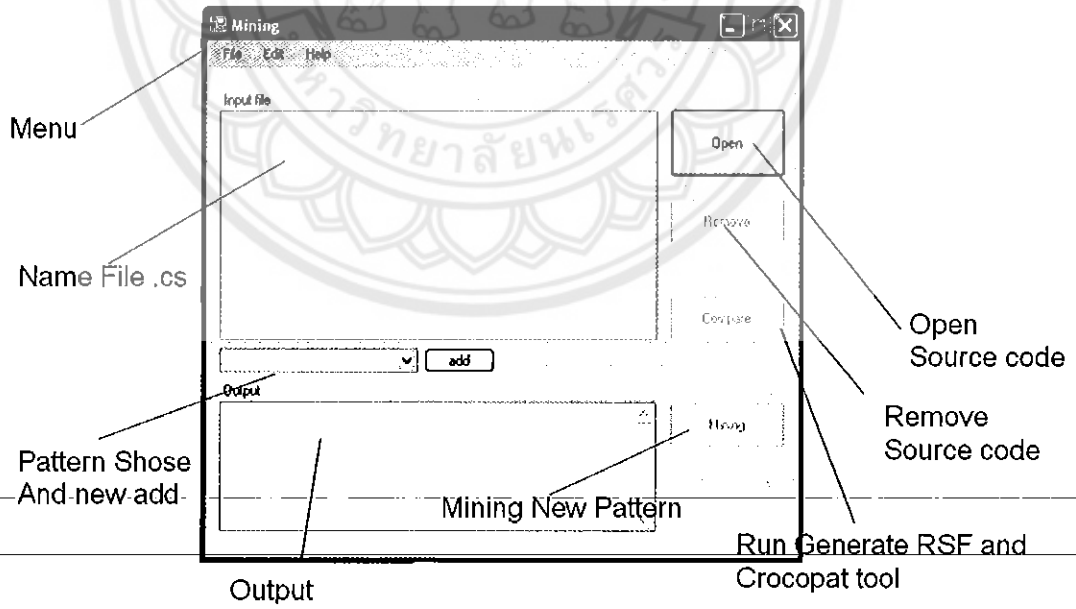
รูปที่ 3.7 การทำงานบน Command Mode

Coco\R เมื่อเราทำการ Run โดยไม่ใส่ Parameter ตัวโปรแกรมจะแสดงวิธีการใช้งานออกมา คือจะต้องมี File ATG เป็น Paramerter ในการ Run เมื่อทำการ Run โดยใส่ CCharp2.ATG ลงไปด้วยก็จะได้ File Parser.cs และ Scananer.cs ที่ได้จากการ Generate จากตัวโปรแกรม Coco.exe และจะต้องใช้ Parser.frame และ Scanner.frame ในการช่วย Generate Code ของ Parser.cs และ Scanner.cs ออกมาด้วย ดังในรูปที่ 3.7



