การออกแบบและพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบด้วยจาวาบีน

นาย กิรพัฒน์ วิเชียรกิจ

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาศาสตร์คอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2545
ISBN 974-17-1678-8

DESIGN AND DEVELOPMENT OF SOFTWARE COMPONENTS OF DESIGN PATTERNS USING JAVABEAN

Mr. Kirapuk Vichiankij

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computer Science
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2002

ISBN 974-17-1678-8

หัวข้อวิทยานิพนธ์	การออกแบบและพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบด้วยจาวาบีน
โดย	นาย กิรพัฒน์ วิเชียรกิจ
สาขาวิชา	วิทยาศาสตร์คอมพิวเตอร์
อาจารย์ที่ปรึกษา	ผู้ช่วยศาสตราจารย์ ดร. พรศิริ หมื่นไชยศรี

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

…….. คณบดี คณะวิศวกรรมศาสตร์
(ศาสตราจารย์ ดร.สมศักดิ์ ปัญญาแก้ว)

คณะกรรมการสอบวิทยานิพนธ์

…… ประธานกรรมการสอบ
(ผู้ช่วยศาสตราจารย์ กอบกุล เตชะวณิช)

…….. อาจารย์ที่ปรึกษา
(ผู้ช่วยศาสตราจารย์ ดร.พรศิริ หมื่นไชยศรี)

……กรรมการ
(ผู้ช่วยศาสตราจารย์ ดร.ธาราทิพย์ สุวรรณศาสตร์)

……กรรมการ
(อาจารย์ ดร.อาทิตย์ ทองทักษ์)

กิรพัฒน์ วิเชียรกิจ : การออกแบบและพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบด้วยจาวาบีน. (DESIGN AND DEVELOPMENT OF SOFTWARE COMPONENTS OF DESIGN PATTERNS USING JAVABEAN) อ. ที่ปรึกษา : ผู้ช่วยศาสตราจารย์ ดร. พรศิริ หมื่นไชยศรี, 121 หน้า. ISBN 974-17-1678-8.

วิทยานิพนธ์นี้ได้ทำการออกแบบและพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบด้วยจาวาบีนที่สามารถนำไปใช้ได้ในเครื่องมือช่วยการพัฒนาภาษาจาวาแบบเห็นภาพ โดยการออกแบบและพัฒนาแบ่งเป็น 2 ส่วน คือ ส่วนการติดต่อกับผู้ใช้ของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบขณะพัฒนาโปรแกรม และส่วนการสร้างชุดคำสั่งของรูปแบบการออกแบบ ซึ่งส่วนประกอบซอฟต์แวร์นี้จะสร้างชุดคำสั่งของรูปแบบการออกแบบ พร้อมทั้งคำอธิบายชุดคำสั่งให้ในขณะพัฒนาโปรแกรมด้วยเครื่องมือช่วยการพัฒนาภาษาจาวาแบบเห็นภาพโดยอัตโนมัติ ดังนั้นส่วนประกอบซอฟต์แวร์นี้สามารถช่วยให้การนำรูปแบบการออกแบบกลับไปใช้ใหม่ทำได้ง่ายยิ่งขึ้น โดยทำให้ผู้พัฒนาเขียนชุดคำสั่งน้อยลง และสามารถควบคุมความถูกต้องของชุดคำสั่งได้ดีขึ้น

ภายหลังจากการพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ได้ทำการทดสอบการใช้ส่วนประกอบซอฟต์แวร์นี้ในการพัฒนาโปรแกรมประยุกต์จำนวนสามโปรแกรม ผลการทดสอบพบว่า ส่วนประกอบซอฟต์แวร์นี้สามารถช่วยในการเขียนโครงชุดคำสั่งของรูปแบบการออกแบบ ซึ่งเป็นโครงสร้างหลักของโปรแกรมได้อย่างถูกต้อง

ภาควิชา วิศวกรรมคอมพิวเตอร์ ลายมือชื่อนิสิต .....

สาขาวิชา วิทยาศาสตร์คอมพิวเตอร์ ลายมือชื่ออาจารย์ที่ปรึกษา .....

ปีการศึกษา 2545

4270225721 : MAJOR COMPUTER SCIENCE

KEY WORD: DESIGN PATTERN / SOFTWARE COMPONENT / JAVA BEAN

KIRAPUK VICHIANKIJ : DESIGN AND DEVELOPMENT OF SOFTWARE COMPONENTS OF DESIGN PATTERNS USING JAVABEAN , THESIS ADVISOR : ASSISTANT PROFESSOR PORNSIRI MUENCHAISRI, Ph.D., 121 pp. ISBN 974-17-1678-8.

This thesis presents a design and development of software components of design patterns using java bean that can be used in java visual tools. The design and development consists of two sections: user interface of software components of design patterns, and the code generating part of design patterns. These software components are capable of automatically generating java source codes of design patterns with comments in java visual tools. Thus, design patterns reusability is easily achieved with these software components, the amount of addition coding is reduced, and reliability of the source codes is increased.

After developing software components of design patterns, they were tested by constructing three applications. The result of the tests found that these software components could help developers to construct design patterns source code that is the main structure of the program correctly.

Department Computer Engineering Student's signature .....

Field of study Computer Science Advisor's signature .....

Academic year 2002

กิตติกรรมประกาศ

ขอขอบคุณ ผู้ช่วยศาสตราจารย์ ดร.พรศิริ หมื่นไชยศรี อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่ให้ความช่วยเหลืออย่างดียิ่ง ท่านได้สละเวลาให้คำแนะนำ และข้อคิดเห็นต่างๆ ประกอบการทำงานวิจัยของข้าพเจ้ามาโดยตลอด ทำให้วิทยานิพนธ์ฉบับนี้สำเร็จไปได้ด้วยดี

ขอขอบคุณ ผู้ช่วยศาสตราจารย์ กอบกุล เตชะวณิช ผู้ช่วยศาสตราจารย์ ดร. ธาราทิพย์ สุวรรณศาสตร์ และ อาจารย์ ดร. อาทิตย์ ทองทักษ์ กรรมการวิทยานิพนธ์ ที่ท่านได้กรุณาให้คำแนะนำและข้อชี้แนะ ในการตรวจสอบ และแก้ไขวิทยานิพนธ์ฉบับนี้

ขอขอบคุณเพื่อนๆ พี่ๆ น้องๆ ที่ได้ให้คำแนะนำ และกำลังใจแก่ข้าพเจ้าตลอดเวลาที่ศึกษาในภาควิชาวิศวกรรมคอมพิวเตอร์แห่งนี้

ท้ายที่สุด ข้าพเจ้าใคร่ขอกราบขอบพระคุณบิดา มารดา ของข้าพเจ้าที่สนับสนุน และให้กำลังใจแก่ข้าพเจ้าเสมอมาจนสำเร็จการศึกษา

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

หน้า

สารบัญภาพ

หน้า

สารบัญภาพ (ต่อ)

หน้า

สารบัญภาพ (ต่อ)

หน้า

สารบัญภาพ (ต่อ)

หน้า

สารบัญภาพ (ต่อ)

หน้า

บทที่ 1

บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

การเขียนโปรแกรมเชิงวัตถุ (Object-oriented programming) ช่วยให้เกิดการเปลี่ยนแปลงในการออกแบบและพัฒนาซอฟต์แวร์ โดยทำให้สามารถพัฒนาซอฟต์แวร์เป็นส่วนๆ ได้ดีขึ้น จึงเกิดอุตสาหกรรมการทำชิ้นส่วนเหล่านี้ขึ้น และเรียกชิ้นส่วนเหล่านี้ว่า ส่วนประกอบซอฟต์แวร์ (Software component) ซึ่งช่วยให้ส่วนของโปรแกรมต่างๆ ที่ต้องเขียนเป็นประจำไม่ต้องเขียนซ้ำ แต่เอาส่วนประกอบเหล่านี้กลับมาใช้ได้ใหม่ (Reuse) และช่วยให้การปรับปรุง บำรุงรักษาซอฟต์แวร์ทำได้ง่าย

อย่างไรก็ตาม ยังพบว่าส่วนประกอบซอฟต์แวร์สามารถช่วยผู้พัฒนาโปรแกรมนำชุดคำสั่งในการทำงานกลับมาใช้ใหม่ (Code reuse) แต่สถาปัตยกรรมการออกแบบไม่ได้มีการนำกลับมาใช้ ดังนั้นจึงมีการค้นคว้าวิจัยเพื่อนำสถาปัตยกรรมการออกแบบมาใช้ใหม่จนสำเร็จ เรียกว่า รูปแบบการออกแบบ (Design pattern) ซึ่งผู้ออกแบบระบบสามารถนำสถาปัตยกรรมที่เคยคิดได้แล้วกลับมาใช้ใหม่ได้ (Architectures reuse) แต่การนำกลับมาใช้ใหม่นี้สามารถนำรูปแบบการออกแบบกลับมาใช้ใหม่ได้ในระดับของแนวความคิด หรือวิธีการออกแบบที่อธิบายไว้ได้ในหนังสือเท่านั้น ถ้าจะนำมาใช้ในโปรแกรมประยุกต์ใหม่ก็จะต้องอ่านให้เข้าใจแนวคิดแล้วนำมาเขียนชุดคำสั่งใหม่เองทั้งหมด จะเห็นว่าผู้ที่จะนำสถาปัตยกรรมกลับมาใช้ใหม่ไม่สามารถนำมาใช้ได้อย่างง่ายๆ เหมือนส่วนประกอบซอฟต์แวร์ที่เป็นการนำกลับมาใช้ในระดับชุดคำสั่งในการทำงาน

ที่ผ่านมาจักกะ วิลจามา (Jukka Viljamaa) [5] ได้มีทำงานวิจัยในเรื่องคล้ายๆ กันนี้ โดยมีการใช้รูปแบบการออกแบบในการสร้างโครงร่าง (Framework) ที่จะใช้สำหรับพัฒนาโปรแกรมประยุกต์ โดยจะเน้นหนักที่การสร้างเครื่องมือที่จะสนับสนุนการนำเอารูปแบบการออกแบบไปใช้ในโครงร่าง แต่ไม่ได้เน้นที่การใช้รูปแบบการออกแบบในการสร้างโปรแกรมประยุกต์

ต่อมา นิเวศน์ จรัสดำรง [10] ได้ทำวิจัยในลักษณะการพัฒนาโครงร่าง ซึ่งงานวิจัยนี้เป็นการพัฒนาโครงร่าง ซึ่งจะมีการนำเอารูปแบบการออกแบบเข้าไปใช้ช่วยสร้างชุดคำสั่ง ซึ่งไม่สามารถนำชุดคำสั่งที่ได้ไปพัฒนาโปรแกรมประยุกต์ต่อ โดยใช้เครื่องมือช่วยการพัฒนาแบบเห็นภาพ (Visual tools) ซึ่งเป็นที่นิยมในปัจจุบันได้

วิทยานิพนธ์นี้มีจุดมุ่งหมายในการแปลงรูปแบบการออกแบบให้อยู่ในรูปของส่วนประกอบซอฟต์แวร์ที่สามารถนำไปใช้ได้ในเครื่องมือช่วยการพัฒนาภาษาจาวาแบบเห็นภาพ (Java visual tools) ซึ่งจะทำให้ง่ายและสะดวกขึ้นในการนำกลับไปใช้ใหม่

1.2 วัตถุประสงค์ของการวิจัย

ออกแบบและพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบด้วยจาวาบีนที่สามารถนำไปใช้ได้ในเครื่องมือช่วยการพัฒนาภาษาจาวาแบบเห็นภาพ

1.3 ขอบเขตของการวิจัย

1.3.1 พัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ โดยจะยึดตามหนังสือ Design Patterns: Element of Reusable Object-Oriented Software [1] เป็นหลัก ทั้งหมด 23 รูปแบบการออกแบบ อันได้แก่ แอ็บสแทรคท์แฟคทอรี บิวเดอร์ แฟคทอรีเมท็อด โปรโตไทพ์ ซิงเกิลตอน อแด็พเตอร์ บริดจ์ คอมโพสิต เดคอเรเตอร์ ฟาซาด ไฟลเวจ พร็อกซี เชนออฟเรสปอนสิบิลิตี คอมมานด์ อินเตอร์พรีเตอร์ อิเทอเรเตอร์ เมดิเอเตอร์ มีเมนโต ออบเซิร์ฟเวอร์ สเตท สเทรทิจี เทมเพลตเมท็อด และ วิสิเตอร์

1.3.2 ส่วนประกอบซอฟต์แวร์ที่จะสร้างจะเป็นจาวาบีน โดยสนับสนุนภาษาจาวามาตรฐานในปัจจุบัน

1.3.3 ทดสอบการใช้งานได้ของส่วนประกอบซอฟต์แวร์ที่พัฒนาขึ้น ด้วยการพัฒนาโปรแกรมประยุกต์โดยใช้ส่วนประกอบซอฟต์แวร์ที่พัฒนาขึ้น

1.4 ขั้นตอนในการดำเนินงานวิจัย

1.4.1 ศึกษาแนวคิดของรูปแบบการออกแบบ

1.4.2 ศึกษาแนวคิดและการพัฒนาซอฟต์แวร์ให้อยู่ในรูปส่วนประกอบซอฟต์แวร์

1.4.3 ศึกษาการพัฒนาส่วนประกอบซอฟต์แวร์ด้วยจาวาบีน

1.4.4 ออกแบบวิธีการแปลงรูปแบบการออกแบบให้เป็นส่วนประกอบซอฟต์แวร์ที่ใช้งานได้ง่าย

1.4.5 ออกแบบและพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบแต่ละแบบ

1.4.6 ทดสอบส่วนประกอบซอฟต์แวร์ที่พัฒนาได้ว่าทำงานได้ถูกต้อง

1.4.7 ทดสอบพัฒนาโปรแกรมที่ไม่ซับซ้อนมากโดยใช้ส่วนประกอบซอฟต์แวร์ที่สร้างขึ้นกับเครื่องมือช่วยการพัฒนาภาษาจาวาแบบเห็นภาพ

1.4.8 สรุปผล และจัดทำวิทยานิพนธ์

1.5 ประโยชน์ที่คาดว่าจะได้รับ

1.5.1 สามารถนำส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบไปใช้ทำให้ประหยัดเวลาในการพัฒนาโปรแกรม และการทดสอบโปรแกรม

1.5.2 ทำให้การออกแบบและพัฒนาโปรแกรมเชิงวัตถุมีระบบระเบียบมากขึ้น ด้วยการหันมาใช้รูปแบบการออกแบบที่เหมาะสม แทนที่จะเพียงพัฒนาโปรแกรมแต่ละครั้งให้ทำงานได้ถูกต้องเท่านั้น ซึ่งจะทำให้การแก้ไขโปรแกรมในภายหลังง่ายขึ้น และยังลดความเสี่ยงที่การออกแบบที่คิดขึ้นใหม่ในแต่ละครั้งมีจุดบกพร่อง

บทที่ 2

ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

2.1 ทฤษฎีที่เกี่ยวข้อง

2.1.1 รูปแบบการออกแบบ [4]

รูปแบบการออกแบบ คือ สถาปัตยกรรม หรือโครงสร้างการพัฒนาซอฟต์แวร์ที่สามารถนำกลับมาใช้ใหม่ได้ และสามารถแก้ไขปัญหาเฉพาะอย่างในลักษณะเดียวกันที่พบเป็นประจำได้ โดยในรูปแบบการออกแบบจะอธิบายเมท็อดในคลาสหรือคลาสย่อย (Subclass) ที่ทำงานร่วมกันหรือคลาสหลายๆ คลาสที่ทำงานร่วมกัน [2] โดยรูปแบบการออกแบบนี้ทำให้สถาปัตยกรรมซอฟต์แวร์บางอย่างซึ่งเคยใช้มาแล้วนำกลับมาใช้ได้ในโปรแกรมประยุกต์หรือระบบใหม่โดยไม่ต้องคิดวิธีการขึ้นใหม่ แต่จะใช้วิธีการเดิมซึ่งถูกต้องและมีประสิทธิภาพ [3] ซึ่งจะมีการกำหนดเป็นรูปแบบทั่วไปไว้โดย อิริก แกมมา (Erich Gamma) และคณะ ซึ่งเรียกว่า แกงค์ ออฟ โฟร์ (Gang of Four หรือ GoF) [1] เพื่อให้มีการนำกลับมาประยุกต์ใช้ใหม่กับโปรแกรมประยุกต์ใหม่ๆ ได้

รูปแบบการออกแบบแบ่งออกเป็น 3 ประเภทใหญ่ๆ คือ [4]

1. รูปแบบการออกแบบสำหรับสร้างวัตถุ (Creational patterns) เป็นรูปแบบการออกแบบที่ใช้สร้างวัตถุของคลาส (Instance) ในรูปแบบต่างๆ

2. รูปแบบการออกแบบโครงสร้าง (Structural patterns) เป็นรูปแบบการออกแบบที่เป็นโครงสร้างตัวต่อประสาน (Interface) ของคลาสต่างๆ เพื่อทำงานอย่างหนึ่ง ส่วนใหญ่มักใช้ในลักษณะเป็นตัวต่อประสานของกลุ่มของคลาสนั้นกับคลาสอื่น