รูปที่ 3.8 การ Genrate File Scanner และ Parser ของ C#

รูปหน้าต่างของ Interface ที่ประกอบขึ้นจากการออกแบบข้างต้น



รูปที่ 3.9 หน้าต่างของโปรแกรม

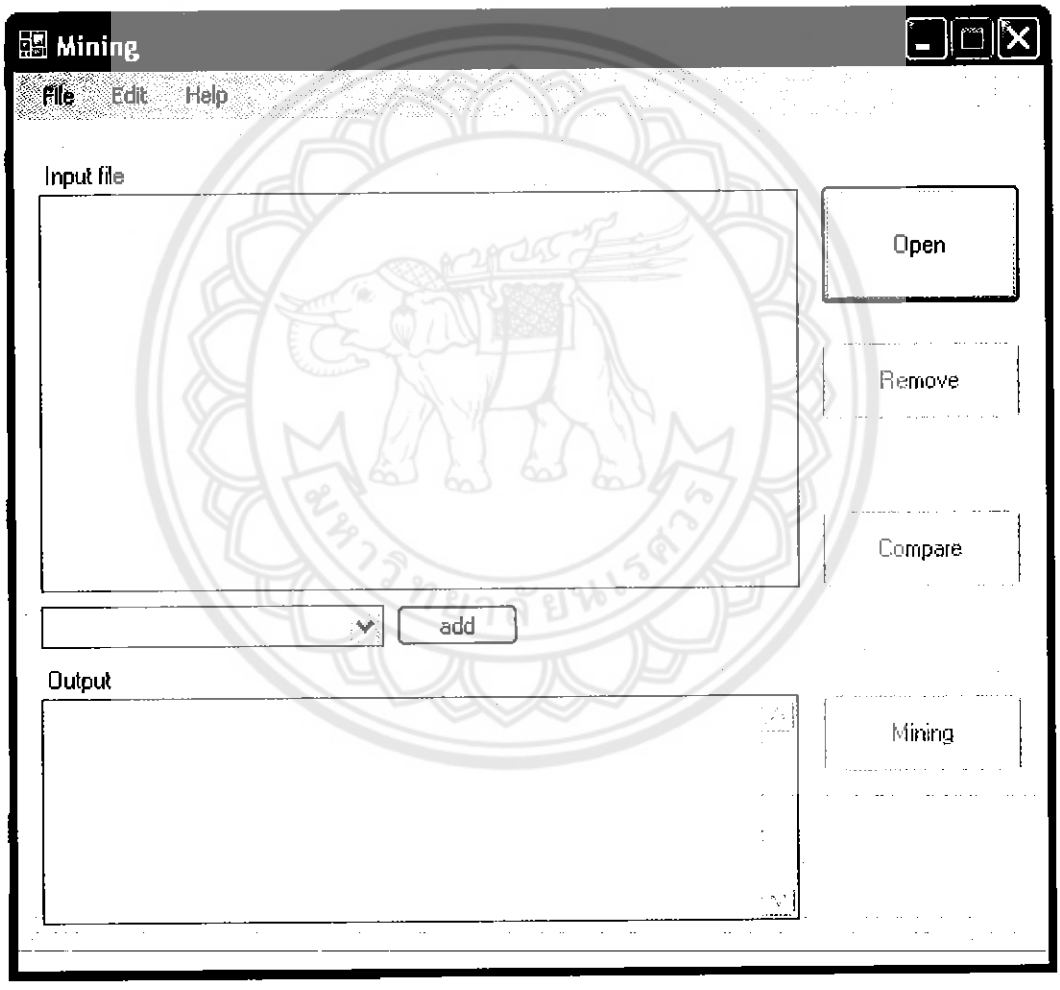
บทที่ 4

ผลการทดลอง

4.1 การใช้งานโปรแกรม

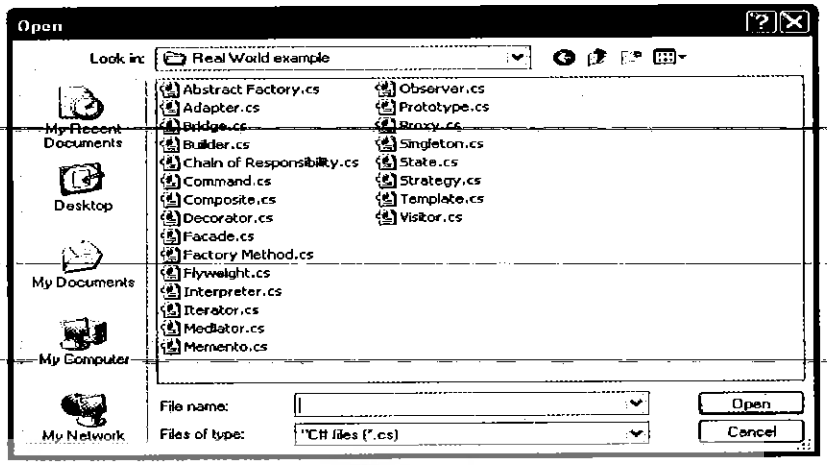
วิธีใช้โปรแกรม

1.เปิดโปรแกรม



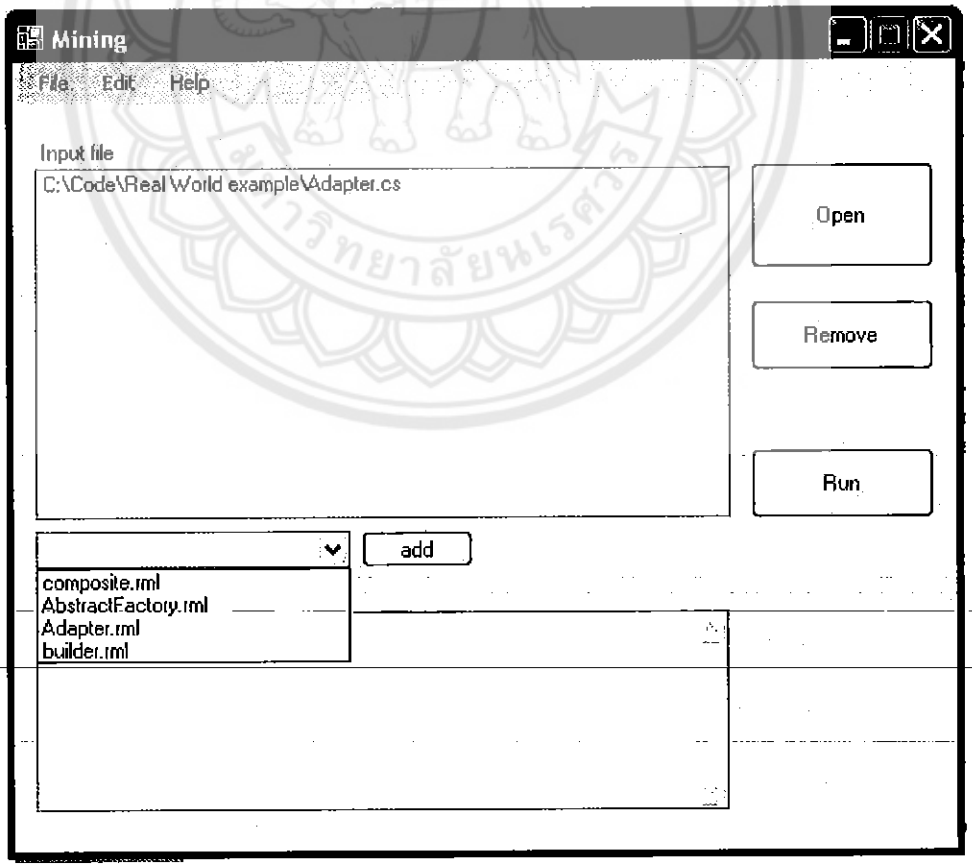
รูปที่ 4.1 การเปิดโปรแกรม

2. คลิก Open เลือกไฟล์ซอร์สโค้ด C#



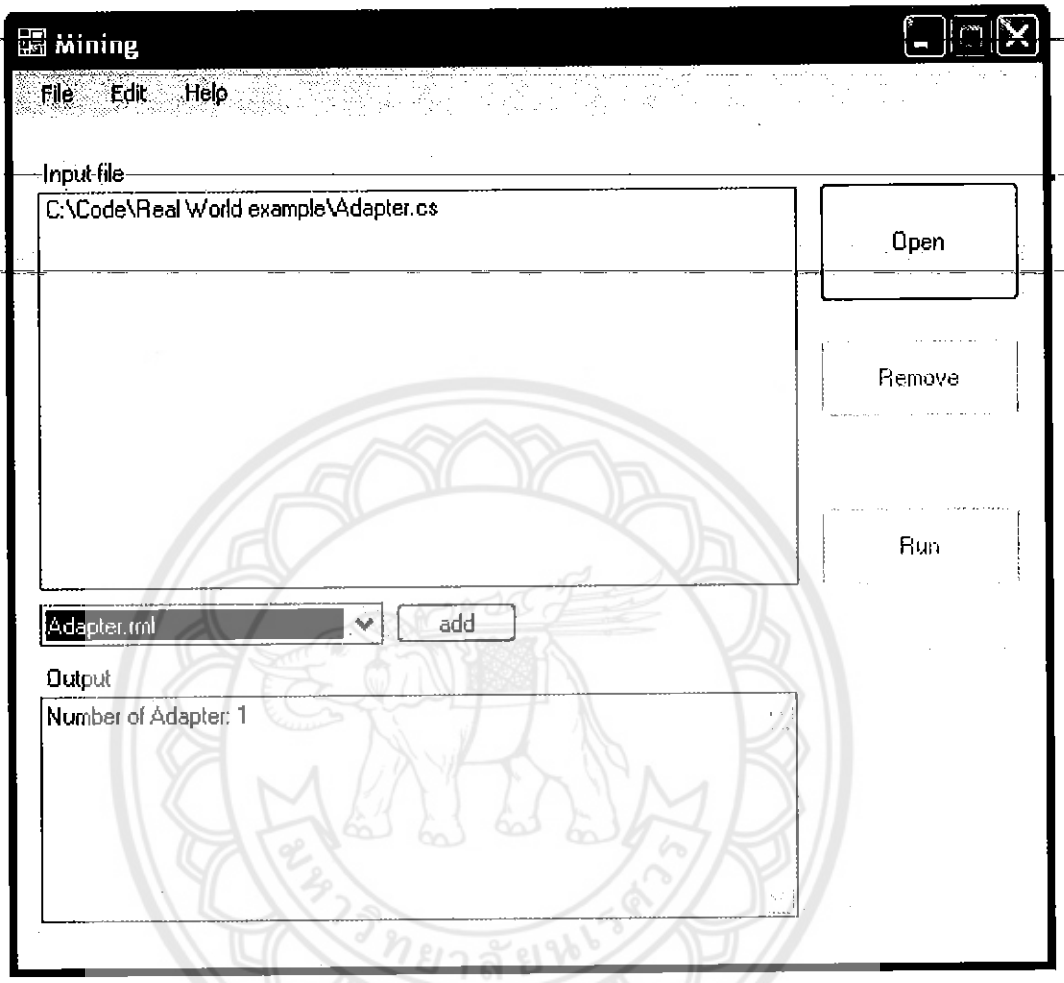
รูปที่ 4.2 การเลือก Code C#

3. เลือก pattern ที่ต้องการตรวจสอบจาก list ด้านล่าง หากต้องการเพิ่ม pattern ใน list ทำได้โดยคลิก add แล้วเลือกไฟล์ rml ที่ต้องการ



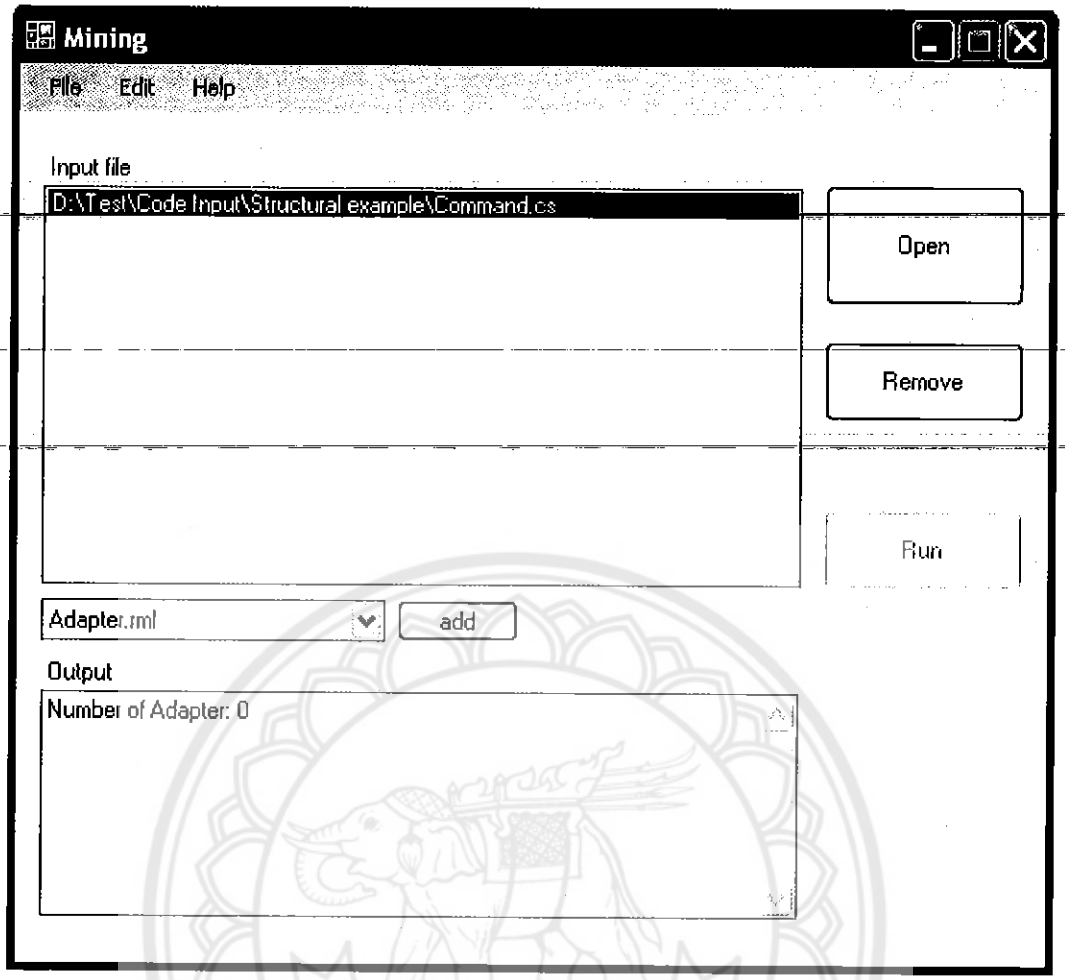
รูปที่ 4.3 การเลือก Code C#ในโปรแกรม

4. คลิก run เพื่อเริ่มทำงานได้ผลลัพธ์แสดงจำนวน pattern ที่เราตรวจสอบที่มีในโค้ดโปรแกรม ถ้าไม่มี pattern ในโค้ดโปรแกรมจะบอกเป็นจำนวน 0



รูปที่ 4.4 การ Run Program

ใน Pattern อื่นๆ ก็สามารถเลือก File RML เข้ามาโดยกดปุ่ม Add การเพิ่ม File RML เข้ามา หากมีการ Run Source Code ที่ทำการเปิดมา แล้วไม่พบ Pattern ใน ช่อง Output ก็จะแสดงดังรูปที่ 3.28



รูปที่ 4.5 Output เมื่อไม่พบ Pattern

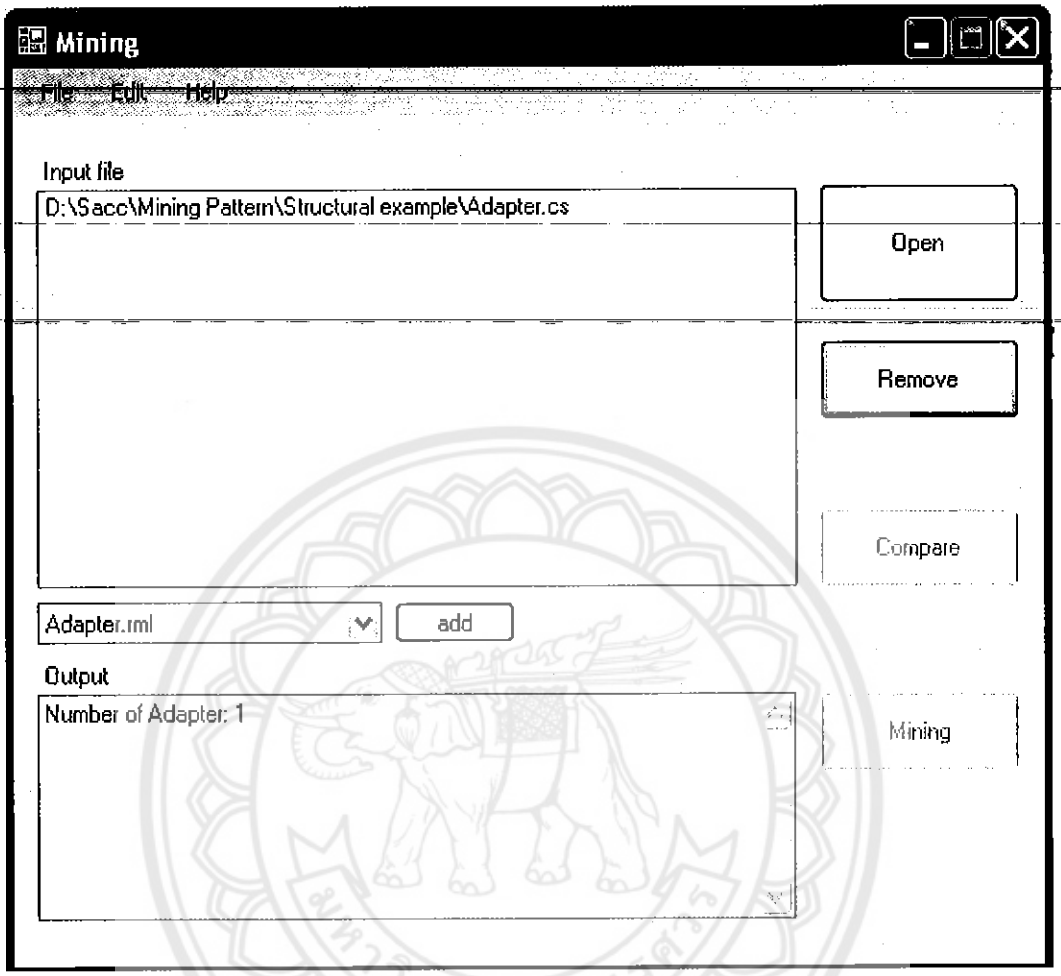
ใน Pattern อื่นๆ ผลของการ Run ก็จะเป็นในลักษณะที่เป็นอย่างตัวอย่างในรูปข้างต้น และสามารถทำการ Remove เพื่อนำ Source Code ตัวใหม่ที่สงสัยว่ามี Pattern ที่ต้องการตรวจสอบเพิ่มเข้ามาใหม่ได้ และก็งานในลักษณะเดียวกัน

4.2 Exam Pattern [Adapter pattern]

```
//Structural code
// Adapter pattern -- Structural example

using System;

namespace DoFactory.GangOfFour.Adapter.Structural
{

 // Mainapp test application

 class MainApp
 {
  static void Main()
  {
   // Create adapter and place a request
   Target target = new Adapter();
   target.Request();

   // Wait for user
   Console.Read();
  }
 }

 // "Target"

 class Target
 {
  public virtual void Request()
  {
   Console.WriteLine("Called Target Request()");
```

```
    }
  }

  // "Adapter"

  class Adapter : Target
  {
    private Adaptee adaptee = new Adaptee();

    public override void Request()
    {
      // Possibly do some other work
      // and then call SpecificRequest
      adaptee.SpecificRequest();
    }
  }

  // "Adaptee"

  class Adaptee
  {
    public void SpecificRequest()
    {
      Console.WriteLine("Called SpecificRequest()");
    }
  }
}
```