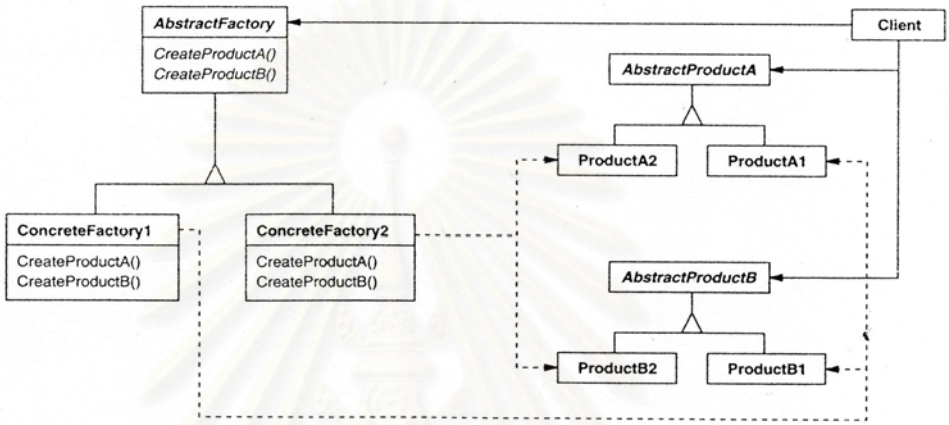
3. รูปแบบการออกแบบพฤติกรรม (Behavioral patterns) เป็นรูปแบบการออกแบบที่เป็นการทำงานอย่างหนึ่งโดยมีพฤติกรรมเฉพาะตัว

2.1.1.1 รูปแบบการออกแบบสำหรับสร้างวัตถุ

2.1.1.1.1 แอ็บสแทรคท์แฟคทอรี (Abstract factory)

รูปแบบการออกแบบแอ็บสแทรคท์แฟคทอรี ใช้สำหรับสร้างตัวต่อประสาน (Interface) หรือคลาสนามธรรม (Abstract Class) ในการสร้างวัตถุซึ่งเกี่ยวข้องกัน ทำให้ไม่ต้องระบุคลาสย่อยในการทำงานจริงของโปรแกรม

รูปแบบการออกแบบนี้ ช่วยให้การปรับเปลี่ยนคลาสที่ใช้ในการสร้างวัตถุระหว่างโปรแกรมทำงาน (Run time) สามารถทำได้โดยไม่ต้องเรียกเมท็อดที่ทำหน้าที่เดียวกันของแต่ละคลาสแยกจากกัน แต่จะเขียนรวมไว้ในตัวต่อประสานหรือคลาสนามธรรมแทน ซึ่งทำให้ปรับเปลี่ยนคลาสได้โดยผู้เรียกใช้ (Client) สามารถใช้ชุดคำสั่งเดิมในการทำงานกับทุกคลาสได้ โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.1



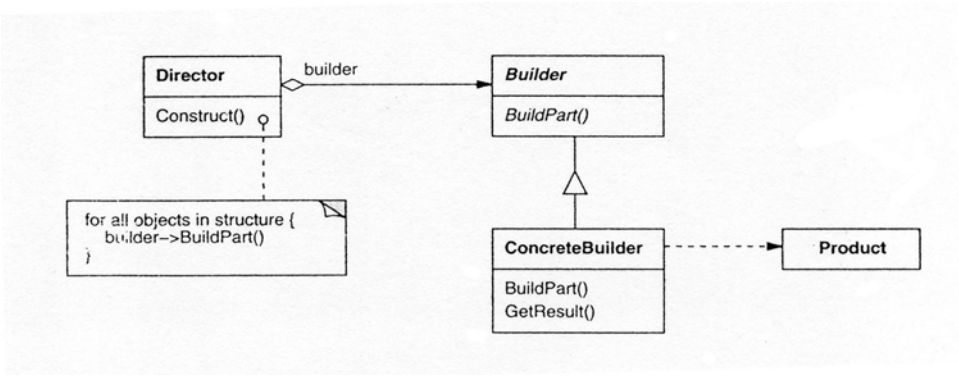
รูปที่ 2.1 โครงสร้างของรูปแบบการออกแบบแอ็บสแทรคท์แฟคทอรี [1]

คลาส ConcreteFactory1 และ ConcreteFactory2 มีตัวต่อประสาน AbstractFactory ทำให้สามารถสลับคลาสลูกซึ่งเป็นคลาสที่ใช้สร้างวัตถุที่เกี่ยวข้องกันระหว่างโปรแกรมทำงานได้ง่าย เนื่องจากทั้งคลาส ConcreteFactory1 และ ConcreteFactory2 จะถือว่าเป็นคลาส AbstractFactory ด้วย

2.1.1.1.2 บิวเดอร์ (Builder)

รูปแบบการออกแบบบิวเดอร์ ใช้สำหรับแบ่งการสร้างวัตถุที่ซับซ้อนโดยการสร้างทีละส่วน ทำให้กระบวนการสร้างวัตถุแบบหนึ่งสามารถสร้างวัตถุที่หลากหลายได้

รูปแบบการออกแบบนี้ ช่วยให้การสร้างวัตถุที่มีโครงสร้างซับซ้อนหลายๆ วัตถุแต่มีโครงสร้างเหมือนกัน สามารถทำได้โดยไม่ต้องเขียนกระบวนการสร้างใหม่ของทุกวัตถุ แต่จะเขียนรวมไว้ในคลาสหลักแทน ซึ่งทำให้ปรับเปลี่ยนคลาสได้โดยผู้สั่งสร้าง (Director) สามารถใช้ชุดคำสั่งเดิมในกระบวนการสร้างทำการสร้างวัตถุได้ โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.2



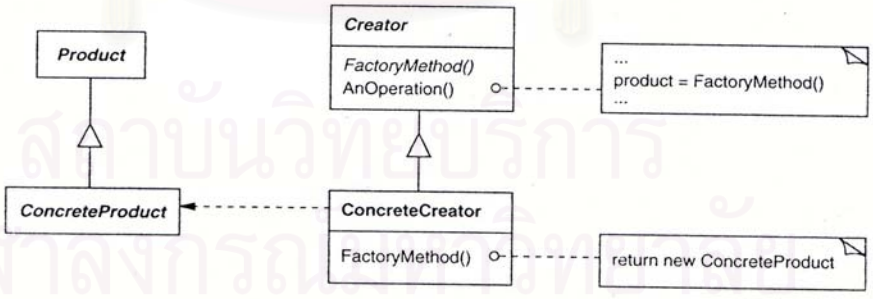
รูปที่ 2.2 โครงสร้างของรูปแบบการออกแบบบิวเดอร์ [1]

คลาส Builder และ ConcreteBuilder จะมีเมท็อด BuilderPart() ซึ่งใช้ในการสั่งสร้างชิ้นส่วน และมีเมท็อด GetResult() สำหรับใช้รับค่าวัตถุสุดท้ายซึ่งสร้างครบทุกส่วนแล้ว โดยคลาส Director จะเป็นผู้สั่งสร้างทีละส่วน

2.1.1.1.3 แฟคทอรีเมท็อด (Factory method)

รูปแบบการออกแบบแฟคทอรีเมท็อด ใช้สำหรับสร้างตัวต่อประสานที่มีจุดประสงค์ในการสร้างวัตถุ แต่ให้คลาสลูกเป็นผู้ตัดสินใจว่าจะสร้างวัตถุจากคลาสใด

รูปแบบการออกแบบนี้ ช่วยให้สามารถใช้เมท็อดการสร้างวัตถุในคลาสหลักที่ยังไม่ได้กำหนดวัตถุจริงที่จะสร้างได้ โดยให้คลาสลูกเป็นผู้ตัดสินใจว่าจะสร้างวัตถุนั้นจากคลาสใด ดังนั้น ทำให้สามารถเขียนการทำงานของคลาสหลักได้โดยไม่จำเป็นต้องรู้วัตถุที่ใช้ทำงานจริงขณะโปรแกรมทำงาน โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.3



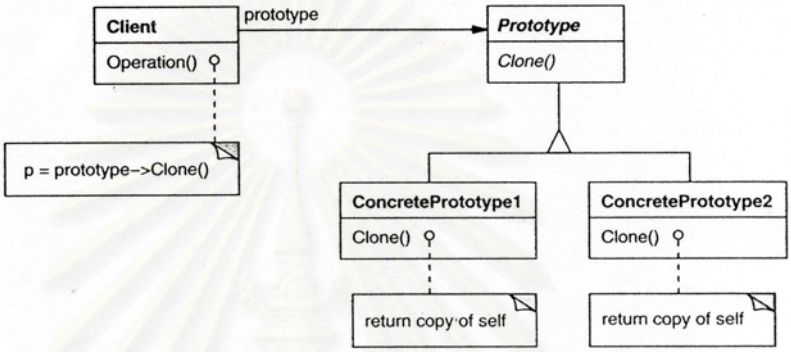
รูปที่ 2.3 โครงสร้างของรูปแบบการออกแบบแฟคทอรีเมท็อด [1]

คลาสนามธรรม Creator มีเมท็อด FactoryMethod() อยู่ซึ่งยังเป็นเมท็อดนามธรรม แต่มีการเรียกใช้ในเมท็อด AnOperation() จากนั้นคลาสลูก ConcreteCreator จึงระบุการทำงานของเมท็อด FactoryMethod() ว่าจะสร้างวัตถุนั้นอย่างไร ซึ่งในที่นี้คลาส ConcreteCreator ได้เลือกสร้างวัตถุจากคลาส ConcreteProduct

2.1.1.1.4 โปรโตไทพ์ (Prototype)

รูปแบบการออกแบบโปรโตไทพ์ ใช้สร้างวัตถุที่เหมือนกันกับวัตถุตัวแม่แบบโดยใช้การคัดลอก

รูปแบบการออกแบบนี้ช่วยให้การสร้างวัตถุของคลาสใดๆ ที่มีความหลากหลายของสถานะไม่มากนักสามารถสร้างได้โดยง่ายกว่าการสร้างโดยตรง รวมถึงทำให้การสร้างวัตถุที่ไม่รู้คลาสที่จะสร้าง ให้สามารถสร้างวัตถุจากตัวแม่แบบแทนได้ โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.4

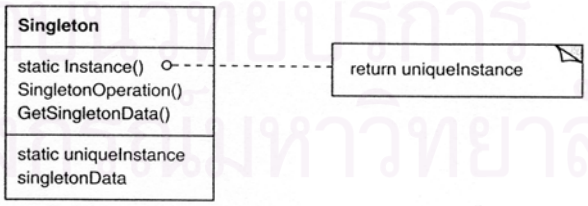


รูปที่ 2.4 โครงสร้างของรูปแบบการออกแบบโปรโตไทพ์ [1]

คลาส Prototype จะมีเมท็อด Clone() ที่ใช้ในการคัดลอกตัวเองอยู่ ซึ่งจะส่งคืนวัตถุที่เหมือนกับตัวเอง

2.1.1.1.5 ซิงเกิลตอน (Singleton)

รูปแบบการออกแบบซิงเกิลตอน ใช้สำหรับสร้างวัตถุของคลาสที่ต้องการโดยให้มีเพียงวัตถุเดียวเท่านั้น ปกติใช้ในลักษณะโกลบอล (Global) โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.5



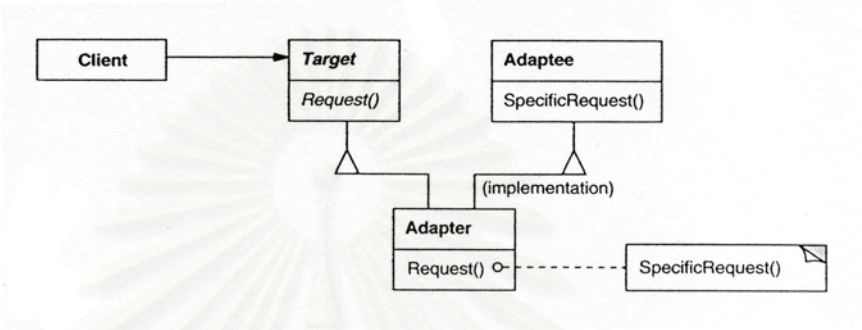
รูปที่ 2.5 โครงสร้างของรูปแบบการออกแบบซิงเกิลตอน [1]

คลาส Singleton มีเมท็อด Instance() ที่ใช้ในการส่งวัตถุตัวเองอยู่และมีเขตข้อมูลแบบสถิต uniqueInstance ซึ่งใช้ในการเก็บวัตถุตัวเอง ทั้งนี้ คลาสนี้ต้องป้องกันการถูกสร้างจากภายนอกโดยกำหนดให้ตัวสร้าง (Constructor) ไม่เป็นสาธารณะด้วย

2.1.1.2 รูปแบบการออกแบบแบบโครงสร้าง ได้แก่

2.1.1.2.1 อแด็พเตอร์ (Adapter)

รูปแบบการออกแบบอแด็พเตอร์ ใช้สำหรับแปลงโครงสร้างตัวต่อประสานที่ไม่เหมือนกันกับที่ต้องการใช้ให้ใช้ด้วยกันได้ โดยการสร้างคลาสใหม่ขึ้นเพื่อปรับเปลี่ยนตัวต่อประสาน ทำให้สามารถนำคลาสที่มีอยู่กลับมาใช้ใหม่ได้ โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.6



รูปที่ 2.6 โครงสร้างของรูปแบบการออกแบบอแด็พเตอร์ [1]

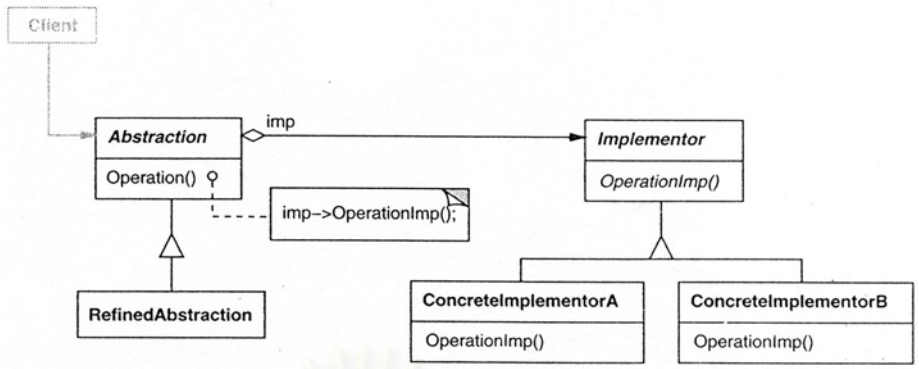
คลาส Adaptee เป็นคลาสที่มีเมท็อดคำร้องที่ต้องการใช้ และคลาส Target เป็นคลาสที่มีเมท็อดคำร้องที่ใช้อยู่ ซึ่งทั้งสองมี เมท็อดคำร้องไม่เหมือนกัน จึงทำการสร้างคลาส Adapter ขึ้นมาเพื่อแปลงการเรียกเมท็อด ทำให้คลาสทั้งสองใช้ด้วยกันได้

2.1.1.2.2 บริดจ์ (Bridge)

รูปแบบการออกแบบบริดจ์ ใช้เพื่อหลีกเลี่ยงการผูกติดกันของคลาสส่วนนามธรรมกับส่วนการทำงานจริง ทำให้ทั้งสองส่วนเป็นอิสระจากกัน และปรับเปลี่ยนได้ง่าย

เมื่อส่วนนามธรรมสามารถมีส่วนทำงานจริงได้หลายรูปแบบ วิธีการปกติคือการใช้การรับทอด (Inheritance) แต่การทำเช่นนั้นทำให้ส่วนนามธรรมผูกติดกับส่วนการทำงานจริง ซึ่งยากต่อการปรับเปลี่ยนหรือเพิ่มขยายทั้งสองส่วนโดยอิสระ รูปแบบการออกแบบนี้สามารถช่วยให้คลาสส่วนนามธรรมกับส่วนการทำงานจริงเป็นอิสระจากกัน โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.7

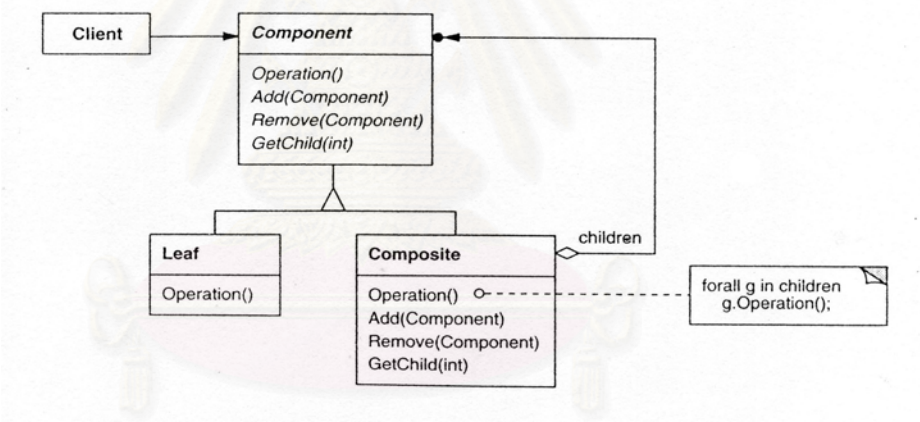
คลาส Abstraction เป็นส่วนนามธรรมโดยมีเขตข้อมูล imp สำหรับเก็บค่าคลาส Implementor ซึ่งเป็นส่วนการทำงานจริง โดยมีชุดคำสั่งในส่วนนามรรมซึ่งจะเรียกใช้เมท็อดของส่วนการทำงานจริง ทำให้ทั้งสองส่วนไม่ผูกติดกัน สามารถปรับเปลี่ยนขณะโปรแกรมทำงานได้



รูปที่ 2.7 โครงสร้างของรูปแบบการออกแบบบริดจ์ [1]