ผลการรันโปรแกรม



รูปที่ 4.6 Output Adapter Pattern

Real-world code Adapter pattern

Compound	เป็น	Target
RichCompound	เป็น	Adapter
ChemicalDatabank	เป็น	Adaptee
AdapterApp	เป็น	Client

```
---------------------------------------------------
// Adapter pattern -- Real World example

using System;

namespace DoFactory.GangOfFour.Adapter.RealWorld
{

 // MainApp test application

 class MainApp
 {
  static void Main()
  {
   // Non-adapted chemical compound
   Compound stuff = new Compound("Unknown");
   stuff.Display();

   // Adapted chemical compounds
   Compound water = new RichCompound("Water");
   water.Display();

   Compound benzene = new RichCompound("Benzene");
   benzene.Display();

   Compound alcohol = new RichCompound("Alcohol");
```

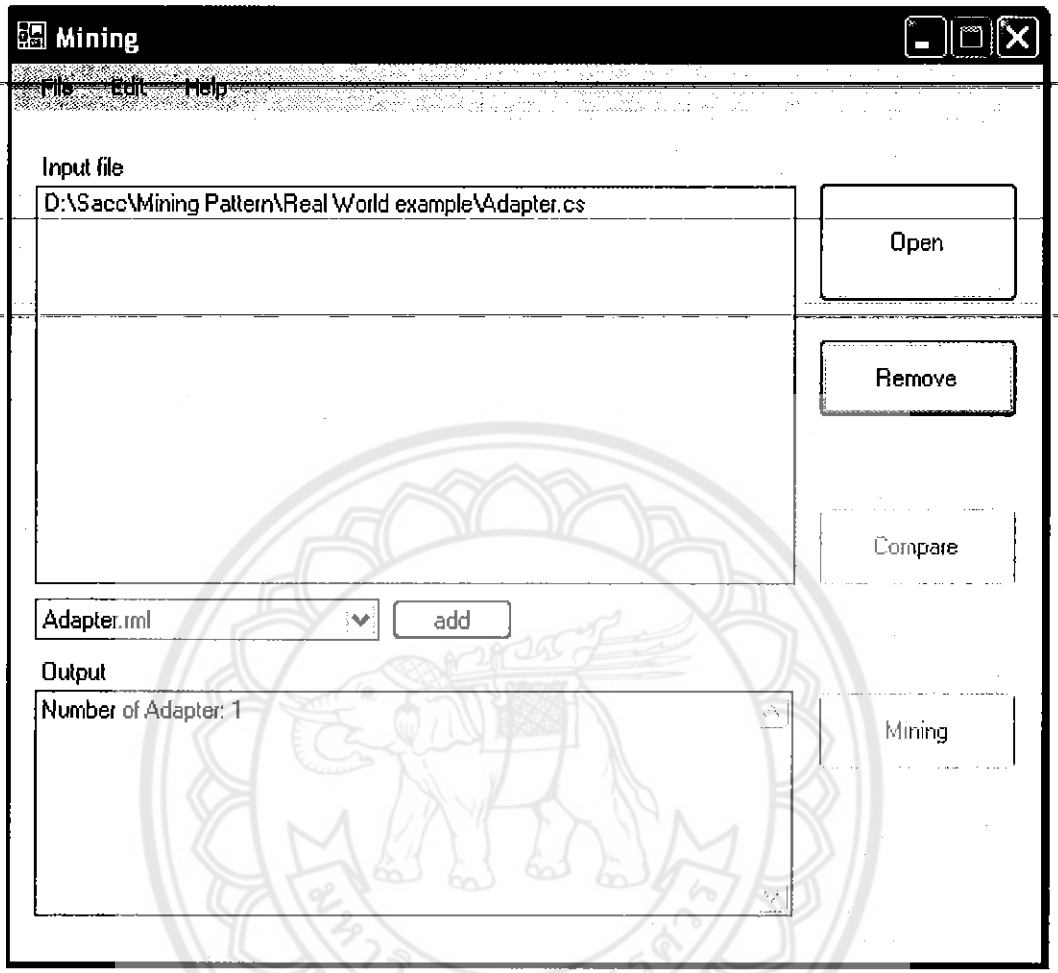
```
            alcohol.Display();

            // Wait for user
            Console.Read();
        }
    }

    // "Target"

    class Compound
    {
      protected string name;
      protected float boilingPoint;
      protected float meltingPoint;
      protected double molecularWeight;
      protected string molecularFormula;

      // Constructor
      public Compound(string name)
      {
        this.name = name;
      }

      public virtual void Display()
      {
        Console.WriteLine("\nCompound: {0} ------ ", name);
      }
    }

    // "Adapter"

    class RichCompound : Compound
```

```
{
 private ChemicalDatabank bank;

 // Constructor
 public RichCompound(string name) : base(name)
 {
 }

 public override void Display()
 {
  // Adaptee
  bank = new ChemicalDatabank();
  boilingPoint = bank.GetCriticalPoint(name, "B");
  meltingPoint = bank.GetCriticalPoint(name, "M");
  molecularWeight = bank.GetMolecularWeight(name);
  molecularFormula = bank.GetMolecularStructure(name);

  base.Display();
  Console.WriteLine(" Formula: {0}", molecularFormula);
  Console.WriteLine(" Weight : {0}", molecularWeight);
  Console.WriteLine(" Melting Pt: {0}", meltingPoint);
  Console.WriteLine(" Boiling Pt: {0}", boilingPoint);
 }
}

// "Adaptee"

class ChemicalDatabank
{
 // The Databank 'legacy API'
 public float GetCriticalPoint(string compound, string point)
 {
```

```
        float temperature = 0.0F;

        // Melting Point
        if (point == "M")
        {
         switch (compound.ToLower())
         {
          case "water" : temperature = 0.0F; break;
          case "benzene" : temperature = 5.5F; break;
          case "alcohol" : temperature = -114.1F; break;
         }
        }
        // Boiling Point
        else
        {
         switch (compound.ToLower())
         {
          case "water" : temperature = 100.0F; break;
          case "benzene" : temperature = 80.1F; break;
          case "alcohol" : temperature = 78.3F; break;
         }
        }
        return temperature;
       }

       public string GetMolecularStructure(string compound)
       {
        string structure = "";

        switch (compound.ToLower())
        {
         case "water" : structure = "H20"; break;
```

```
            case "benzene" : structure = "C6H6"; break;
            case "alcohol" : structure = "C2H6O2"; break;
          }
          return structure;
        }

        public double GetMolecularWeight(string compound)
        {
          double weight = 0.0;
          switch (compound.ToLower())
          {
            case "water" : weight = 18.015; break;
            case "benzene" : weight = 78.1134; break;
            case "alcohol" : weight = 46.0688; break;
          }
          return weight;
        }
      }
    }
```

ผลการรันโปรแกรม



รูปที่ 4.7 Output Adapter Pattern 2 (Real-world code)

ใน Pattern ที่เหลืออีก 7 Pattern ที่ Program สามารถตรวจสอบได้ก็จะทำงาน และมี Output ออกมาในลักษณะคล้ายกัน ในช่อง Output จะแสดงออกมาว่า Source Code ที่นำมาทดสอบมี Pattern นั้นๆที่ทำการเลือกแล้วกี่ตัว

4.3 ผลการทดลองใน Code ที่มี Standard Pattern

ทดลองใช้โปรแกรมในการตรวจสอบโค้ดโปรแกรมจากโค้ดโปรแกรมตัวอย่างของ Microsoft MSDN 101 C# Samples เพื่อตรวจสองหา Desing Pattern ที่มีในโค้ดโปรแกรม