2.1.1.2.3 คอมโพสิต (Composite)

รูปแบบการออกแบบคอมโพสิต ใช้ในการรวมกันของหลายๆ คลาสที่มีลักษณะเดียวกันเป็นโครงสร้างต้นไม้ ทำให้ผู้เรียกใช้มองวัตถุเดี่ยวกับโครงสร้างของวัตถุที่รวมตัวกันในลักษณะเดียวกัน โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.8



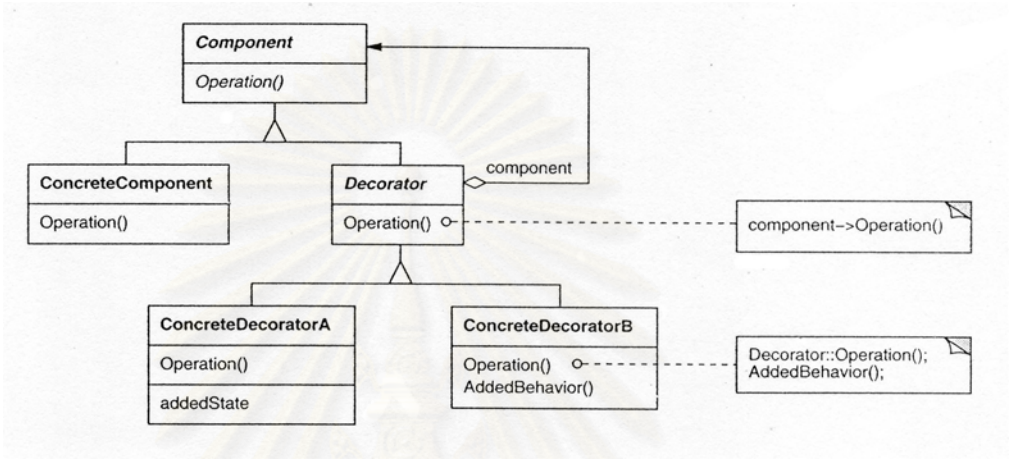
รูปที่ 2.8 โครงสร้างของรูปแบบการออกแบบคอมโพสิต [1]

คลาส Composite มีเขตข้อมูลโครงสร้าง children ใช้เก็บข้อมูลวัตถุของคลาส Component ซึ่งเก็บได้หลายวัตถุ ถ้าข้อมูลที่เก็บคือวัตถุของคลาส Composite จะสามารถทำให้มีการเก็บต่อๆ กันไปเป็นโครงสร้างต้นไม้ของคลาส Component ได้ หรือถ้าเป็นส่วนใบ (Leaf) ของโครงสร้างต้นไม้จะเก็บวัตถุเดี่ยวแทน

2.1.1.2.4 เดคอเรเตอร์ (Decorator)

รูปแบบการออกแบบเดคอเรเตอร์ ใช้สำหรับเพิ่มความสามารถเข้าไปในวัตถุแบบพลวัต ซึ่งทำให้เกิดการขยายวัตถุที่ยืดหยุ่น

รูปแบบการออกแบบนี้ ใช้เมื่อต้องการเพิ่มเติมความสามารถให้กับวัตถุใดวัตถุหนึ่งเท่านั้น โดยไม่ต้องการเพิ่มความสามารถให้กับคลาส เนื่องจากถ้าเพิ่มเติมให้กับคลาสจะมีผลกับทุกวัตถุของคลาส รูปแบบการออกแบบนี้ยังสามารถใช้ในกรณีที่การสร้างคลาสย่อยไม่เหมาะสม เนื่องจากมีส่วนขยายหลายรูปแบบ การสร้างคลาสย่อยจะทำให้เกิดคลาสปริมาณมาก รวมถึงสามารถใช้ในกรณีที่ไม่สามารถสร้างคลาสย่อยได้อีกด้วย โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.9



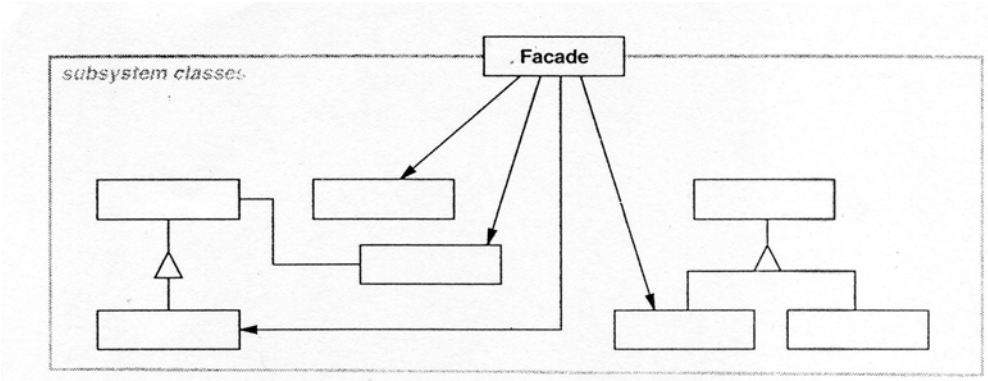
รูปที่ 2.9 โครงสร้างของรูปแบบการออกแบบเดคอเรเตอร์ [1]

คลาส Component ซึ่งเป็นคลาสที่ต้องการเพิ่มความสามารถ และคลาสลูก Decorator มีเขตข้อมูล component ซึ่งเก็บวัตถุของคลาส Component ที่ต้องการขยายความสามารถ มีเมท็อด Operation() ซึ่งทำการเรียกการทำงานของวัตถุที่เก็บไว้ในเขตข้อมูล component เมื่อมีการสร้างคลาสลูกของ Decorator จะสามารถเพิ่มความสามารถให้กับวัตถุของคลาส Component ได้ โดยการเรียกใช้ เมท็อด Operation() ของคลาส Decorator และทำการเพิ่มความสามารถหลังจากเรียกใช้

2.1.1.2.5 ฟาซาด (Facade)

รูปแบบการออกแบบฟาซาด ใช้สำหรับเป็นศูนย์กลางในการติดต่อกับระบบย่อย ทำให้การใช้ระบบย่อยง่ายขึ้น โดยให้คลาสภายนอกติดต่อกับฟาซาดแทน โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.10

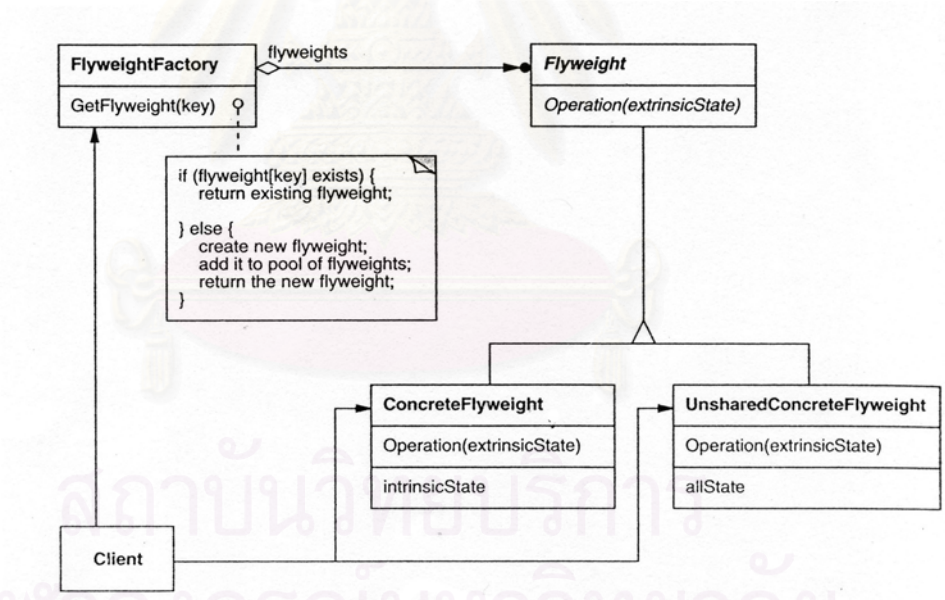
คลาส Facade จะเป็นศูนย์กลางในการติดต่อกับระบบย่อย ทำให้การใช้งานระบบจากภายนอกดูง่ายขึ้นโดยการติดต่อมาที่คลาส Façade เท่านั้นแทนการติดต่อกับคลาสภายในโดยตรง



รูปที่ 2.10 โครงสร้างของรูปแบบการออกแบบฟาซาด [1]

2.1.1.2.6 ไฟลเวจ (Flyweight)

รูปแบบการออกแบบไฟลเวจ ใช้ในการทำงานร่วมกันแบบแบ่งสรร (sharing) ของวัตถุที่มีขนาดเล็กและมีปริมาณมากอย่างมีประสิทธิภาพ โดยการสร้างกองรวมที่ใช้ในการแบ่งสรรวัตถุขึ้น โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.11

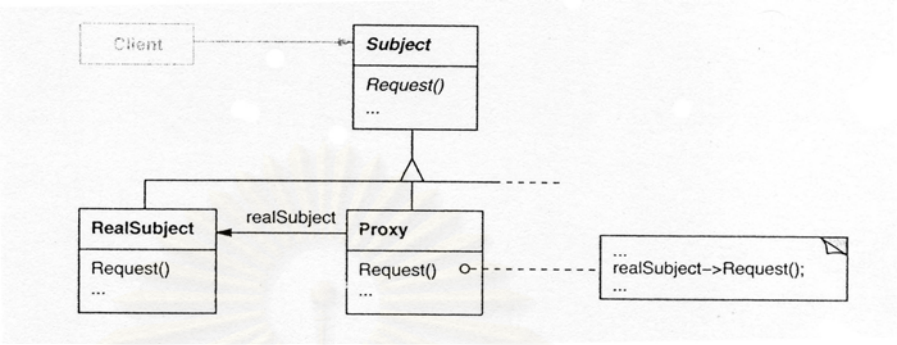


รูปที่ 2.11 โครงสร้างของรูปแบบการออกแบบไฟลเวจ [1]

คลาส FlyweightFactory จะทำการสร้างวัตถุของคลาส Flyweight ซึ่งเป็นคลาสที่ต้องการแบ่งสรรขึ้นและบันทึกลงในกองรวม และให้คลาส Client ติดต่อขอใช้วัตถุที่แบ่งสรรที่คลาส FlyweightFactory หากคลาส Client เรียกใช้วัตถุของคลาส Flyweight ใดที่มีอยู่แล้วก็จะไปดึงจากในกองรวมออกมาใช้งานได้ทันที ทำให้ลดการสร้างวัตถุของคลาส Flyweight ลงได้มาก

2.1.1.2.7 พร็อกซี (Proxy)

รูปแบบการออกแบบพร็อกซี ใช้เป็นตัวแทนในการเข้าถึงวัตถุอื่น หรือใช้เพื่อควบคุมหรือกระทำการบางอย่างก่อนการเข้าถึงวัตถุอื่น โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.12



รูปที่ 2.12 โครงสร้างของรูปแบบการออกแบบพร็อกซี [1]

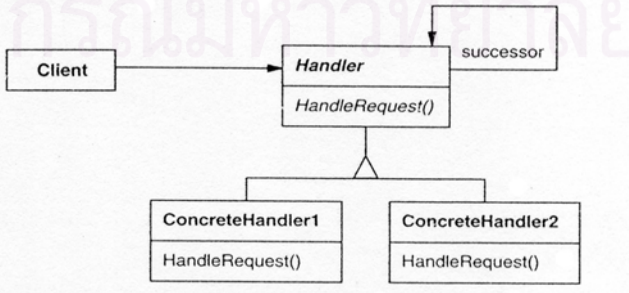
คลาส RealSubject เป็นคลาสที่ทำงานจริง และคลาส Proxy เป็นคลาสตัวแทนในการใช้คลาส RealSubject ซึ่งคลาส Client จะเก็บวัตถุของคลาส Proxy แทนคลาส RealSubject ในการใช้งาน โดยจะมองว่าเป็นวัตถุของคลาส Subject

2.1.1.3 รูปแบบการออกแบบแบบพฤติกรรม

2.1.1.3.1 เชนออฟเรสปอนสิบิลิตี (Chain of responsibility)

รูปแบบการออกแบบเชนออฟเรสปอนสิบิลิตี ใช้เพื่อหลีกเลี่ยงการผูกติดกันของผู้ร้องขอและผู้รับ ซึ่งมีวัตถุผู้รับหลายวัตถุที่มีโอกาสจัดการตามคำร้อง

รูปแบบการออกแบบนี้ ใช้ในกรณีที่อาจจะมีวัตถุที่สามารถจัดการตามคำร้องได้หลายวัตถุ แต่วัตถุที่มีหน้าที่จัดการในขณะนั้นมีเพียงวัตถุเดียว และผู้ร้องขอไม่สามารถรู้ว่าเป็นวัตถุใด หรือไม่ต้องการระบุลงไปซึ่งจะทำให้ไม่ยืดหยุ่นต่อการทำงานโดยรูปแบบการออกแบบนี้จะทำให้วัตถุผู้รับคำร้องอยู่ในลักษณะของโซ่ของวัตถุ เมื่อมีคำร้องจะทำการส่งไปตามโซ่ของวัตถุจนกระทั่งมีผู้รับกระทำตามคำร้อง โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.13



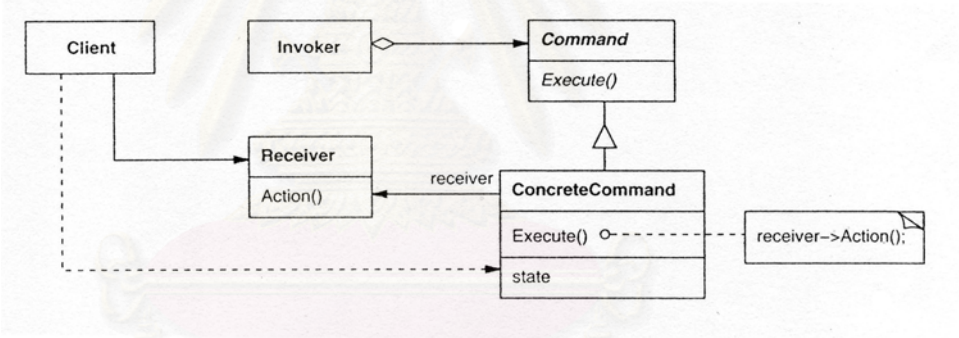
รูปที่ 2.13 โครงสร้างของรูปแบบการออกแบบเชนออฟเรสปอนสิบิลิตี [1]

คลาส Handler มีเขตข้อมูลชื่อ successor ซึ่งเก็บวัตถุของคลาสตนเอง เพื่อใช้เป็นโซ่ของวัตถุ เมื่อคลาส Client เรียกเมท็อด HandlerRequest() จะทำการตรวจสอบว่าเป็นคำร้องที่จัดการได้หรือไม่ หากจัดการไม่ได้สามารถส่งต่อคำร้องไปตามโซ่ของวัตถุได้

2.1.1.3.2 คอมมานด์ (Command)

รูปแบบการออกแบบคอมมานด์ ใช้ทำให้การร้องขอหรือการปฏิบัติการเป็นวัตถุ ซึ่งทำให้จัดการเกี่ยวกับการร้องขอหรือการปฏิบัติการได้หลากหลาย

รูปแบบการออกแบบนี้ ใช้เมื่อต้องการคำร้องให้ทำการปฏิบัติการบางอย่างแต่ไม่รู้ว่าเมท็อดที่ทำงานตามที่ต้องการอยู่ในคลาสใด รวมถึงอาจจะไม่รู้ว่าเมท็อดที่ต้องการคือเมท็อดใด ดังนั้นรูปแบบการออกแบบนี้จะทำให้คำร้องอยู่ในรูปวัตถุแทนที่จะเป็นการเรียกเมท็อดที่สามารถส่งผ่านไปยังวัตถุอื่นๆ ได้ โดยกุญแจสำคัญของรูปแบบการออกแบบนี้ คือคลาสนามธรรมของคำร้องที่ประกาศเมท็อดการปฏิบัติการ (คลาส Command ในรูปที่ 2.14) โดยให้คลาสย่อยของคลาสนี้จะเก็บวัตถุของผู้รับคำร้องและจัดการตามคำสั่ง โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.14



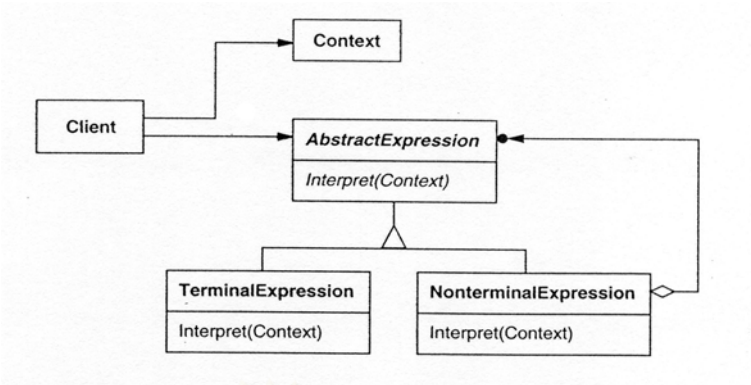
รูปที่ 2.14 โครงสร้างของรูปแบบการออกแบบคอมมานด์ [1]

คลาส Invoker เป็นคลาสที่ออกคำสั่งโดยคำสั่งจะอยู่ในรูปของวัตถุของคลาส ConcreteCommand เพื่อไปเรียกเมท็อดในคลาส Receiver อีกต่อหนึ่ง

2.1.1.3.3 อินเตอร์พรีเตอร์ (Interpreter)