Program	Composite	Builder	Adapter	Abstract factory	Facade	Mediator	ChainofRe	Proxy
Advanced - Multithreading - How-To Async Calls	0	0	1	0	2	0	0	1
Advanced - Serialization - How-To Serializing Objects	0	0	1	0	2	0	0	1
Advanced .NET Framework - Interacting Windows Service	1	0	1	1	2	0	0	0
Advanced .NET Framework - Make WIn32 API Calls	0	0	1	0	2	0	0	1
Advanced .NET Framework (GDI+) - Animation with GDI+	0	0	1	0	2	0	0	1
Advanced .NET Framework (GDI+) - Create a Screensaver with GDI+	1	0	1	1	4	0	0	2
Advanced .NET Framework (GDI+) - Use GDI+ to manipulate images	0	0	1	0	2	0	0	1
Advanced .NET Framework (GDI+) - Working with GDI+ Brushes	0	0	1	0	2	0	0	1
Windows Forms - How-To Data Binding with Navigation	1	1	4	4	5	0	0	3
Data Access - Bind Data in a Combo Box	0	0	1	0	2	0	0	1
Data Access - How-To Create a Database	0	0	1	0	2	0	0	1
NET Framework - How-To Send Mail	1	1	1	1	2	0	0	1
Windows Forms - How-To Automate Office	0	0	1	0	2	0	0	1
Advanced .NET Framework (Windows Services) - Create a Windows Service	0	0	1	0	2	0	0	1

ตารางที่ 4.1 ผลการทดลอง

ตัวเลขจะแสดงจำนวน Pattern ที่พบในโปรแกรมที่นำมาทดลอง

จากการทดลองพบว่า Design Pattern ที่พบมากที่สุดคือ Facade Pattern รองลงมาเป็น Adapter Pattern, Proxy Pattern ตามลำดับ ในขณะที่ไม่พบ Mediator Pattern และ Chain of Responsibility เลย ทั้งนี้เนื่องจากเป็นไปตามความถี่บ่อยในการเรียกใช้ และ เนื่องจากกลุ่มโค้ดโปรแกรมที่ใช้มาจากกลุ่มเดียวกันทำให้พบว่าบาง Design Pattern มีการใช้มากเป็นพิเศษ ในขณะที่บาง Design Pattern ไม่พบเลย

4.4 ผลการทดลองการหา Pattern ใน Code

การทดลองโดยการนำ Program mining design pattern program รันทดสอบกับซอสโค้ดโปรแกรมภาษา C# ใน MSDN 101 C# Samples

ผลการรันโปรแกรม

1. โปรแกรม Advanced - Multithreading - How-To Async Calls

Number of Patern1: 3

```
Patern1(frmAbout,System,frmMain,frmTaskProgress) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System)
&INHERITANCE(frmTaskProgress, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_,_));
```

Pattern1.rml

2. โปรแกรม Advanced - Serialization - How-To Serializing Objects

Number of Patern1: 1

Number of Patern2: 3

```
Patern1(Class2,ISerializable,SerializationInfo,StreamingContext) :=
INHERITANCE(Class2, ISerializable)
&CONTAINMENT(Class2, SerializationInfo)
&CONTAINMENT(Class2, StreamingContext) ;
PRINT "Number of Patern1: ", #(Patern1(Class2,_,_,_));
```

Pattern1.rml

```
Patern2(frmAbout,System,frmMain) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System) ;
PRINT "Number of Patern2: ", #(Patern2(frmAbout,_,_));
```

Pattern2.rml

3. โปรแกรม Advanced .NET Framework - Interacting Windows Service

Number of Patern1: 2

```
Patern1(frmAbout,System,frmMain) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System)
&CONTAINMENT(frmMain, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_));
```

Pattern1.rml

4. โปรแกรม Advanced .NET Framework - Make WIn32 API Calls

Number of Patern1: 1

Number of Patern2: 3

```
Patern1(Class2,ISerializable,SerializationInfo,StreamingContext) :=
INHERITANCE(Class2, ISerializable)
&CONTAINMENT(Class2, SerializationInfo)
&CONTAINMENT(Class2, StreamingContext) ;
PRINT "Number of Patern1: ", #(Patern1(Class2,_,_,_));
```

Pattern1.rml

```
Patern2(frmAbout,System,frmMain) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System) ;
PRINT "Number of Patern2: ", #(Patern2(frmAbout,_,_));
```

Pattern2.rml

5. โปรแกรม Interop - Automate IE

Number of Patern1: 3

```
Patern1(frmAbout,System,frmMain,mshtml,frmStatus) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System)
&CONTAINMENT(frmMain, mshtml)
&INHERITANCE(frmStatus, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_,_));
```

Pattern1.rml

6.โปรแกรม Language - How-To Arrays

Number of Patern1: 4

Number of Patern2: 3

```
Patern1(frmAbout,System,frmInputBox,KeyEventArgs,frmMain,Array,WhichListBox) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmInputBox, System)
&CONTAINMENT(frmInputBox, KeyEventArgs)
&INHERITANCE(frmMain, System)
&CONTAINMENT(frmMain, Array)
&CONTAINMENT(frmMain, WhichListBox) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_,_,_,_));
```

Pattern1.rml

```
Patern2(Customer,IComparable,CompareField) :=
INHERITANCE(Customer, IComparable)
&CONTAINMENT(Customer, CompareField) ;
PRINT "Number of Patern2: ", #(Patern2(Customer,_,_));
```

Pattern2.rml

7. โปรแกรม WinForms - Dynamic Control Creation

Number of Patern1: 3

```
Patern1(frmAbout,System,frmMain,Object,EventArgs,XmlNode,Control,Point,frmSurvey
Form) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System)
&CONTAINMENT(frmMain, Object)
&CONTAINMENT(frmMain, EventArgs)
&CONTAINMENT(frmMain, XmlNode)
&CONTAINMENT(frmMain, Control)
&CONTAINMENT(frmMain, Point)
&INHERITANCE(frmSurveyForm, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_,_,_,_,_,_,_));
```

Pattern1.rml

8. โปรแกรม Windows Forms- How-To Custom Exceptions

Number of Patern1: 2

```
Patern1(frmAbout,System,frmMain) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System)
&CONTAINMENT(frmMain, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_));
```

Pattern1.rml

9. โปรแกรม Windows Forms - XP Theme Support

Number of Patern1: 2

```
Patern1(frmAbout,System,frmMain) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_));
```

Pattern1.rml

10. โปรแกรม Windows Forms - Use the Clipboard

Number of Patern1: 2

```
Patern1(frmAbout,System,frmMain) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_));
```

Pattern1.rml

11. โปรแกรม Windows Forms - Use Regular Expressions

Number of Patern1: 2

```
Patern1(frmAbout,System,frmImageViewer,frmMain,ImageAttributes,Stream,frmStatus)
:=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmImageViewer, System)
&INHERITANCE(frmMain, System)
&CONTAINMENT(frmMain, ImageAttributes)
&CONTAINMENT(frmMain, Stream)
&INHERITANCE(frmStatus, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_,_,_,_,_));
```

Pattern1.rml

12. โปรแกรม Windows Forms - Use Format Codes to Format Data in Strings

Number of Patern1: 2

```
Patern1(frmAbout,System,frmMain) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_));
```

Pattern1.rml

13. โปรแกรม Windows Forms - Use Crystal Reports

Number of Patern1: 11.

Number of Patern2: 11.