รูปแบบการออกแบบอินเตอร์พรีเตอร์ ใช้เป็นตัวแปลภาษาที่มีไวยกรณ์ไม่ซับซ้อน โดยรูปแบบการออกแบบนี้จะสร้างโครงสร้างของประโยคขึ้นเป็นโครงสร้างต้นไม้ และนำประโยคที่ต้องการแปลมาเปรียบเทียบไปตามโครงสร้างต้นไม้

อนึ่ง รูปแบบการออกแบบนี้จะไม่ใช้กับงานที่คำนึงถึงประสิทธิภาพมากนัก งานการแปลที่ซับซ้อนควรทำในรูปแบบอื่น โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.15



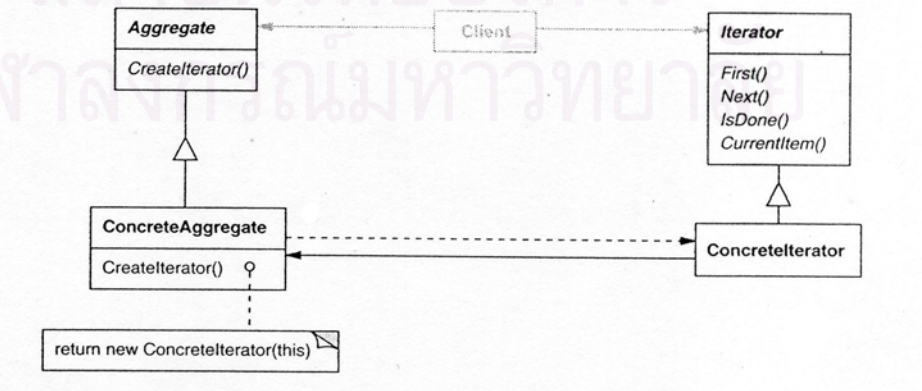
รูปที่ 2.15 โครงสร้างของรูปแบบการออกแบบอินเตอร์พรีเตอร์ [1]

คลาส TerminalExpression เป็นคลาสของคำๆ เดียว คลาส NonterminalExpression เป็นคลาสของประโยคที่มีคำมากกว่าหนึ่งคำ มีเขตข้อมูลโครงสร้างซึ่งใช้เก็บข้อมูลวัตถุของคลาส AbstractExpression เพื่อใช้เก็บข้อมูลคำต่อไปของประโยคเป็นโครงสร้างต้นไม้ที่ใช้ในการแปลภาษาตามที่ระบุไว้ในคลาส Context

2.1.1.3.4 อิเทอเรเตอร์ (Iterator)

รูปแบบการออกแบบอิเทอเรเตอร์ ใช้ทำให้การเข้าถึงวัตถุที่เป็นโครงสร้างข้อมูล (Aggregate Object) ไว้ได้แบบเป็นลำดับได้ โดยไม่จำเป็นต้องรู้ถึงโครงสร้างข้อมูลของวัตถุนั้น

การเข้าถึงโครงสร้างข้อมูลในบางครั้ง อาจต้องการที่จะเข้าถึงข้อมูลในลักษณะต่างๆ เช่น ต้องการลำดับในการเข้าถึงหลายๆ แบบในโครงสร้างข้อมูลเดียว รูปแบบการออกแบบนี้ได้มอบความรับผิดชอบในการเข้าถึงและสำรวจ (Traversal) โครงสร้างข้อมูลให้กับคลาสอีกคลาสหนึ่ง เพื่อให้คลาสนั้นจัดการและเก็บข้อมูลการเข้าถึง โดยไม่ต้องรวมการจัดการการเข้าถึงซึ่งอาจมีหลายแบบในคลาสของโครงสร้างข้อมูล โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.16



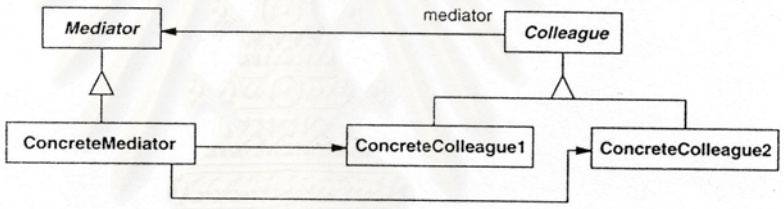
รูปที่ 2.16 โครงสร้างของรูปแบบการออกแบบอิเทอเรเตอร์ [1]

คลาส Aggregate คือคลาสของวัตถุโครงสร้างข้อมูล มีเมท็อด CreateIterator() เพื่อใช้ในการสร้างคลาส Iterator ที่ใช้ในการเข้าถึงโครงสร้างข้อมูลภายใน เมื่อคลาส Client เรียกใช้ จะสามารถใช้เมท็อดของคลาส Iterator จัดการข้อมูลของคลาส Aggregate ได้

2.1.1.3.5 เมดิเอเตอร์ (Mediator)

รูปแบบการออกแบบเมดิเอเตอร์ ใช้ทำให้วัตถุสามารถติดต่อกันได้โดยไม่ต้องอ้างถึงกันแบบชัดแจ้ง ทำให้วัตถุเปลี่ยนแปลงการติดต่อกันได้อย่างอิสระ

ในกรณีที่มีวัตถุจำนวนมากที่ทำงานขึ้นกับกันและกัน การติดต่อกันระหว่างวัตถุจะมีปริมาณมากและซับซ้อน รูปแบบการออกแบบนี้จะรวมการติดต่อไว้ที่คลาสเดียว โดยให้ทุกคลาสที่ทำงานร่วมกันติดต่อมาที่คลาสนี้คลาสเดียว เพื่อลดการติดต่อสื่อสารที่ซับซ้อนลง รวมถึงเปลี่ยนแปลงการติดต่อระหว่างวัตถุเหล่านี้ให้กระทำได้สะดวกขึ้น นอกจากนี้ยังทำให้วัตถุที่ทำงานร่วมกันเป็นอิสระจากกันด้วย โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.17



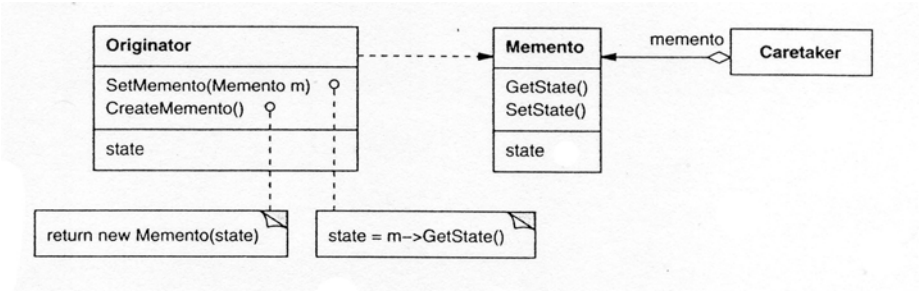
รูปที่ 2.17 โครงสร้างของรูปแบบการออกแบบอิเทอเรเตอร์ [1]

คลาส ConcreteMediator เป็นคลาสศูนย์กลางที่ใช้จัดการการติดต่อกันของคลาส ConcreteColleague โดยคลาส ConcreteColleague แต่ละคลาสไม่ต้องอ้างถึงกัน

2.1.1.3.6 มีเมนโต (Memento)

รูปแบบการออกแบบมีเมนโต ใช้สำหรับบันทึกสถานะภายในของวัตถุไว้ภายนอกวัตถุ และสามารถนำกลับคืนได้ดังเดิมโดยไม่ทำลายความเป็นส่วนตัวของวัตถุ

โดยปกติแล้ว วัตถุจะมีการป้องกันการเข้าถึงข้อมูลภายในของวัตถุตามหลักของการเขียนโปรแกรมเชิงวัตถุ แต่ในกรณีที่ต้องการการเก็บค่าภายใน เช่น ในกรณีของการทำกลับ (Undo) รูปแบบการออกแบบนี้จะสามารถทำให้สามารถเก็บข้อมูลของวัตถุไว้ภายนอกได้ในรูปของวัตถุอื่น และทำให้วัตถุนี้ไม่สามารถเข้าถึงข้อมูลภายในได้เพื่อป้องกันความเป็นส่วนตัวของวัตถุโดยให้เขตข้อมูลและเมท็อดทั้งหมดเป็นแบบส่วนตัว มีเพียงวัตถุของคลาสเจ้าของเท่านั้นที่เข้าถึงได้ โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.18



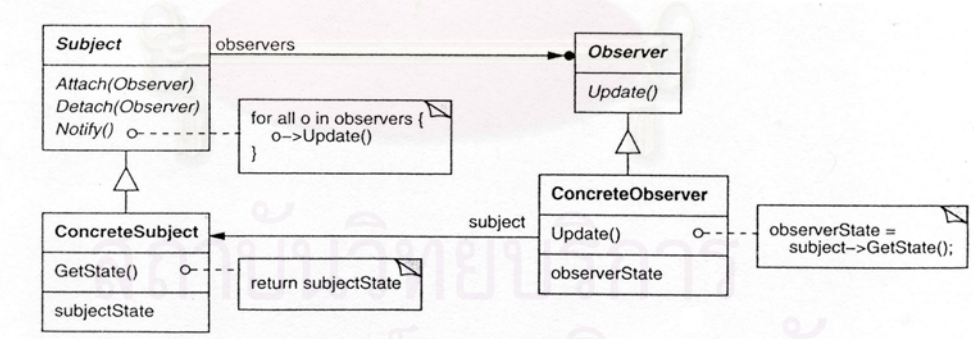
รูปที่ 2.18 โครงสร้างของรูปแบบการออกแบบมีเมนโต [1]

คลาส Originator ใช้เมท็อด CreateMemento() สำหรับสร้างวัตถุของคลาส Memento เพื่อใช้เก็บข้อมูลไว้ภายนอกวัตถุ และคืนค่ากลับให้วัตถุโดยใช้ เมท็อด SetMemento(Memento m)

2.1.1.3.7 ออบเซิร์ฟเวอร์ (Observer)

รูปแบบการออกแบบออบเซิร์ฟเวอร์ ใช้สำหรับกำหนดความเกี่ยวข้องของวัตถุหนึ่งต่อหลายๆ วัตถุ โดยเมื่อสถานะของวัตถุนั้นๆ เปลี่ยน ทุกวัตถุที่เกี่ยวข้องจะได้รับแจ้งเตือน

รูปแบบการออกแบบนี้ทำให้การผูกติดกันของวัตถุสามารถทำได้ขณะโปรแกรมทำงาน จึงทำให้คลาสที่เกี่ยวข้องเป็นอิสระจากกันมากขึ้นโดยไม่จำเป็นต้องผูกติดกันแต่แรก ทั้งนี้เพื่อให้การนำกลับมาใช้ใหม่ของคลาสที่เกี่ยวข้องทำได้สะดวกยิ่งขึ้น โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.19



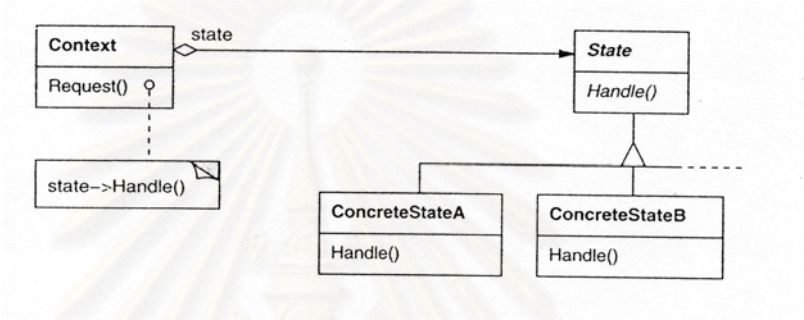
รูปที่ 2.19 โครงสร้างของรูปแบบการออกแบบออบเซิร์ฟเวอร์ [1]

คลาส Subject เป็นคลาสที่ถูกสังเกตความเปลี่ยนแปลง และคลาส Observer เป็นคลาสที่คอยสังเกตความเปลี่ยนแปลง เมื่อคลาส Subject มีการเปลี่ยนแปลง จะทำการเรียกเมท็อด Notify() เพื่อแจ้งเตือนวัตถุของคลาส Observer ทุกวัตถุที่คอยสังเกตการณ์อยู่ ซึ่งเก็บค่าวัตถุเหล่านี้ไว้ในเขตข้อมูล observers โดยเรียกเมท็อด Update() ของคลาส Observer ทุกคลาส

2.1.1.3.8 สเตท (State)

รูปแบบการออกแบบสเตทใช้ทำให้วัตถุเปลี่ยนพฤติกรรมเมื่อสถานะภายในเปลี่ยน จะปรากฏโดยการเปลี่ยนคลาสทำงาน

รูปแบบการออกแบบนี้ทำให้การทำงานของคลาสที่ขึ้นกับสถานะภายในแยกการทำงานแต่ละสถานะเป็นคลาสที่อิสระจากกัน โดยสามารถปรับเปลี่ยนการทำงานขณะโปรแกรมทำงานได้หลากหลาย และทำให้การทำงานซึ่งอาจไม่เกี่ยวข้องกันไม่ผูกติดกัน โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.20



รูปที่ 2.20 โครงสร้างของรูปแบบการออกแบบสเตท [1]

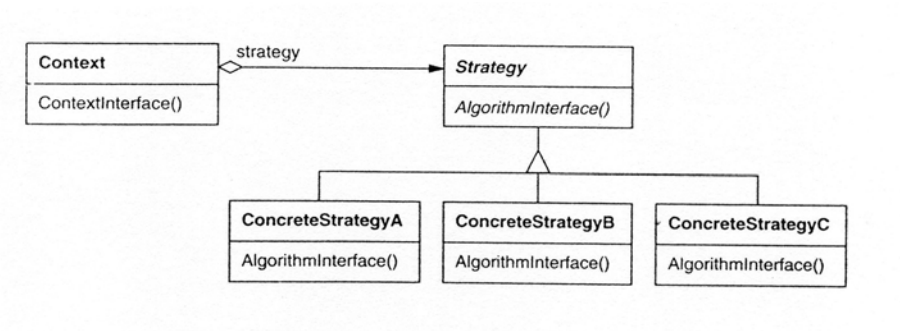
คลาส Context มีเขตข้อมูลของคลาส State ซึ่งใช้เรียกเมท็อด Handle() ในการทำงาน โดยเมื่อเปลี่ยนแปลงค่าของเขตข้อมูลนี้จะเป็นการเปลี่ยนเมท็อดทำงานเป็นเมท็อดของคลาสใหม่ที่เปลี่ยนแปลง

2.1.1.3.9 สเทรทิจี (Strategy)

รูปแบบการออกแบบสเทรทิจี ใช้เป็นกลยุทธ์ในการเลือกขั้นตอนการดำเนินงาน (Algorithm) ในเวลาโปรแกรมทำงานโดยจะระบุแต่ละขั้นตอนการดำเนินงานไว้ในแต่ละคลาสโดยมีตัวต่อประสานเหมือนกันซึ่งสามารถเปลี่ยนไปมาได้

รูปแบบการออกแบบนี้ทำให้การทำงานของคลาสที่สามารถเปลี่ยนแปลงการทำงานได้เป็นคลาสที่อิสระจากกัน โดยสามารถปรับเปลี่ยนการทำงานขณะโปรแกรมทำงานได้หลากหลาย และทำให้ขั้นตอนการดำเนินงานซึ่งอาจไม่เกี่ยวข้องกันไม่ผูกติดกัน รวมถึงทำให้โครงสร้างข้อมูลที่ใช้เฉพาะขั้นตอนการดำเนินงานหนึ่งอยู่ในคลาสของขั้นตอนการดำเนินงานนั้นๆ เท่านั้น โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.21

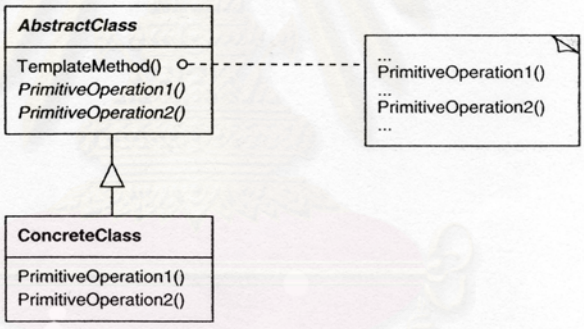
คลาส Context มีเขตข้อมูลของคลาส Strategy ซึ่งมี เมท็อด AlgorithmInterface() อยู่ โดยเมื่อเปลี่ยนแปลงค่าของเขตข้อมูลนี้จะเป็นการเปลี่ยนเมท็อดทำงานเป็นเมท็อดของคลาสใหม่ที่เปลี่ยนแปลง



รูปที่ 2.21 โครงสร้างของรูปแบบการออกแบบสเทรทิจี [1]

2.1.1.3.10 เทมเพลตเมท็อด (Template method)

รูปแบบการออกแบบเทมเพลตเมท็อด ใช้สำหรับสร้างเมท็อดซึ่งมีการกำหนดการทำงานหลักไว้แล้ว แต่ยังมีบางส่วนของเมท็อดที่ยังไม่สามารถระบุการทำงานที่แน่นอนได้ รูปแบบการออกแบบนี้จะทำให้เมท็อดนั้นสามารถระบุและเปลี่ยนแปลงการทำงานในย่อยเหล่านั้นได้ในภายหลังรวมถึงสามารถระบุการทำงานย่อยได้หลายแบบอีกด้วย โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.22



รูปที่ 2.22 โครงสร้างของรูปแบบการออกแบบเทมเพลตเมท็อด [1]