Number of Patern3: 11.

```
Patern1(AllCustomersOrders,ReportClass,CustomerOrders,TenMostExpensiveProducts,T
op5ProductsSold) :=
INHERITANCE(AllCustomersOrders, ReportClass)
&INHERITANCE(CustomerOrders, ReportClass)
&INHERITANCE(TenMostExpensiveProducts, ReportClass)
&INHERITANCE(Top5ProductsSold, ReportClass) ;
PRINT "Number of Patern1: ", #(Patern1(AllCustomersOrders,_,_,_,_));
```

Pattern1.rml

```
Patern2(CachedAllCustomersOrders,Component,ICachedReport,CachedCustomerOrders,
CachedTenMostExpensiveProducts,CachedTop5ProductsSold) :=
INHERITANCE(CachedAllCustomersOrders, Component)
&INHERITANCE(CachedAllCustomersOrders, ICachedReport)
&INHERITANCE(CachedCustomerOrders, Component)
&INHERITANCE(CachedCustomerOrders, ICachedReport)
&INHERITANCE(CachedTenMostExpensiveProducts, Component)
&INHERITANCE(CachedTenMostExpensiveProducts, ICachedReport)
&INHERITANCE(CachedTop5ProductsSold, Component)
&INHERITANCE(CachedTop5ProductsSold, ICachedReport) ;
PRINT "Number of Patern2: ", #(Patern2(CachedAllCustomersOrders,_,_,_,_,_));
```

Pattern2.rml

```
Patern3(frmAbout,System,frmMain,frmStatus) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System)
&INHERITANCE(frmStatus, System) ;
PRINT "Number of Patern3: ", #(Patern3(frmAbout,_,_,_));
```

Pattern3.rml

14. โปรแกรม Windows Forms - Simple Printing

Number of Patern1: 2

```
Patern1(frmAbout,System,frmMain) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmMain, System)
&CONTAINMENT(frmMain, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_));
```

Pattern1.rml

15. โปรแกรม NET Framework - How-To Use the EventLog

Number of Patern1: 5

```
Patern1(frmAbout,System,frmCreateDelete,frmMain,frmRead,frmWrite) :=
INHERITANCE(frmAbout, System)
&INHERITANCE(frmCreateDelete, System)
&INHERITANCE(frmMain, System)
&INHERITANCE(frmRead, System)
&INHERITANCE(frmWrite, System) ;
PRINT "Number of Patern1: ", #(Patern1(frmAbout,_,_,_,_,_));
```

Pattern1.rml

บทที่ 5
สรุป

5.1 สรุปที่มาและความสัมพันธ์

เนื่องจากการเขียนโปรแกรมในปัจจุบันนั้นมีรูปแบบการเขียนโปรแกรมที่ทำงานในลักษณะเดียวกันที่หลากหลายทำให้อยากในการนำโค้ดโปรแกรมที่แต่ละคนเขียนนั้นมาใช้ร่วมกันหรืออยากที่จะบอกว่าการเขียนในรูปแบบใดที่ดีที่สุดดังนั้นจึงมีคนกลุ่มหนึ่งได้ทำการรวบรวมรูปการเขียนโปรแกรมขึ้นมาเป็น Design pattern ซึ่งคนกลุ่มนี้คือ Gang of Four ซึ่งได้รวบรวมไว้เป็น 3 หมวดทั้งหมด 23 Pattern ที่ได้การยอมรับว่าสามารถนำมาใช้ได้จริงและมีประสิทธิภาพ

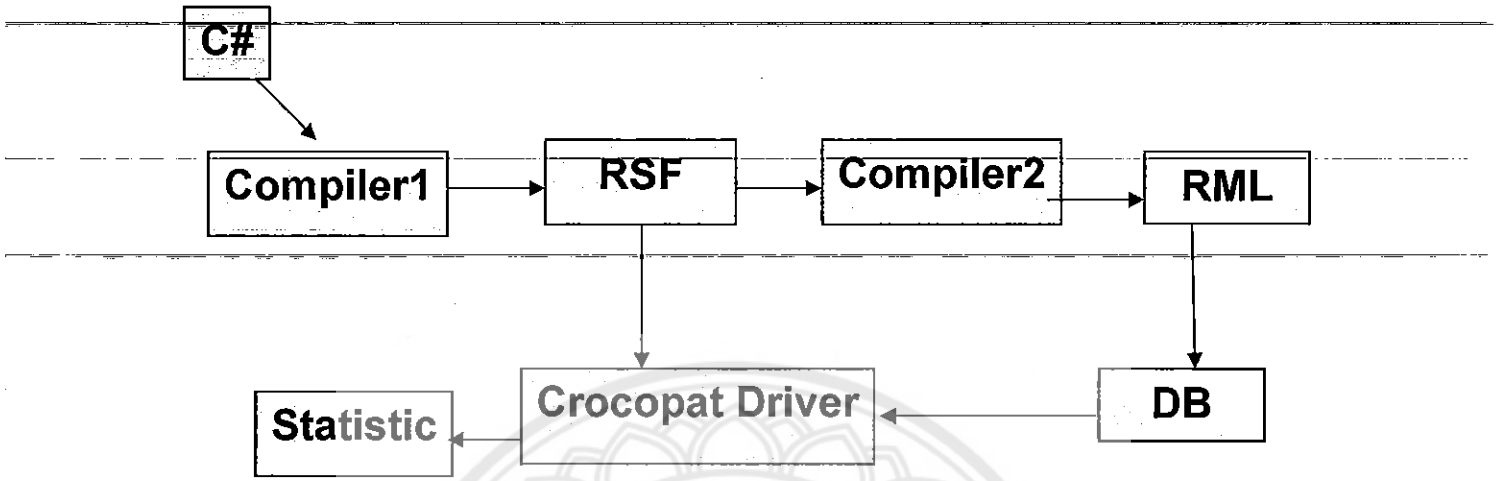
แต่ในการเขียนโปรแกรมยังมีรูปแบบที่หลากหลายกว่านั้นคืออาจมี Pattern อื่นๆ อีกนอกเหนือจาก 23 pattern ที่กล่าวในข้างต้นแล้วดังนั้นเพื่อทำการตรวจสอบหา pattern ที่มีในโปรแกรมในรูปแบบอื่นที่มีการใช้กันอยู่ในโค้ดโปรแกรมจึงได้จัดทำ โปรแกรมสำหรับตรวจหา Design pattern นี้ขึ้นมาเพื่อทำการตรวจหา Pattern ต่างๆ จากโค้ดโปรแกรมที่มีอยู่จัดทำเป็นสถิติว่ามีการใช้ Pattern ใดบ้างกับโค้ดโปรแกรมที่ทำการตรวจสอบ

5.2 สรุปการดำเนินงาน

5.2.1. ต้องศึกษาข้อมูลพื้นฐานที่ใช้ในการทำงานต่างๆ ดังนี้

- การเขียนโปรแกรมด้วย Microsoft visual studio 2005 โดยใช้ภาษา C#
- โครงสร้างและไวยกรณ์ของ ภาษา C# ซึ่งเป็นโค้ดโปรแกรมที่จะนำมาใช้ในการตรวจหาPattern
- ศึกษาการใช้งานโปรแกรม Crocopat ซึ่งเป็น Tool สำหรับ Pattern Recognizer
- ศึกษาการใช้งานโปรแกรม Coco-R ซึ่งเป็น Tool สำหรับสร้าง Compiler

5.2.2. ทำการเขียนโปรแกรมมาใช้งานซึ่งโปรแกรมประกอบด้วยองค์ประกอบดังนี้



รูปที่ 5.1 สรุปโครงสร้างของโปรแกรม

1. Compiler C# to RSF

คือส่วนที่แปลงโค้ดโปรแกรมภาษา C# มาเป็นไฟล์ RSF ซึ่งโค้ดโปรแกรมในส่วนนี้ได้ใช้ Tool ที่ใช้ในการสร้าง Compiler คือ Coco-R ช่วยการสร้างโค้ดโปรแกรมซึ่งประกอบด้วย

1.1 Scanner ทำหน้าที่ในการอ่านซ้อสโค้ด C# เข้ามาและทำการ จัดแบ่งกลุ่มตัวอักษรเป็น Token ซึ่ง Token หนึ่งจะเก็บค่ากลุ่มคำหรืออักขระที่มีความหมายในโค้ดภาษา C# โดยก่อนที่จะทำการแบ่งเป็น Token ได้ทำการตัดส่วนที่เป็น Comment ของโค้ดโปรแกรมออกแล้ว