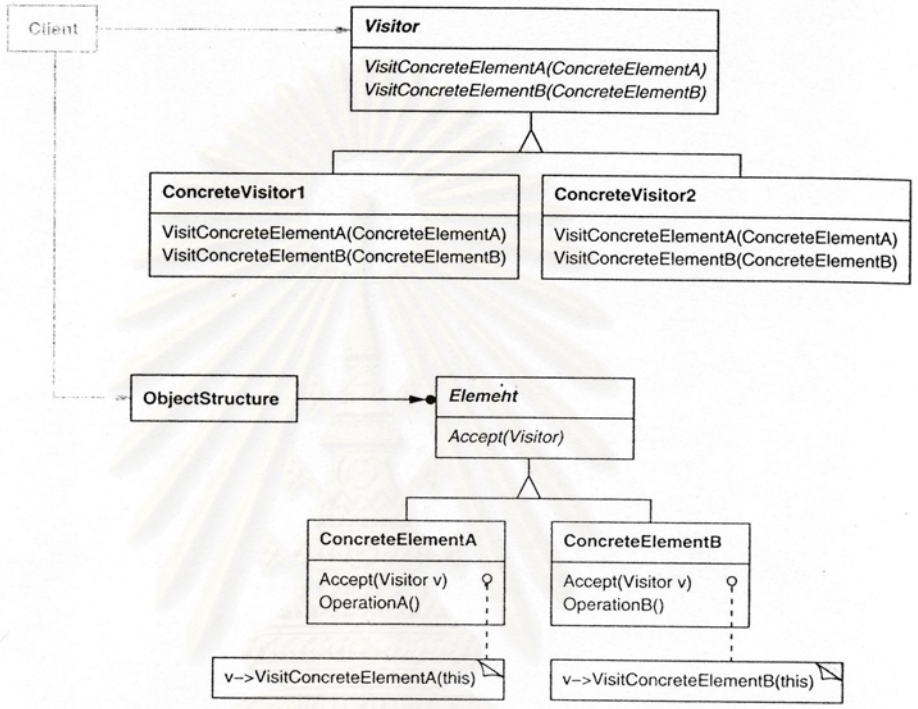
คลาส AbstractClass เป็นคลาสนามธรรมที่มีการกำหนดการทำงานของเมท็อด TemplateMethod() ไว้แล้ว ซึ่งมีการเรียกใช้เมท็อดนามธรรม PrimitiveOperation() ในการทำงานด้วย โดยให้คลาสลูก ConcreteClass เป็นผู้กำหนดการทำงานของเมท็อด PrimitiveOperation() ในภายหลัง

2.1.1.3.11 วิสิเตอร์ (Visitor)

รูปแบบการออกแบบวิสิเตอร์ ใช้สำหรับเพิ่มการปฏิบัติการใหม่ลงบนโครงสร้างของวัตถุเดิม โดยไม่มีการเปลี่ยนแปลงของเดิมที่มีอยู่

โครงสร้างวัตถุเดิมที่มีอยู่อาจจะมีตัวต่อประสานซึ่งแตกต่างกัน รวมถึงการปฏิบัติการใหม่ที่ต้องการจะกระทำลงบนคลาสย่อยบางคลาสเท่านั้น แต่ไม่ต้องการปรับ

เปลี่ยนโครงสร้างวัตถุเดิม รูปแบบการออกแบบนี้จะแยกการปฏิบัติการใหม่เป็นโครงสร้างของคลาสต่างหาก โดยทำให้คลาสเดิมมีเมท็อดที่ใช้ในการรับวัตถุของคลาสนี้เท่านั้น การกระทำดังนี้ทำให้การเพิ่มการปฏิบัติการกับคลาสเดิมสามารถทำได้โดยไม่ต้องเปลี่ยนแปลงโครงสร้างของคลาสเดิม โครงสร้างของรูปแบบการออกแบบนี้ปรากฏในรูป 2.23



รูปที่ 2.23 โครงสร้างของรูปแบบการออกแบบวิสิเตอร์ [1]

คลาส Element เป็นโครงสร้างของคลาสที่มีอยู่แล้วและต้องการเพิ่มการปฏิบัติการใหม่ลงไปโดยไม่ต้องการเปลี่ยนแปลงโครงสร้างของคลาสที่มีอยู่ รูปแบบการออบแบบนี้จะเพิ่มเมท็อด acceptVisitor(Visitor v) ลงไป แทนที่จะเพิ่มเมท็อดใหม่ลงไปตรงๆ โดยเมท็อดนี้จะรับวัตถุของคลาส Visitor และสั่งการทำงานเพิ่มเติมที่เตรียมไว้สำหรับคลาสตนเองเท่านั้น ซึ่งการทำงานที่เพิ่มเติมจะไประบุที่คลาส ConcreteVisitor แทน

2.1.2 **ส่วนประกอบซอฟต์แวร์** [11]

ในอดีตนั้นได้เกิดปัญหาการพัฒนาโปรแกรม กล่าวคือไม่สามารถควบคุมค่าใช้จ่าย และเวลาในการผลิตซอฟต์แวร์ได้ ทั้งนี้เนื่องจากซอฟต์แวร์เริ่มมีความซับซ้อนมากกว่าแต่ก่อน การพัฒนาซอฟต์แวร์ต้องเสียเวลาในการสร้างและทดสอบใหม่มากขึ้น ขณะเดียวกันวงการอุตสาหกรรมทางอิเล็คโทรนิค เช่น รถยนต์ โทรทัศน์ ต่างก็เผชิญกับการผลิตที่มีความซับซ้อนเช่นกัน แต่ก็ไม่ได้ก่อให้เกิดปัญหาให้แก่อุตสาหกรรมเหล่านี้ ทั้งนี้เนื่องจากอุตสาหกรรมเหล่านี้มีการออกแบบ

ชิ้นส่วนอุปกรณ์เป็นแบบส่วนประกอบดังเช่น การสร้างเครื่องคอมพิวเตอร์ต้องอาศัยตัวต่อประสานชิ้นส่วนอุปกรณ์เข้าด้วยกัน อาทิ หน่วยความจำ หน่วยประมวลผลกลาง บัส แผงแป้นอักขระ หน่วยขับแผ่นบันทึก จอภาพ โดยอุปกรณ์ต่างๆ สามารถเชื่อมต่อกันได้เพราะมีมาตรฐานในตัวต่อประสาน ซึ่งทำให้สามารถนำส่วนประกอบต่างๆ จากผู้ผลิตหลายรายมาสร้างเป็นระบบได้ เป็นผลให้สามารถประหยัดงบประมาณและเวลาในการพัฒนาเป็นอย่างมาก

อาศัยหลักการคล้ายกันนี้ ส่วนประกอบซอฟต์แวร์คือซอฟต์แวร์ที่ถูกออกแบบสำหรับเป็นส่วนประกอบ (Component) ของโปรแกรมซึ่งสามารถนำมาประกอบเป็นโปรแกรมได้ โดยส่วนประกอบซอฟต์แวร์นี้จะถูกสร้างเพียงครั้งเดียว และสามารถนำไปใช้ในการสร้างโปรแกรมอื่นได้ในคราวต่อไป โดยในการสร้างส่วนประกอบซอฟต์แวร์นี้จะมีข้อกำหนดสำหรับการสร้าง ซึ่งส่วนประกอบซอฟต์แวร์แต่ละแบบก็จะมีข้อกำหนดในการสร้างต่างกันไป เช่น ส่วนประกอบซอฟต์แวร์แบบจาวาบีน (JavaBean) ซึ่งจะกล่าวถึงในหัวข้อถัดไป

ส่วนประกอบซอฟต์แวร์จะมีส่วนการติดต่อกับผู้ใช้และส่วนการทำงานอยู่ 2 ชุด ชุดแรกใช้ในขณะออกแบบ ซึ่งปรากฏในเครื่องมือช่วยการพัฒนาแบบเห็นภาพให้ผู้พัฒนาโปรแกรมใช้งานและมีการตอบสนองขณะพัฒนาโปแกรม อีกชุดหนึ่งใช้ขณะโปรแกรมทำงาน เป็นการทำงานของส่วนประกอบซอฟต์แวร์จริง โดยการทำงานของส่วนประกอบซอฟต์แวร์นี้จะกลายเป็นส่วนหนึ่งของการทำงานของโปรแกรมประยุกต์ที่พัฒนานั้น

2.1.3 **จาวาบีน** [7]

จาวาบีนได้รับการพัฒนาขึ้นในจาวารุ่นที่ 1.1 โดยมีจุดมุ่งหมายเพื่อรองรับการทำงานสถาปัตยกรรมส่วนประกอบซอฟต์แวร์ โดยมีแพคเกจ (Package) ของจาวาชื่อ java.beans ซึ่งสนับสนุนการพัฒนาส่วนประกอบซอฟต์แวร์ [8] โดยการเขียนจาวาบีนต่างจากการเขียนจาวาตรงที่ต้องมีการเรียกใช้คลาส และตัวต่อประสานของแพคเกจ java.beans และต้องทำตามข้อกำหนดในการเขียนจาวาบีน (Javabean API specifications) [9] ซึ่งกำหนดโดยบริษัทซัน ไมโครซิสเต็มส์

จาวาบีนมีคุณสมบัติหลัก ดังนี้

1. การพินิจภายใน (Introspection) ทำให้เครื่องมือช่วยการพัฒนาสามารถวิเคราะห์การทำงานของบีน และสามารถรู้ถึงเมท็อดและเขตข้อมูลในบีนได้

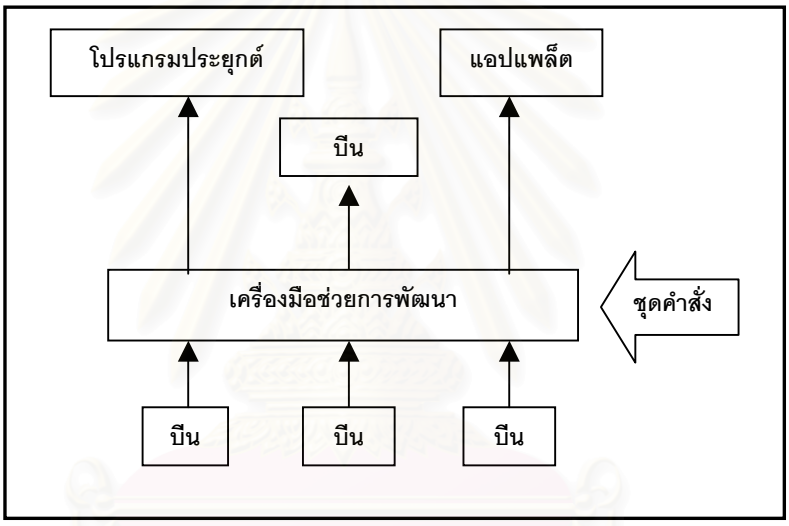
2. การปรับแต่ง (Customization) ทำให้ผู้ใช้สามารถปรับแต่งคุณลักษณะหรือพฤติกรรมของบีนได้

3. เหตุการณ์ (Events) ทำให้บีนสามารถสร้างเหตุการณ์ได้ รวมถึงแจ้งแก่เครื่องมือช่วยการพัฒนาว่าบีนนั้นๆ สามารถสร้างเหตุการณ์และรับมือกับเหตุการณ์ใดได้บ้าง

4. คุณลักษณะ (Properties) ทำให้ผู้ใช้ปรับเปลี่ยนข้อมูลของบีนได้อย่างเป็นระบบ และสนับสนุนการปรับแต่ง

5. การมีอยู่ (Persistence) ทำให้เครื่องมือช่วยการพัฒนาสามารถบันทึกและส่งคืนข้อมูลของบีนที่ได้รับการปรับแต่งแล้วได้

การนำจาวาบีนไปใช้งานเพื่อสร้างโปรแกรมประยุกต์ แอปเพล็ต (Applet) หรือบีนนั้น ทำโดยเรียกใช้บีนและเชื่อมต่อบีนเข้าด้วยกันหรืออาจเขียนชุดคำสั่งเพิ่มเติมเพื่อควบคุมโปรแกรมให้เหมาะสมสำหรับการใช้งานตามที่ต้องการ การทำงานดังกล่าวจะทำโดยอาศัยเครื่องมือช่วยการพัฒนาดังรูปที่ 2.24



รูปที่ 2.24 แสดงการนำบีนไปใช้งานโดยเครื่องมือช่วยการพัฒนา

เครื่องมือช่วยการพัฒนาแบบเห็นภาพ (Visual tool) เป็นเครื่องมือช่วยการพัฒนาโปรแกรม ซึ่งสามารถมองเห็นรูปร่างลักษณะของโปรแกรมในรูปแบบกราฟฟิก (Graphic) และสามารถปรับเปลี่ยนคุณลักษณะของบีนในขณะออกแบบได้ (Design time) ภายในเครื่องมือช่วยการพัฒนาแบบเห็นภาพนั้น บีนจะอยู่ในรูปแบบของกล่องเครื่องมือ (Tool box) เมื่อต้องการสร้างโปรแกรมที่มีบางส่วนของโปรแกรมที่ได้ทำเป็นส่วนประกอบซอฟต์แวร์ไว้แล้ว ก็เพียงนำส่วนประกอบซอฟต์แวร์ ซึ่งอยู่ในรูปของบีนที่อยู่ในกล่องเครื่องมือมาลากแล้วปล่อย (Drag and drop) เพื่อเชื่อมต่อกันลงในฟอร์มสำหรับสร้างโปรแกรม รวมถึงสามารถเปลี่ยนแปลงคุณลักษณะเริ่มต้นของบีนนั้นให้เป็นไปตามที่ต้องการได้

2.2 งานวิจัยที่เกี่ยวข้อง

2.2.1 งานวิจัย “Framework Editor for Java (FRED)” [5] เป็นโครงการเพื่อพัฒนาโครงร่างสำหรับพัฒนาโปรแกรมในลักษณะของเครื่องมือแบบเคส (CASE tools) ที่สามารถนำการออกแบบกลับมาใช้ใหม่โดยมีตัวเรียบเรียงรูปแบบการออกแบบ (Pattern composer) สำหรับให้ผู้ใช้เลือกรูปแบบการออกแบบเพื่อนำมาสร้างโครงร่าง จากนั้นจะมีตัวเรียบเรียงโครงร่าง (Framework composer) เพื่อให้ผู้ใช้เลือกเพื่อสร้างโครงร่าง และมีตัวเรียบเรียงโปรแกรมประยุกต์ (Application composer) เพื่อให้ผู้ใช้นำโครงร่างไปสร้างโปรแกรมประยุกต์ตามต้องการ โดยโครงร่างนี้พัฒนาด้วยภาษาจาวา ซึ่งงานวิจัยนี้มีการใช้รูปแบบการออกแบบแต่อยู่ในรูปของโครงร่างที่จะใช้พัฒนาเป็นโปรแกรมประยุกต์ โดยจะเน้นที่การสร้างเครื่องมือที่จะสนับสนุนการนำเอารูปแบบการออกแบบไปใช้ในโครงร่าง

2.2.2 งานวิจัย “Framework Adaptive Composition Environment (FACE)” [6] เป็นการเสนอวิธีการนำรูปแบบการออกแบบมาใช้ในการพัฒนาโปรแกรม โดยใช้หลักการประกอบวัตถุเข้าด้วยกัน (Object composition) เพื่อให้ง่ายต่อการเข้าใจ และเมื่อพัฒนาโปแกรมเสร็จแล้วต่อไปในอนาคตสามารถปรับปรุงหรือเพิ่มเติมได้ง่าย แต่งานวิจัยนี้ไม่ได้นำการประกอบวัตถุนั้นไปใช้ในตัวแปลโปรแกรม (Compiler) หรือเครื่องมือช่วยสร้างแบบเห็นภาพ

2.2.3 โครงร่างและรูปแบบการออกแบบเพื่อพัฒนาโปรแกรมประยุกต์ (Framework and Design Patterns for Application Development) [10] เป็นงานวิจัยเพื่อพัฒนาโปรแกรมสำหรับช่วยในการสร้างโปรแกรมประยุกต์ มีลักษณะโดยรวมเป็นโปรแกรมแยกต่างหากจากเครื่องมือพัฒนาที่ช่วยในการสร้างชุดคำสั่งโครงร่าง โดยใส่คุณลักษณะเริ่มต้นที่ต้องการและโปรแกรมจะสร้างชุดคำสั่งกลับมาให้ผู้ใช้ โปรแกรมนี้สนับสนุนการใช้รูปแบบการออกแบบด้วยโดยเมื่อผู้ใช้ต้องการใช้ ซึ่งผู้ใช้จะเลือกรูปแบบการออกแบบที่ต้องการจากบัญชีรายชื่อรูปแบบการออกแบบให้เหมาะสมกับงานที่ต้องการ แล้วโปรแกรมจะมีกรอบข้อความขึ้นให้ใส่ข้อมูลอื่นๆ ที่จำเป็น แล้วจึงสร้างชุดคำสั่งออกมาให้ในรูปของแฟ้มข้อมูลของชุดคำสั่งเพื่อพัฒนาต่อไป ซึ่งงานวิจัยนี้เป็นการพัฒนาโครงร่างที่มีการนำเอารูปแบบการออกแบบเข้าไปใช้ช่วยในการสร้างชุดคำสั่ง ซึ่งไม่สามารถนำชุดคำสั่งที่ได้ไปพัฒนาโปรแกรมประยุกต์ต่อโดยใช้เครื่องมือช่วยการพัฒนาแบบเห็นภาพ ซึ่งเป็นที่นิยมในปัจจุบันได้ ทำให้ไม่สะดวกในการใช้งานจริง

บทที่ 3

การออกแบบซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

3.1 ภาพรวมของการออกแบบซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

ส่วนการติดต่อกับผู้ใช้และส่วนการทำงานของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบที่ทำการสร้างในวิทยานิพนธ์นี้เป็นส่วนการติดต่อกับผู้ใช้และส่วนการทำงานในขณะออกแบบเท่านั้น โดยจะไม่มีส่วนการติดต่อกับผู้ใช้และส่วนการทำงานในขณะโปรแกรมทำงาน เนื่องจากส่วนประกอบซอฟต์แวร์นี้จะทำหน้าที่ในการช่วยเขียนชุดคำสั่งในขณะออกแบบเพื่อสะดวกในการใช้รูปแบบการออกแบบเท่านั้น ไม่มีผลในการทำงานของโปรแกรม

ซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ แบ่งเป็น 2 ส่วนใหญ่ๆ คือ

1. ส่วนการติดต่อกับผู้ใช้ของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ซึ่งคือคุณลักษณะของส่วนประกอบซอฟต์แวร์นั้นๆ ที่ปรากฏในหน้าต่างคุณลักษณะของเครื่องมือช่วยการพัฒนาแบบเห็นภาพ

2. ส่วนการทำงานของรูปแบบการออกแบบ ซึ่งคือส่วนการทำงานของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบในขณะออกแบบ ในส่วนนี้แบ่งย่อยได้อีก 2 ส่วน คือ ส่วนการสร้างชุดคำสั่ง และส่วนจัดการแฟ้มข้อมูลชุดคำสั่ง โดยส่วนจัดการแฟ้มข้อมูลชุดคำสั่งมีหน้าที่หลักเพียงอ่านและเขียนแฟ้มข้อมูลเท่านั้น

ในการออกแบบส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ จะทำการออกแบบส่วนติดต่อกับผู้ใช้ก่อน ซึ่งในที่นี้ก็คือคุณลักษณะของส่วนประกอบซอฟต์แวร์ โดยจะกำหนดว่าต้องมีอะไรบ้าง และมีการเปลี่ยนแปลงกับชุดคำสั่งอย่างไรถ้ามีการเปลี่ยนแปลงคุณลักษณะแต่ละคุณลักษณะ ซึ่งจะนำไปสู่วิธีการสร้างชุดคำสั่งของรูปแบบการออกแบบ

การทำงานโดยรวมของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ คือ การช่วยสร้างชุดคำสั่งให้กับผู้เขียนโปรแกรม โดยจะทำการสร้าง เปลี่ยนแปลง หรือลบชุดคำสั่งเมื่อมีการเปลี่ยน

แปลงคุณลักษณะของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ทั้งนี้ผู้เขียนโปรแกรมจะต้องเลือกส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบที่ต้องการใช้ด้วย ดังแสดงในรูปที่ 3.1

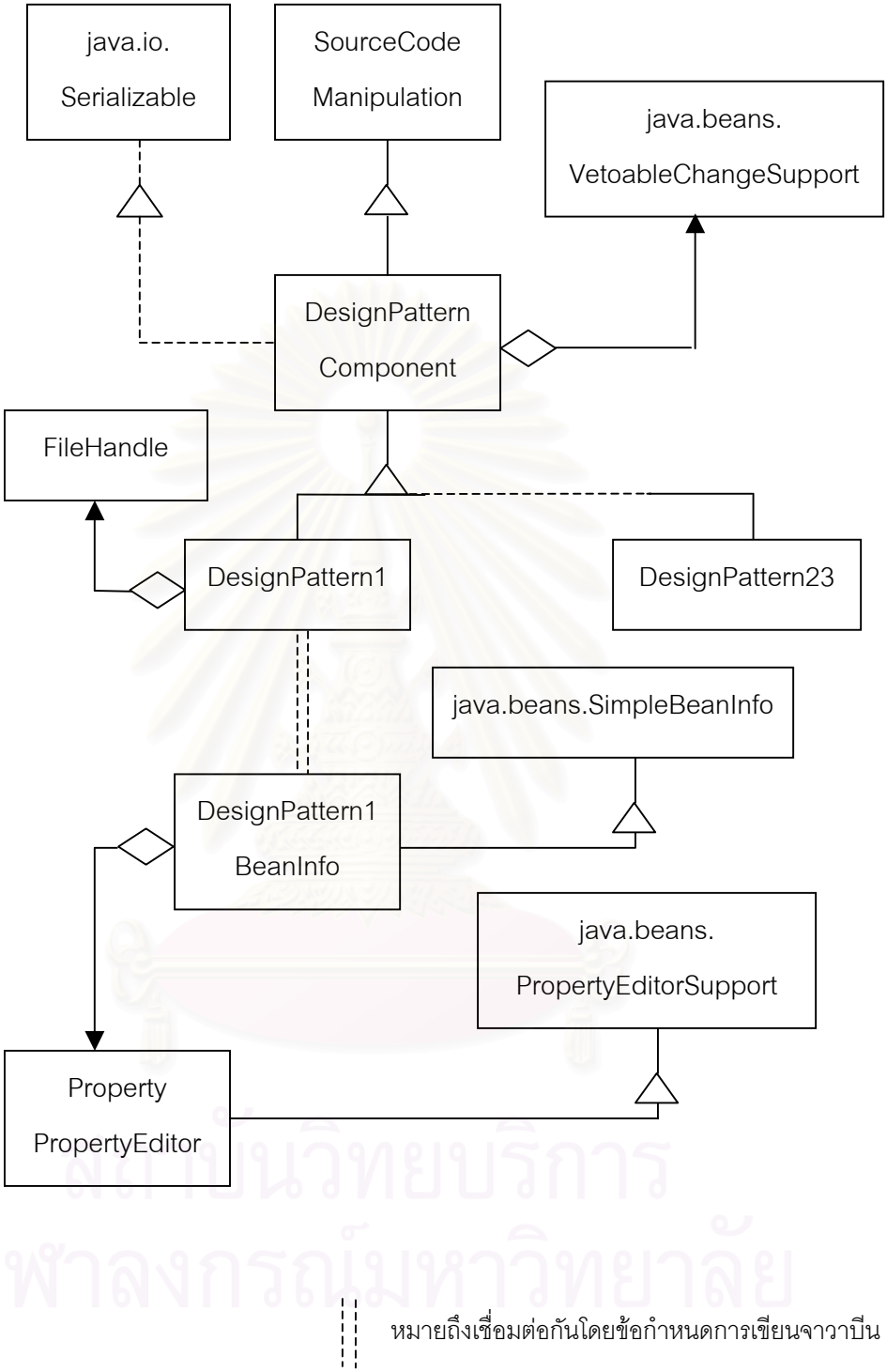
รูปที่ 3.1 การทำงานโดยรวมของซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

3.2 โครงสร้างคลาสของซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

ส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบที่ออกแบบในวิทยานิพนธ์นี้ คือคลาสที่ชื่อ DesignPattern เป็นจาวาบีน ซึ่งมีทั้งหมด 23 คลาส ตามจำนวนของรูปแบบการออกแบบของอิริก แกมมา [1] ได้แก่ คลาส DesignPattern ในรูปที่ 3.2 โดยใช้ชื่อของรูปแบบการออกแบบเป็นชื่อคลาส นอกจากนี้ ยังมีคลาสอีกคลาสหนึ่งซึ่งเป็นตัวรวบรวมสารสนเทศของบีนให้กับเครื่องมือช่วยการพัฒนาแบบเห็นภาพ ซึ่งเป็นคลาสชื่อเดียวกับบีนแล้วตามด้วยคำว่า BeanInfo ตามข้อกำหนดในการเขียนจาวาบีน [9]

3.2.1 **คลาส** SourceCodeManipulation

คลาสนี้ เป็นคลาสหลักของซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ มีเมท็อดต่างๆ ที่ใช้สำหรับจัดการสร้าง ลบ หรือเปลี่ยนแปลงชุดคำสั่งต่างๆ ของภาษาจาวาที่จำเป็นต้องใช้ในการสร้างชุดคำสั่งของรูปแบบการออกแบบ รายละเอียดของเมท็อดในคลาสนี้ดูได้ในภาคผนวก ข



รูปที่ 3.2 โครงสร้างคลาสของซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบและคลาสของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

3.2.2 **คลาส** DesignPatternComponent

คลาสนี้ เป็นคลาสนามธรรมที่เป็นคลาสหลักของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบทุกรูปแบบที่ออกแบบในวิทยานิพนธ์นี้ คลาสนี้จะรวมความสามารถ (aggregate) ของคลาส java.beans.VetoableChangeSupport เข้าไว้ด้วย ซึ่งทำให้คลาสนี้สามารถตรวจสอบค่าข้อมูลรับเข้าจากเครื่องมือช่วยการพัฒนาแบบเห็นภาพ ตามข้อกำหนดในการเขียนจาวาบีน [9] ได้ นอกจากนี้คลาสนี้ได้ทำการอิมพลีเมนต์ (implements) ตัวต่อประสาน java.io.Serializable ตามข้อกำหนดในการเขียนจาวาบีนเช่นกัน เพื่อให้จาวาบีนสามารถทำกระบวนการการมีอยู่ตามคุณสมบัติหลักของจาวาบีนได้

คลาสนี้ มีเมท็อดที่สำคัญอยู่ 3 เมท็อด คือ adjustStructure() deleteStructure() และ applyEngine() ซึ่ง 2 เมท็อดแรกใช้สำหรับ สร้าง เปลี่ยนแปลง และ ลบ ชุดคำสั่งของรูปแบบการออกแบบแต่ละรูปแบบตามรูปที่ 3.2 ซึ่งได้กำหนดไว้เป็นเมท็อดนามธรรม ซึ่งคลาสลูกของคลาสนี้ก็คือส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบในวิทยานิพนธ์นี้ จะเป็นผู้กำหนดการสร้างและลบ ส่วนเมท็อด applyEngine() นั้น จะเป็นกลไกหลักในการสร้างชุดคำสั่ง โดยเมท็อดนี้จะตรวจสอบค่า คุณลักษณะ Apply และ FinalVersion ถ้าเป็นไปตามข้อกำหนดในหัวข้อ 3.2 ก็จะทำการเปลี่ยนแปลงชุดคำสั่งโดยจะไปเรียกเมท็อด adjustStructure() และ deleteStructure() ให้ทำงานอีกต่อหนึ่ง

3.2.3 **คลาส** FileHandle

คลาสนี้ เป็นคลาสที่ใช้จัดการแฟ้มข้อมูลของชุดคำสั่งที่จะใช้รูปแบบการออกแบบแต่ละแฟ้มโดยตรง โดยคลาส DesignPattern เป็นคนเรียกใช้ โดยใช้กี่วัตถุของคลาส ขึ้นอยู่กับชนิดของรูปแบบการออกแบบ

3.2.4 **คลาส** DesignPattern

คลาสนี้ คือส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบทุกรูปแบบ ที่ออกแบบในวิทยานิพนธ์นี้ 23 รูปแบบ โดยชื่อที่ใช้จริงๆ ของคลาสคือชื่อของรูปแบบการออกแบบ เช่น ObserverPattern เป็นต้น คลาสนี้จะระบุรายละเอียดการ สร้าง เปลี่ยนแปลง ลบ ชุดคำสั่งของรูปแบบการออกแบบ โดยระบุการทำงานในเมท็อด adjustStructure() และ deleteStructure() รายละเอียดการสร้างชุดคำสั่งของแต่ละรูปแบบการออกแบบจะอธิบายอย่างละเอียดในบทที่ 4

3.2.5 **คลาส** DesignPatternBeanInfo

คลาสนี้ คือคลาสที่รวบรวมสารสนเทศของบีนของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบแต่ละรูปแบบที่อยู่ในวิทยานิพนธ์นี้ทั้ง 23 รูปแบบ สำหรับเป็นข้อมูลให้กับเครื่องมือช่วย

การพัฒนาแบบเห็นภาพ ซึ่งจะเป็นคลาสชื่อเดียวกับบีนแล้วตามด้วยคำว่า BeanInfo ภายใต้ข้อกำหนดการเขียนจาวาบีน [9] เช่น ObserverPatternBeanInfo โดยสารสนเทศนี้ประกอบด้วยชื่อของคลาสส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ชื่อของคุณลักษณะแต่ละคุณลักษณะ ชนิดของคุณลักษณะ คลาส PropertyPropertyEditor ที่เกี่ยวข้องกับคุณลักษณะนั้น (ถ้ามี) และสัญรูป (Icon) ของส่วนประกอบซอฟต์แวร์ ที่จะปรากฏในกล่องเครื่องมือในเครื่องมือช่วยการพัฒนาแบบเห็นภาพ

3.2.6 **คลาส** PropertyPropertyEditor

คลาสนี้ ใช้สำหรับเป็นส่วนติดต่อกับผู้ใช้เพิ่มเติมนอกเหนือจากที่เครื่องมือช่วยการพัฒนาแบบเห็นภาพเตรียมไว้ให้ในการรับเข้าข้อมูลของคุณลักษณะที่ซับซ้อนกว่าการกรอกข้อความปกติ โดยจะมีคลาสนี้หนึ่งคลาสต่อข้อมูลที่รับเข้าหนึ่งข้อมูล โดยชื่อที่ใช้จริงๆ ของคลาสคือ ชื่อของคุณลักษณะตามด้วยคำว่า PropertyEditor เช่น MethodPropertyEditor เป็นต้น คลาสนี้จะทำการตรวจสอบข้อมูลรับเข้าเบื้องต้นก่อนนำเข้าด้วย

3.3 **ส่วนการติดต่อกับผู้ใช้ของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ**

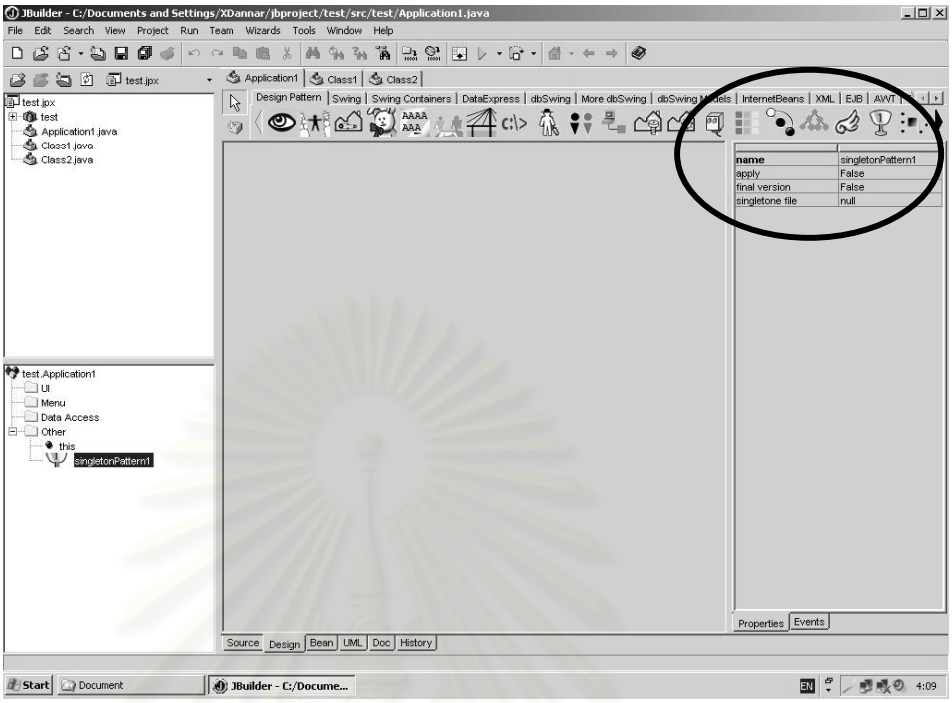
ส่วนติดต่อกับผู้ใช้ที่ปรากฏในเครื่องมือช่วยการพัฒนาแบบเห็นภาพทั่วไปนั้น ใช้กลไกของเครื่องมือช่วยการพัฒนาแบบเห็นภาพ ร่วมกับสถาปัตยกรรมของส่วนประกอบซอฟต์แวร์ซึ่งในที่นี้ใช้จาวาบีน ในการสร้าง โดยส่วนประกอบซอฟต์แวร์ทั้งหมดจะปรากฏเป็นสัญรูปในกล่องเครื่องมือของเครื่องมือช่วยการพัฒนาแบบเห็นภาพ โดยได้กำหนดสัญรูปแทนรูปแบบการออกแบบแต่ละรูปแบบดังตารางที่ 3.1

เมื่อนำส่วนประกอบซอฟต์แวร์หรือบีน จากกล่องเครื่องมือของเครื่องมือช่วยการพัฒนาแบบเห็นภาพมาลากแล้วปล่อยลงบนฟอร์ม เราสามารถแก้ไขคุณลักษณะของบีนได้ ซึ่งเครื่องมือช่วยการพัฒนาแบบเห็นภาพจะแสดงหน้าต่างของคุณลักษณะทั้งหมดของบีนนั้นๆ ที่อนุญาตให้แก้ไขได้ขึ้นมา ซึ่งคุณลักษณะใดที่ยอมให้แก้ไขได้หรือไม่ได้นั้น จะต้องทำการกำหนดไว้ในตอนสร้างบีนขึ้นมา ตามข้อกำหนดในการเขียนจาวาบีน [9] ซึ่งกำหนดโดยบริษัท ซัน ไมโครซิสเต็มส์ หากอนุญาตให้แก้ไขได้ เครื่องมือช่วยการพัฒนาแบบเห็นแบบจะแสดงขึ้นมาในหน้าต่างสำหรับแก้ไข ถ้าไม่อนุญาต เครื่องช่วยสร้างแบบเห็นภาพจะไม่แสดงให้เห็น