1.2 Parser ทำหน้าที่ในการตรวจไวยกรณ์ของภาษา C# ว่าถูกต้องตามไวยกรณ์โดยทำการวนลูปตรวจสอบไปทีละ Token หากพบไวยกรณ์ก็แจ้งออกมา

1.3 Recognition เป็นส่วนที่ทำหน้าที่ตรวจหาโครงสร้างของโปรแกรมเพื่อนำมาสร้างเป็นไฟล์ RSF ต่อไปการทำงานส่วนนี้ก็จะทำการวนลูปตรวจสอบเพื่อหาความสัมพันธ์ของของ Class ในโปรแกรมภาษา C# โดยความสัมพันธ์ที่ตรวจสอบมี 3 อย่างคือ Inheritance, containment และ Implement โดยค่าที่ได้จะเก็บไว้ในลิสท์โครงสร้างของลิสท์ประกอบด้วย Class Name, list Inheritance, list containment และ list Implement นับเป็นหนึ่ง Node

2. Compiler RSF to RML

คือส่วนที่ทำการแปลงความสัมพันธ์ทั้งหมดที่มีในโค้ดโปรแกรมภาษา C# หรือ RSF มาเป็น RML ซึ่งเป็นความสัมพันธ์ย่อยๆ ที่มีอยู่ในโปรแกรมซึ่งการแบ่งออกมาเป็นความสัมพันธ์ย่อยๆ นี้ทำการแบ่งโดยใช้ความสัมพันธ์แบบ Inheritance และ containment เป็นหลักโดยหา Class ที่มีความสัมพันธ์ดังกล่าวต่อกันทั้งหมดเป็นโครงสร้างความสัมพันธ์ย่อยๆ ต่อกันเพื่อนำมาสร้างมาเป็น RMLแล้วเก็บไว้ในฐานข้อมูลต่อไป ดังนั้นในหนึ่ง RSF จึงสามารถหาได้หลายๆ RML

3. Database

คือส่วนที่เก็บรวบรวมไฟล์ RML ไว้เพื่อใช้ในการรันเปรียบเทียบกับไฟล์ RSF ต่อไป

4. Crocopat Driver

เป็นส่วนที่ใช้ในการสั่งรันโปรแกรมCrocopat เพื่อใช้ในการตรวจสอบระหว่าง RSF กับ RML ได้ค่าเป็นสถิติจำนวนรูปแบบความสัมพันธ์ที่มีใน RML ที่มีใน RSF นั่นเองโดยตัวที่ใช้ในการรันจะเขียนเป็น Bath file ที่มีรูปแบบคำสั่งดังนี้

```
crocopat\crocopat Patern1.rml < test.rsf > output.txt
echo . >> output.txt
crocopat\crocopat Patern2.rml < test.rsf >> output.txt
```

5.3 สรุปผลการทดลองการหา Pattern ใหม่ใน Code

จากการทดลองพบว่า Pattern ที่พบในมีความหลากหลายโดยบาง Pattern ที่พบเป็นรูปง่ายๆ ไม่มีความซับซ้อน ซึ่ง Pattern ในลักษณะนี้จะสามารถมีโอกาสที่จะมีจำนวนครั้งที่พบในโปรแกรมได้มากกว่า Pattern ที่มีความซับซ้อนและในบาง Pattern ส่วนมากที่มีความซับซ้อนจะพบครั้งเดียวแล้วไม่พบอีกเลย

5.4 สรุปการหา Pattern โดยใช้ Standard Pattern Code

ในการ Run Program สามารถนำ Code ที่มี Patatern ดังนี้มาตรวจสอบคือ

1. Builder
2. Composite
3. Adapter
4. Chain of Res.
5. Facad
6. Mediator
7. Proxy
8. AbstractFactory

ตัวโปรแกรม จะแสดงว่าใน Code ที่นำมาทดสอบจะมี Pattern เหล่านี้อยู่กี่ตัวโดย ข้อจำกัดคือ ต้องทดสอบทีละตัว

5.5 ปัญหา และการแก้ไข

ปัญหาที่พบ

1. การทำงานของโปรแกรมพบว่ามีบางโค้ดโปรแกรมที่มี Design Pattern แล้วตรวจสอบไม่พบว่ามี Design Pattern นั้นเนื่องมาจากในการเขียนโปรแกรมนั้นมีการนำไปประยุกต์ใช้ที่หลากหลายทำให้ตรวจสองไม่พบ
2. ในการทดลองใช้โปรแกรมเพื่อตรวจสอบโค้ดโปรแกรมเพื่อหาความถี่ว่า Design Pattern ตัวไหนมีการนำมาใช้บ่อยนั้นเนื่องจากโค้ดโปรแกรมที่นำมาใช้เป็น Input นั้นเป็นตัวอย่างจากกลุ่มโปรแกรมเดียวกันทำให้ผลที่ได้พบว่าบาง Design Pattern มีการใช้บ่อยมากในขณะที่ บาง Design Pattern แทบจะไม่มีการใช้เลย

การแก้ไข

1. แก้ไข RML File ให้มีการแสดงค่าของ Pattern ออกมาเป็นจำนวน Pattern และ Genrate RSF ที่ ตรงตามความสัมพันธ์ของ Code มากขึ้น
2. หา Code ที่มีการใช้งานจริง และหลากหลายขึ้น เช่น Real World example Code ที่นำไปใช้งานจริง

5.6 ข้อเสนอแนะ

1. สามารถนำ Code ที่มีความซ้ำซ้อนมากขึ้น และมี Pattern หลายตัวอยู่ใน Code เดียวกัน มาหาว่า มี Pattern อะไรบ้างใน Code นั้น
2. สามารถแสดงสถิติของการ Run Pattern ต่างๆ เก็บรวบรวมเอาไว้

เอกสารอ้างอิง

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

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

[3] 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. http://www.engr.mun.ca/~theo/JavaCC-FAQ/. 2006.

[5] Balanyi, Z., Ferenc, R. "**Mining design patterns from C++ source code**" Proc. IEEE-ISCAS. 2003, pp. 305-314

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

[6] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals1" **Towards Pattern Based Design Recovery**" Proc. IEEE-ISCAS. 2003

[7] David Braun, Jeff Sivils, Alex Shapiro, Jerry Versteegh "**Object Oriented Analysis and Design Team**"[Online]. Available: http://unicoi.kennesaw.edu

[8] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, Janos Lele "**Design Pattern Mining Enhanced by Machine Learning**" Proc. IEEE-ISCAS. 2005

[9] สิทธิโชค เชาวกุล. **เอกสารประกอบการเรียนวิชา Compiler**. 2547

[10] Jayanta. **"Binary decision diagram"**. [Online]. http://en.wikipedia.org/wiki/Binary_decision_diagram. 2007

[11] Austin. "Data & Object Factory" . [Online]. http://www.dofactory.com/Default.aspx. 2001 - 2007

[12] Hanspeter Mössenböck. **"The Compiler Generator Coco/R"**. [Online]. http://ssw.jku.at/Coco/#CS. 2006

ภาคผนวก ก

ตัวอย่างโปรแกรมที่มี Design Pattern ฝังอยู่

Code ที่นำมาเป็นตัวอย่างนี้ เป็น Pattern ที่อยู่นอกนเหนือจากการทำงานของ Program และสามารถ พัฒนาตัว File RML เพิ่มเติมได้ และนำมาประยุกต์ใช้กับ Program ต่อไปได้