จากรูปที่ 3.3 ทางด้านขวาของรูป แสดงให้เห็นหน้าต่างที่ใช้ในการแก้ไขคุณลักษณะของบีน

ตารางที่ 3.1 สัญรูปของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

สัญรูป	รูปแบบการออกแบบ
	แอ็บสแทรคท์แฟคทอรี
	บิวเดอร์
	แฟคทอรีเมทธอด
	โปรโตไทพ์
	ซิงเกิลตอน
	อแด็พเตอร์
	บริดจ์
	คอมโพสิต
	เดคอเรเตอร์
	ฟาซาด
	ไฟลเวจ
	พร็อกซี
	เชน ออฟ เรสปอนสิบิลิตี
	คอมมานด์
	อินเตอร์พรีเตอร์
	อิเทอเรเตอร์
	เมดิเอเตอร์
	มีเมนโต
	ออบเซิร์ฟเวอร์
	สเตท
	สเทรทิจี
	เทมเพลตเมทธอด
	วิสิเตอร์



รูปที่ 3.3 ตัวอย่างหน้าต่างการแก้ไขคุณลักษณะของบีน

ในการออกแบบส่วนการติดต่อกับผู้ใช้ของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบนั้น มีการแบ่งคุณลักษณะที่อนุญาตให้แก้ไขได้เป็น 6 ชนิด ได้แก่

1. ชื่อแฟ้มข้อมูลคลาสหลักของชุดคำสั่งที่ต้องการใช้รูปแบบการออกแบบ คุณลักษณะนี้จะระบุแฟ้มข้อมูลเพียงหนึ่งแฟ้มข้อมูลที่เป็นคลาสแม่ หรือคลาสหลักของรูปแบบการออกแบบ

2. ชื่อแฟ้มข้อมูลคลาสย่อยของชุดคำสั่งที่ต้องการใช้รูปแบบการออกแบบ คุณลักษณะนี้จะระบุแฟ้มข้อมูลหลายๆ แฟ้มที่เป็นคลาสย่อยของรูปแบบการออกแบบ

3. ชื่อเมท็อดที่ต้องการใช้ในรูปแบบการออกแบบ เมท็อดนี้จะไปปรากฏในชุดคำสั่งในส่วนของรูปแบบการออกแบบ ซึ่งอาจมีเพียงเมท็อดเดียว หลายๆ เมท็อด หรืออาจจะไม่มีเลยก็ได้ขึ้นกับรูปแบบการออกแบบที่เลือกมาใช้เช่นกัน

4. คุณลักษณะอื่นๆ ที่เป็นคุณลักษณะเฉพาะของบางรูปแบบการออกแบบ คุณลักษณะนี้มักปรากฏในรูปของชื่อบางอย่างที่จำเป็นต้องใช้ภายในรูปแบบการออกแบบเพื่อใช้ในการส่งข้อความ (Message) ระหว่างคลาส เช่น ชื่อคำสั่งของรูปแบบการออกแบบคอมมานด์

5. คุณลักษณะที่บอกว่า ต้องการให้ส่วนประกอบซอฟต์แวร์สร้างชุดคำสั่งลงในโปรแกรมเลยหรือไม่ คุณลักษณะนี้ชื่อว่า Apply โดยจะเป็นคุณลักษณะแบบบูล (Boolean) มีค่าเริ่มต้นเป็นเท็จ

6. คุณลักษณะที่บอกว่าโปรแกรมที่กำลังพัฒนานั้น จะนำไปใช้งานจริงแล้วหรือว่ายังอยู่ระหว่างการพัฒนา คุณลักษณะนี้ชื่อว่า FinalVersion โดยจะเป็นคุณลักษณะแบบบูล มีค่าเริ่มต้นเป็นเท็จ คุณลักษณะนี้เสมือนตัวป้องกันการสร้างชุดคำสั่ง หากกำหนดคุณลักษณะนี้ให้เป็นจริง หมายถึงจะนำไปใช้งานจริงแล้ว จะไม่อนุญาตให้ทำการเปลี่ยนแปลงคุณลักษณะอื่นๆ ได้ รวมถึงส่วนการสร้างชุดคำสั่งตามหัวข้อ 3.2 จะไม่ทำงานด้วย หากเป็นเท็จหมายถึงยังอยู่ระหว่างการพัฒนา ส่วนประกอบซอฟต์แวร์จะทำงานตามปกติ สาเหตุของการที่ต้องมีคุณลักษณะนี้ เพื่อเป็นการป้องกันการสร้างชุดคำสั่งขณะใช้งานจริง

อนึ่ง ส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบแต่ละรูปแบบ ไม่จำเป็นต้องมีคุณลักษณะครบทั้ง 6 ชนิดก็ได้ รวมถึงสามารถมีคุณลักษณะชนิดใดชนิดหนึ่งมากกว่าหนึ่งคุณลักษณะได้ ยกเว้นคุณลักษณะ Apply และคุณลักษณะ FinalVersion ซึ่งต้องมีอยู่ทุกรูปแบบการออกแบบ และต้องมีเพียงอย่างละหนึ่งคุณลักษณะเท่านั้น

3.4 ส่วนการสร้างชุดคำสั่งของรูปแบบการออกแบบ

ส่วนของการสร้างชุดคำสั่งนี้จะทำงานก็ต่อเมื่อมีการกำหนดให้คุณลักษณะ FinalVersion เป็นเท็จ มิฉะนั้นจะไม่ทำการใดๆ การทำงานของส่วนการสร้างชุดคำสั่งมีกฎในการสร้างชุดคำสั่งเมื่อมีการเปลี่ยนแปลงคุณลักษณะ ดังนี้

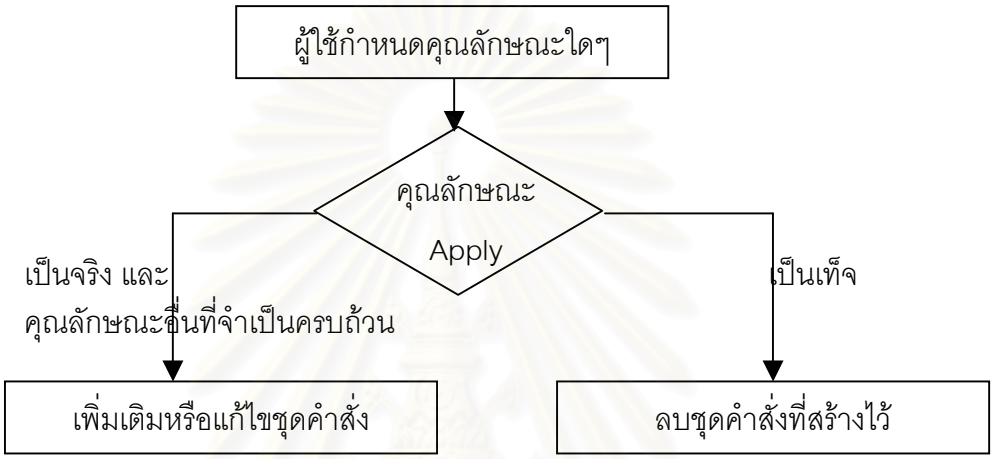
1. เมื่อมีการกำหนดคุณลักษณะแฟ้มข้อมูลของชุดคำสั่ง ถ้าหากแฟ้มข้อมูลนั้นยังไม่ได้ถูกสร้าง ส่วนประกอบซอฟต์แวร์จะทำการสร้างชุดคำสั่งภาษาจาวา โดยสร้างเป็นคลาสว่างที่มีตัวสร้างสาธารณะ (Public Constructor)

2. เมื่อเปลี่ยนแปลงคุณลักษณะ Apply ให้เป็นจริง ส่วนประกอบซอฟต์แวร์จะทำการสร้างชุดคำสั่งตามคุณลักษณะอื่นๆ ที่กำหนดไว้ หากผู้ใช้ยังไม่ได้กำหนดคุณลักษณะอื่นๆ ที่จำเป็นให้ครบถ้วน จะไม่สามารถเปลี่ยนแปลงคุณลักษณะ Apply ให้เป็นจริงได้

3. เมื่อเปลี่ยนแปลงคุณลักษณะ Apply ให้เป็นเท็จ จะทำการลบชุดคำสั่งที่สร้างขึ้นออก ถ้าผู้ใช้ยังไม่ได้ทำการเปลี่ยนแปลงใดๆ หากผู้ใช้ได้เพิ่มเติมหรือแก้ไขชุดคำสั่งที่ส่วนประกอบซอฟต์แวร์ได้สร้างไว้ ชุดคำสั่งส่วนนั้นจะคงไว้ ยกเว้นการเปลี่ยนแปลงคำอธิบายชุดคำสั่ง (Comment) ทั้งนี้ เพื่อเป็นการป้องกันการลบชุดคำสั่งของผู้ใช้ที่สร้างขึ้นเองโดยไม่ได้ตั้งใจ

4. เมื่อมีการเปลี่ยนคุณลักษณะใดๆ ในขณะที่คุณลักษณะ Apply เป็นจริง ส่วนประกอบซอฟต์แวร์จะทำการเปลี่ยนแปลงชุดคำสั่ง ให้เป็นไปตามที่กำหนดตามคุณลักษณะที่กำหนดขึ้นใหม่

5. เมื่อเปลี่ยนแปลงคุณลักษณะ FinalVersion ส่วนประกอบซอฟต์แวร์จะไม่ทำการใดๆ กับชุดคำสั่ง



รูปที่ 3.4 หลักการทำงานของส่วนสร้างชุดคำสั่งเมื่อเปลี่ยนแปลงคุณลักษณะใดๆ

จากรูปที่ 3.4 จะเห็นว่าการเพิ่มเติม แก้ไข หรือลบชุดคำสั่ง ผู้ใช้จะสั่งการโดยผ่านคุณลักษณะ Apply หากกำหนดให้คุณลักษณะ Apply เป็นจริงและกำหนดคุณลักษณะอื่นๆ ที่จำเป็นในการสร้างชุดคำสั่งของรูปแบบการออกแบบครบถ้วน ส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบจะทำการเพิ่มเติม หรือแก้ไขชุดคำสั่ง ตามกฎการสร้างชุดคำสั่งข้อ 2 และ 4 หากกำหนดให้คุณลักษณะ Apply เป็นเท็จ จะทำการลบชุดคำสั่งที่สร้างไว้ออก ตามกฎการสร้างชุดคำสั่งข้อ 3

การเพิ่มเติมหรือแก้ไขชุดคำสั่งจะกระทำโดยเพิ่มเติมหรือแก้ไขเมท็อด เขตข้อมูล (Field) หรือคลาส ที่จำเป็นต้องมีเบื้องต้นสำหรับรูปแบบการออกแบบที่เลือก แล้วปิดแฟ้มข้อมูลชุดคำสั่งทันที โดยการเพิ่มเติมหรือแก้ไขชุดคำสั่งของแต่ละรูปแบบการออกแบบจะอธิบายอย่างละเอียดในบทที่ 4

บทที่ 4

การพัฒนาซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

การพัฒนาซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ในวิทยานิพนธ์นี้ ได้รับการพัฒนาขึ้นภายใต้ข้อกำหนดในการเขียนจาวาบีน [9] และใช้ชุดพัฒนาภาษาจาวา Java2 SDK รุ่น 1.4.1 ในการพัฒนา

4.1 การพัฒนาซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

การพัฒนาซอฟต์แวร์สำหรับสร้างส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบจะแบ่งเป็น 3 ส่วน คือ

1. ส่วนที่ใช้กำหนดคุณลักษณะ ซึ่งเป็นส่วนติดต่อกับผู้ใช้
2. ส่วนที่ใช้ในการสร้างชุดคำสั่ง
3. ส่วนที่ใช้จัดการแฟ้มข้อมูลของชุดคำสั่ง

4.1.1 ส่วนที่ใช้กำหนดคุณลักษณะ

ส่วนที่ใช้กำหนดคุณลักษณะ ประกอบไปด้วยเมท็อดสองเมท็อดต่อหนึ่งคุณลักษณะ ภายใต้ข้อกำหนดในการเขียนจาวาบีน โดยใช้เมท็อดสาธารณะ (Public method) ที่ขึ้นต้นด้วยคำว่า get และ set ตามด้วยชื่อของคุณลักษณะ ซึ่งเมท็อด get เป็นการส่งค่าของคุณลักษณะนั้นออกไป ส่วนเมท็อด set เป็นการรับค่าจากผู้ใช้เข้ามา รวมถึงการตรวจสอบค่ารับเข้าด้วย โดยปกติแล้วทั้งสองเมท็อดนี้จะถูกเรียกใช้โดยเครื่องมือช่วยการพัฒนาแบบเห็นภาพ [9] เมื่อกำหนด เมท็อดทั้งสองแล้ว เครื่องมือช่วยการพัฒนาแบบเห็นภาพจะแสดงรายชื่อของคุณลักษณะนั้นออกมาทางหน้าต่างแสดงคุณลักษณะ

เมื่อมีการเรียกเมท็อด set ส่วนประกอบซอฟต์แวร์จะทำการตรวจสอบค่ารับเข้าโดยใช้ เมท็อด fireVetoableChange() ของคลาส java.beans.VetoableChangeSupport ตามข้อกำหนดการตรวจสอบค่ารับเข้าของจาวาบีน หากไม่ถูกต้องส่วนประกอบซอฟต์แวร์จะทำการโยนวัตถุ (throw) ของคลาส PropertyVetoException ออกไป ทำให้เมท็อด set หยุดการทำงานกลับไปที่เครื่องช่วยสร้างแบบเห็นภาพ หากการตรวจสอบค่ารับเข้าถูกต้องจะไม่ทำอะไร และการทำงานจะผ่านไปถึงส่วนที่ทำการสั่งสร้างชุดคำสั่ง

```
class DesignPattern1 extends DesignPatternComponent {
//…
public void setFileName(String fileName) throws Exception {
        String newVal = fileName.trim();
        String className = FileManipulation.getNoExtensionFileName(newVal);
        fireVetoableChange("className", null, className);

        if(apply && this.fileName != null) {
                applyEngine(false, null);
                this.className = className;
                this.fileName = newVal;
                pattFile.setupFile(newVal, className);
                applyEngine(true, null);
        }else {
                this.className = className;
                this.fileName = newVal;
                pattFile.setupFile(newVal, className);
        }
}
public String getFileName() {
        return fileName;
}
//…
}
```

รูปที่ 4.1 ตัวอย่างชุดคำสั่งส่วนที่กำหนดคุณลักษณะ

ตัวอย่างชุดคำสั่งคุณลักษณะในรูปที่ 4.1 แสดงให้เห็นส่วนที่กำหนดคุณลักษณะ FileName ของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ซึ่งจะมีเมท็อดที่เกี่ยวข้องสองเมท็อดคือ getFileName() และ setFileName(filename)

4.1.2 ส่วนที่ใช้ในการสร้างชุดคำสั่ง

การสั่งสร้างชุดคำสั่งจะใช้เมท็อด applyEngine() ดังรูปที่ 4.2 ซึ่งเป็นเมท็อดหลักของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ โดยเป็นผู้ตรวจสอบว่าการสั่งนั้นเป็นการสั่งเพิ่มเติมหรือลบชุดคำสั่ง หากสั่งเพิ่มเติมหรือแก้ไข จะไปทำการเรียกเมท็อด adjustStructure() หากสั่งลบ ซึ่งเกิดขึ้นเมื่อกำหนดให้คุณลักษณะ Apply เป็นเท็จ จะทำการเรียกเมท็อด deleteStructure() ซึ่งทั้งสามเมท็อดที่กล่าวมานั้น เป็นเมท็อดที่อยู่ในคลาสนามธรรม DesignPatternComponent ที่เป็นคลาสแม่ (Parent class) ของส่วนประกอบซอฟต์แวร์ในวิทยานิพนธ์นี้ โดยเมท็อด applyEngine() ได้ระบุการทำงานไว้ในคลาสนามธรรม DesignPatternComponent เนื่องจากเป็นกลไกการทำงานที่เหมือนกันของทุกส่วนประกอบซอฟต์แวร์ ส่วนอีกสองเมท็อดเป็นเพียงเมท็อดนามธรรมดังรูปที่ 4.3 ซึ่งระบุการทำงานเฉพาะเจาะจงลงไปในแต่ละส่วนประกอบซอฟต์แวร์โดยการทำงานนั้นได้ระบุในหัวข้อ 4.2

อนึ่ง การสร้างชุดคำสั่งจะใช้เมท็อดของคลาส SourceCodeManipulation ในการสร้างโดยจะสร้างทีละส่วน รายละเอียดของเมท็อดของคลาสนี้ดูได้ในภาคผนวก ข

```
class DesignPatternComponent extends SourceCodeManiplulation {
protected void applyEngine(boolean applyValue, PropertyChangeEvent pce)
throws PropertyVetoException {
        if((getApply() || applyValue) && isPropertiesNotNull() && !getFinalVersion()) {
                try {
                        if(applyValue)
                                adjustStructure();
                        else deleteStructure();
                }catch(Exception e) {
                        throw new PropertyVetoException(e.getMessage(), pce);
                }
        }else throw new PropertyVetoException(
"Please specify releted class name and other needed properties before apply", pce);
}
//…
}
```