// Visitor pattern -- Real World example

```
using System;
using System.Collections;

namespace DoFactory.GangOfFour.Visitor.RealWorld
{

 // MainApp startup application

 class MainApp
 {
  static void Main()
  {
   // Setup employee collection
   Employees e = new Employees();
   e.Attach(new Clerk());
   e.Attach(new Director());
   e.Attach(new President());

   // Employees are 'visited'
   e.Accept(new IncomeVisitor());
   e.Accept(new VacationVisitor());

   // Wait for user
```

```
    Console.Read();
  }
}

// "Visitor"

interface IVisitor
{
  void Visit(Element element);
}

// "ConcreteVisitor1"

class IncomeVisitor : IVisitor
{
  public void Visit(Element element)
  {
    Employee employee = element as Employee;

    // Provide 10% pay raise
    employee.Income *= 1.10;
    Console.WriteLine("{0} {1}'s new income: {2:C}",
      employee.GetType().Name, employee.Name,
      employee.Income);
  }
}

// "ConcreteVisitor2"

class VacationVisitor : IVisitor
{
  public void Visit(Element element)
```

```
        {
         Employee employee = element as Employee;

         // Provide 3 extra vacation days
         Console.WriteLine("{0} {1}'s new vacation days: {2}",
          employee.GetType().Name, employee.Name,
          employee.VacationDays);
        }
    }

    class Clerk : Employee
    {
     // Constructor
     public Clerk() : base("Hank", 25000.0, 14)
     {
     }
    }

    class Director : Employee
    {
     // Constructor
     public Director() : base("Elly", 35000.0, 16)
     {
     }
    }

    class President : Employee
    {
     // Constructor
     public President() : base("Dick", 45000.0, 21)
     {
     }
```

```
}

// "Element"

abstract class Element
{
 public abstract void Accept(IVisitor visitor);
}

// "ConcreteElement"

class Employee : Element
{
 string name;
 double income;
 int vacationDays;

 // Constructor
 public Employee(string name, double income,
  int vacationDays)
 {
  this.name = name;
  this.income = income;
  this.vacationDays = vacationDays;
 }

 // Properties
 public string Name
 {
  get{ return name; }
  set{ name = value; }
 }
```

```
    public double Income
    {
     get{ return income; }
     set{ income = value; }
    }

    public int VacationDays
    {
     get{ return vacationDays; }
     set{ vacationDays = value; }
    }

    public override void Accept(IVisitor visitor)
    {
     visitor.Visit(this);
    }
  }

  // "ObjectStructure"

  class Employees
  {
   private ArrayList employees = new ArrayList();

   public void Attach(Employee employee)
   {
    employees.Add(employee);
   }

   public void Detach(Employee employee)
   {
```

```
    employees.Remove(employee);
  }

  public void Accept(IVisitor visitor)
  {
   foreach (Employee e in employees)
   {
    e.Accept(visitor);
   }
   Console.WriteLine();
  }
 }
}
```

==========

// Singleton pattern -- Real World example

```
using System;
using System.Collections;
using System.Threading;

namespace DoFactory.GangOfFour.Singleton.RealWorld
{

 // MainApp test application

 class MainApp
 {
  static void Main()
  {
   LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
```

```
        LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
        LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
        LoadBalancer b4 = LoadBalancer.GetLoadBalancer();

        // Same instance?
        if (b1 == b2 && b2 == b3 && b3 == b4)
        {
          Console.WriteLine("Same instance\n");
        }

        // All are the same instance -- use b1 arbitrarily
        // Load balance 15 server requests
        for (int i = 0; i < 15; i++)
        {
          Console.WriteLine(b1.Server);
        }

        // Wait for user
        Console.Read();
      }
    }

    // "Singleton"

    class LoadBalancer
    {
      private static LoadBalancer instance;
      private ArrayList servers = new ArrayList();

      private Random random = new Random();

      // Lock synchronization object
```

```
        private static object syncLock = new object();

        // Constructor (protected)
        protected LoadBalancer()
        {
         // List of available servers
         servers.Add("ServerI");
         servers.Add("ServerII");
         servers.Add("ServerIII");
         servers.Add("ServerIV");
         servers.Add("ServerV");
        }

        public static LoadBalancer GetLoadBalancer()
        {
         // Support multithreaded applications through
         // 'Double checked locking' pattern which (once
         // the instance exists) avoids locking each
         // time the method is invoked
         if (instance == null)
         {
          lock (syncLock)
          {
           if (instance == null)
           {
            instance = new LoadBalancer();
           }
          }
         }

         return instance;
        }
```

```
    // Simple, but effective random load balancer

    public string Server
    {
      get
      {
        int r = random.Next(servers.Count);
        return servers[r].ToString();
      }
    }
  }
}
```

// Template pattern -- Real World example

```
using System;
using System.Data;
using System.Data.OleDb;

namespace DoFactory.GangOfFour.Template.RealWorld
{

  // MainApp test application

  class MainApp
  {
    static void Main()
    {
      DataAccessObject dao;
```

```
            dao = new Categories();
            dao.Run();

            dao = new Products();
            dao.Run();

            // Wait for user
            Console.Read();
        }
    }

    // "AbstractClass"

    abstract class DataAccessObject
    {
        protected string connectionString;

        protected DataSet dataSet;

        public virtual void Connect()
        {
            // Make sure mdb is on c:\
            connectionString =
                "provider=Microsoft.JET.OLEDB.4.0; " +
                "data source=c:\\nwind.mdb";
        }

        public abstract void Select();
        public abstract void Process();

        public virtual void Disconnect()
        {
```

```
    connectionString = "";
  }

  // The "Template Method"

  public void Run()
  {
    Connect();
    Select();
    Process();
    Disconnect();
  }
}

// "ConcreteClass"

class Categories : DataAccessObject
{
  public override void Select()
  {
    string sql = "select CategoryName from Categories";
    OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
      sql, connectionString);

    dataSet = new DataSet();
    dataAdapter.Fill(dataSet, "Categories");
  }

  public override void Process()
  {
    Console.WriteLine("Categories ---- ");
```

```
    DataTable dataTable = dataSet.Tables["Categories"];
    foreach (DataRow row in dataTable.Rows)
    {
     Console.WriteLine(row["CategoryName"]);
    }
    Console.WriteLine();
   }
  }

  class Products : DataAccessObject
  {
   public override void Select()
   {
    string sql = "select ProductName from Products";
    OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
     sql, connectionString);

    dataSet = new DataSet();
    dataAdapter.Fill(dataSet, "Products");
   }
   public override void Process()
   {
    Console.WriteLine("Products ---- ");
    DataTable dataTable = dataSet.Tables["Products"];
    foreach (DataRow row in dataTable.Rows)
    {
     Console.WriteLine(row["ProductName"]);
    }
    Console.WriteLine();
   }
  }
 }
```

ประวัติผู้เขียนโครงงาน

ชื่อ นายสุภชาติ เสนวิรัช

ภูมิลำเนา บ้านเลขที่ 157 หมู่ 14 ต. แม่นาเรือ
อ. เมือง จ. พะเยา รหัสไปรษณีย์ 56000

ประวัติการศึกษา

- จบการศึกษาระดับมัธยมศึกษาจาก
  โรงเรียนฟากกว๊านวิทยาคม
- ปัจจุบันกำลังศึกษาอยู่ชั้นปีที่ 4 [พ.ศ. 2550]
  สาขาวิชาวิศวกรรมคอมพิวเตอร์
  คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร

E-mail sandwichzz@yahoo.com

Telephone 08-4990-0576

ชื่อ นายเฉลิมพล สุภาษิ

ภูมิลำเนา บ้านเลขที่ 83 หมู่ 9 ต. อิปุ่ม
อ. ด่านซ้าย จ. เลย รหัสไปรษณีย์ 42120

ประวัติการศึกษา

- จบการศึกษาระดับมัธยมศึกษาจาก
  โรงเรียนหล่มเก่าพิทยาคม
- ปัจจุบันกำลังศึกษาอยู่ชั้นปีที่ 4 [พ.ศ. 2550]
  สาขาวิชาวิศวกรรมคอมพิวเตอร์
  คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร

E-mail lerm_su@hotmail.com

Telephone 08-9577-3526