รูปที่ 4.2 ชุดคำสั่งของเมท็อด ApplyEngine() ในคลาส DesignPatternComponent

```
protected abstract void adjustStructure() throws Exception;
protected abstract void deleteStructure() throws Exception;
```

รูปที่ 4.3 ชุดคำสั่งของเมท็อดที่ใช้จัดการชุดคำสั่งในคลาส DesignPatternComponent

4.1.3 ส่วนที่ใช้จัดการแฟ้มข้อมูลของชุดคำสั่ง

ส่วนที่ใช้จัดการแฟ้มข้อมูลของชุดคำสั่ง ใช้สำหรับอ่านและเขียนแฟ้มข้อมูลของชุดคำสั่งภาษาจาวาที่ได้กำหนดในคุณลักษณะแฟ้มข้อมูล รวมถึงสร้างแฟ้มข้อมูลพร้อมชุดคำสั่งภาษาจาวาเริ่มต้นตามหัวข้อ 3.3.1 หากแฟ้มข้อมูลนั้นยังไม่มีในระบบ

```
class FileHandler {
public char[] loadFile() throws IOException {…}
public void saveFile(char[] fileData) throws IOException {…}
public boolean createFile(byte type) throws IOException {…}
public boolean setupFile(String fname, byte type, String className)
        throws IOException {…}
//…
}
```

รูปที่ 4.4 ชุดคำสั่งของเมท็อดที่ใช้จัดการแฟ้มข้อมูล

จากรูปที่ 4.4 เมท็อด loadFile ใช้ดึงชุดคำสั่งในแฟ้มข้อมูลขึ้นมา เมท็อด saveFile ใช้บันทึกชุดคำสั่งที่อยู่ในตัวแปร fileData กลับลงแฟ้มข้อมูล เมท็อด createFile ใช้สร้างแฟ้มข้อมูลขึ้นใหม่โดยกำหนดชนิดของแฟ้มข้อมูลลงในตัวแปรเสริมว่าต้องการแฟ้มแบบใด ได้แก่ แฟ้มข้อมูลตัวต่อประสาน แฟ้มข้อมูลคลาส และ แฟ้มข้อมูลคลาสนามธรรม และเมท็อด setupFile ใช้กำหนดข้อมูลของแฟ้มข้อมูลใหม่รวมถึงสั่งสร้างขึ้นมาด้วย เมท็อดเหล่านี้อยู่ในคลาส คลาส FileHandle

4.2 การสร้างชุดคำสั่งของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

การสร้างชุดคำสั่งของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบจะกระทำโดยดูโครงสร้างของรูปแบบการออกแบบแต่ละรูปแบบและกำหนดคุณลักษณะของจาวาบีนที่จำเป็น

ตามโครงสร้าง ต่อไปนี้จะกล่าวถึงรายละเอียดของการสร้างชุดคำสั่งของรูปแบบการออกแบบทั้ง 23 ชนิด

4.2.1 การสร้างชุดคำสั่งของรูปแบบการออกแบบแอ็บสแทรคท์แฟคทอรี

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.1 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสหลักที่ต้องการทำให้เป็นแอ็บสแทรคท์แฟคทอรี
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่ต้องการทำให้เป็นแอ็บสแทรคท์แฟคทอรีตามคลาสหลัก ซึ่งมีกี่แฟ้มข้อมูลก็ได้
3. ชื่อเมท็อดในการผลิต
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเติมเมท็อดในการผลิตในทุกคลาสที่เกี่ยวข้อง พร้อมคำอธิบายให้ผู้ใช้เขียนชุดคำสั่งในภายหลัง
2. เพิ่มส่วนขยาย (extends) คลาสหลักที่กำหนดในคุณลักษณะให้กับคลาสย่อย

ตัวอย่างต่อไปนี้ คลาส AbstractFactory คือคลาสหลัก และคลาส ConcreteFactory1 คือคลาสย่อย ดังในรูปที่ 4.5 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.6

```
public class AbstractFactory {
        public AbstractFactory () {
        }
}
```

```
public class ConcreteFactory1 {
        public ConcreteFactory1 () {
        }
}
```

รูปที่ 4.5 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบแอ็บสแทรคท์แฟคทอรี

```
public class AbstractFactory {
        public AbstractFactory () {
        }
        public Object1 makeObject1() {
                //TODO : Implement your method here.
        }
        public Object2 makeObject2(int I) {
                //TODO : Implement your method here.
        }
}
```

```
public class ConcreteFactory1 extends AbstractFactory {
        public ConcreteFactory1 () {
        }
        public Object1 makeObject1() {
                //TODO : Implement your method here.
        }
        public Object2 makeObject2(int I) {
                //TODO : Implement your method here.
        }
}
```

รูปที่ 4.6 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบแอ็บสแทรคท์แฟคทอรี

4.2.2 การสร้างชุดคำสั่งของรูปแบบการออกแบบบิวเดอร์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.2 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 6 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสหลักที่ต้องการทำให้เป็นบิวเดอร์
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่ต้องการทำให้เป็นบิวเดอร์ซึ่งมีกี่แฟ้มข้อมูลก็ได้
3. ชื่อเมท็อดในการสั่งสร้างชิ้นส่วนซึ่งมักมีหลายเมท็อด

4. ชื่อเมท็อดในการสร้างผลผลิตสุดท้ายที่บิวเดอร์สร้าง
5. คุณลักษณะ Apply
6. คุณลักษณะ FinalVersion

ส่วนคลาส Director ในรูปจะอยู่ในส่วนของผู้เรียกใช้รูปแบบการออกแบบนี้ซึ่งจะไม่รวมไว้ในการสร้างชุดคำสั่ง

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเติมเมท็อดในการสั่งสร้างชิ้นส่วน และเมท็อดในการสร้างผลผลิตสุดท้ายลงในคลาสที่เกี่ยวข้องทุกคลาส พร้อมคำอธิบายให้ผู้ใช้เขียนชุดคำสั่งในภายหลัง
2. เพิ่มส่วนขยายคลาสหลักที่กำหนดในคุณลักษณะให้กับคลาสย่อย

ตัวอย่างต่อไปนี้ คลาส Builder คือคลาสหลักต้องการทำให้เป็นบิวเดอร์ และคลาส ConcreteBuilder คือคลาสย่อยต้องการทำให้เป็นบิวเดอร์ ดังในรูปที่ 4.7 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.8

```
public class Builder {
        public Builder () {
        }
}
```

```
public class ConcreteBuilder {
        public ConcreteBuilder () {
        }
}
```

รูปที่ 4.7 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบบิวเดอร์

```
public class Builder {
        public Builder () {
        }
        public Object getResult() {
                //TODO : Implement method to return product here.
        }
        public void buildPart1() {
                //TODO : Implement your method here.
        }
        public void buildPart2(int I) {
                //TODO : Implement your method here.
        }
}
```

```
public class ConcreteBuilder extends Builder {
        public ConcreteBuilder () {
        }
        public Object getResult() {
                //TODO : Implement method to return product here.
        }
        public void buildPart1() {
                //TODO : Implement your method here.
        }
        public void buildPart2(int I) {
                //TODO : Implement your method here.
        }
}
```

รูปที่ 4.8 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบบิวเดอร์

4.2.3 การสร้างชุดคำสั่งของรูปแบบการออกแบบแฟคทอรีเมท็อด

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.3 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของตัวต่อประสานที่ต้องการทำให้มีแฟคทอรีเมท็อด
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่ต้องการทำให้มีแฟคทอรีเมท็อด ซึ่งมีกี่แฟ้มข้อมูลก็ได้
3. ชื่อแฟคทอรีเมท็อด
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเติมแฟคทอรีเมท็อดในตัวต่อประสาน
2. เพิ่มเติมแฟคทอรีเมท็อดในคลาสย่อยเป็นเมท็อดสาธารณะ และมีคำอธิบายให้ผู้ใช้เขียนชุดคำสั่งเพิ่มในภายหลังตามต้องการ
3. เพิ่มการอิมพลีเมนต์ตัวต่อประสานซึ่งกำหนดในคุณลักษณะ ให้กับคลาสย่อยที่ต้องการทำให้มีแฟคทอรีเมท็อด

ตัวอย่างต่อไปนี้ ตัวต่อประสาน Creator คือตัวต่อประสานหลัก และคลาส ConcreteCreator คือคลาสย่อย ดังในรูปที่ 4.9 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.10

```
public interface Creator {
}
```

```
public class ConcreteCreator {
        public ConcreteCreator () {
        }
}
```

รูปที่ 4.9 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบแฟคทอรีเมท็อด

```
public interface Creator {
        public Object factoryMethodToAdd(int i);
}
```

```
public class ConcreteCreator implements Test {
        public ConcreteCreator () {
        }
        public Object factoryMethodToAdd(int i) {
                //TODO : Implement factory method.
        }
}
```

รูปที่ 4.10 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบแฟคทอรีเมท็อด

4.2.4 การสร้างชุดคำสั่งของรูปแบบการออกแบบโปรโตไทพ์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.4 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 3 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการทำให้เป็นโปรโตไทพ์
2. คุณลักษณะ Apply
3. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะไม่มีคุณลักษณะของแฟ้มข้อมูลย่อยของโปรโตไทพ์ แต่หากต้องการดังนี้ ให้ใช้ส่วนประกอบซอฟต์แวร์นี้กับทุกๆ คลาสที่เป็นคลาสย่อยแทน

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อดแบบสาธารณะ โดยใช้ชื่อว่า copy() ซึ่งใช้สำหรับคลาสอื่นภายนอกร้องขอให้คัดลอกวัตถุของคลาสนี้ จะไม่ใช้ชื่อ clone เพราะเมท็อด clone() เป็นเมท็อดในคลาส Object ซึ่งเป็นคลาสหลักของคลาสทุกคลาสในภาษาจาวา หากใช้ชื่อนี้จะเป็นการครอบงำ (Override) ซึ่งถ้ากระทำตามนั้นจะไม่สามารถส่งค่ากลับเป็นวัตถุของคลาสนี้ได้ แต่ต้องส่งค่ากลับเป็นวัตถุคลาส Object จึงทำให้ต้องทำการแปลงวัตถุ (casting) อีกครั้ง ทำให้ใช้งานไม่สะดวก
2. เพิ่มการอิมพลีเมนต์ตัวต่อประสาน Cloneable ให้กับคลาสที่ต้องการ

ตัวอย่างต่อไปนี้ คลาส Prototype คือคลาสที่ต้องการทำให้เป็นโปรโตไทพ์ ดังในรูปที่ 4.11 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.12

```
public class Prototype {
        public Prototype () {
        }
}
```

รูปที่ 4.11 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบโปรโตไทพ์

```
public class Prototype implements Cloneable {
        public Prototype () {
        }
        public Test copy() {
                try {
                        return (Test)this.clone();
                }catch(CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return null;
        }
}
```

รูปที่ 4.12 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบโปรโตไทพ์

4.2.5 การสร้างชุดคำสั่งของรูปแบบการออกแบบซิงเกิลตอน

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.5 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 3 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการทำให้เป็นซิงเกิลตอน
2. คุณลักษณะ Apply

3. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเขตข้อมูลสถิตแบบส่วนตัว (private) โดยใช้ชื่อว่า _<ชื่อคลาส> ซึ่งไว้เก็บวัตถุของคลาสนั้นซึ่งมีเพียงวัตถุเดียว

2. เพิ่มเมท็อดสถิตแบบสาธารณะ โดยใช้ชื่อว่า getInstance() ซึ่งใช้สำหรับคลาสอื่นภายนอกร้องขอเมื่อต้องการเข้าถึงวัตถุของคลาสนี้

3. เปลี่ยนแปลงตัวสร้างทุกตัวที่เป็นสาธารณะให้เป็นแบบอารักขา (protected)

ตัวอย่างต่อไปนี้ คลาส Singleton คือคลาสที่ต้องการทำให้เป็นซิงเกิลตอน ดังในรูปที่ 4.13 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนา แก้ไขชุดคำสั่งตัวตัวพิมพ์ตัวเอียงดังในรูปที่ 4.14

```
public class Singleton {
        public Singleton () {
        }
}
```

รูปที่ 4.13 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบซิงเกิลตอน

```
public class Singleton {
        private static Singleton _Singleton = null;
        protected Singleton () {
        }
        public static Singleton getInstance() {
                if(_Singleton == null)
                        _Singleton = new Singleton ();
                return _Singleton;
        }
}
```

รูปที่ 4.14 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบซิงเกิลตอน

4.2.6 การสร้างชุดคำสั่งของรูปแบบการออกแบบอแด็พเตอร์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.6 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อคลาสที่ต้องการทำให้มีอแด็พเตอร์
2. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการทำให้เป็นอแด็พเตอร์
3. ชื่อเมท็อดที่เป็นเมท็อดร้องขอที่จะใช้ในอแด็พเตอร์
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อดที่เป็นเมท็อดร้องขอที่จะใช้ในอแด็พเตอร์ ลงในคลาสที่ต้องการทำให้ เป็นอแด็พเตอร์ พร้อมคำอธิบายให้ผู้ใช้เขียนชุดคำสั่งเพื่อแปลงคำร้องขอไปยังอีกคลาส
2. เพิ่มส่วนขยายคลาสที่ต้องการทำให้มีอแด็พเตอร์ ให้กับคลาสที่ต้องการทำให้เป็นอแด็พเตอร์

ตัวอย่างต่อไปนี้ คลาส Adapter คือคลาสที่ต้องการทำให้เป็นอแด็พเตอร์ ดังในรูปที่ 4.15 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.16

```
public class Adapter {
        public Adapter () {
        }
}
```

รูปที่ 4.15 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบอแด็พเตอร์

```
public class Adapter extends Adaptee {
        public Adapter () {
        }
        public void reqestMethod() {
                //TODO : Call method from the class that wanted to adapt.
        }
}
```

รูปที่ 4.16 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบอแด็พเตอร์

4.2.7 การสร้างชุดคำสั่งของรูปแบบการออกแบบบริดจ์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.7 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่เป็นส่วนนามธรรมของบริดจ์
2. ชื่อคลาสที่เป็นส่วนการทำงานจริงของบริดจ์
3. ชื่อเมท็อดที่เป็นเมท็อดร้องขอที่จะใช้ในบริดจ์
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

เนื่องจากการทำงานหลักของรูปแบบการออกแบบอยู่ที่คลาสหลักทั้งสอง จึงไม่ได้กำหนดคุณลักษณะของคลาสย่อยไว้ ซึ่งผู้ใช้ต้องเขียนเองหากต้องการ

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเขตข้อมูลของคลาสที่เป็นส่วนการทำงานจริงแบบส่วนตัว เพื่อใช้เก็บข้อมูลขณะโปรแกรมทำงานว่าวัตถุของคลาสส่วนนามธรรมกำลังทำงานกับวัตถุของคลาสส่วนการทำงานจริงวัตถุใดอยู่
2. เพิ่มเมท็อดที่เป็นเมท็อดร้องขอลงในคลาส พร้อมคำอธิบายให้ผู้ใช้ร้องขอไปยังเมท็อดของส่วนการทำงานจริง
3. เพิ่มเมท็อดที่เป็นเมท็อด setImplementor(<Implementor>) เพื่อใช้ในการตั้งค่าส่วนการทำงานจริงแบบพลวัตในขณะโปรแกรมทำงาน

ตัวอย่างต่อไปนี้ คลาส Abstraction คือ คลาสที่เป็นส่วนนามธรรมของบริดจ์ ดังในรูปที่ 4.17 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.18 ส่วนคลาสที่เป็นส่วนการทำงานจริงของบริดจ์จะไม่ทำการเพิ่มเติมหรือเปลี่ยนแปลงชุดคำสั่งใดๆ

```
public class Abstraction {
        public Abstraction() {
        }
}
```

รูปที่ 4.17 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบบริดจ์

```
public class Abstraction {
        private ImplementorClass _imp;
        public Abstraction () {
        }
        public void operation() {
                //TODO : call implementor method to operate.
        }
        public void setImplementor(ImplementorClass imp) {
                this._imp = imp;

        }
}
```

รูปที่ 4.18 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบบริดจ์

4.2.8 การสร้างชุดคำสั่งของรูปแบบการออกแบบคอมโพสิต

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.8 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของตัวต่อประสานที่ต้องการทำให้เป็นคอมโพสิต
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่ต้องการทำให้เป็นคอมโพสิตซึ่งมีกี่แฟ้มข้อมูลก็ได้
3. ชื่อเมท็อดที่เป็นตัวปฏิบัติการในคลาสที่เป็นคอมโพสิต
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

ในการสร้างชุดคำสั่งของส่วนประกอบซอฟต์แวร์นี้ จะทำการสร้างโครงสร้างการทำงานไว้ที่คลาสย่อยแทนที่จะเป็นคลาสหลัก เนื่องจากถ้าสร้างไว้ที่คลาสหลัก คลาสย่อยจะต้องขยายมาเท่านั้น ซึ่งในภาษาจาวา การขยายทำได้แค่จากคลาสหลักเพียงคลาสเดียว แต่การใช้งานโดยปกติของรูปแบบการออกแบบนี้คลาสย่อยจะมีคลาสหลักของตัวเองอยู่แล้ว ดังนั้นจึงไม่สะดวกที่จะทำให้มีคลาสหลักอีกคลาส จึงออกแบบให้คลาสหลักเป็นตัวต่อประสานแทนที่จะเป็นคลาสเพื่อความยืดหยุ่นในการใช้ ยังผลให้โครงสร้างการทำงานอยู่ที่คลาสย่อยดังที่กล่าว

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเขตข้อมูลแบบส่วนตัว โดยใช้ชื่อว่า _childList ลงในคลาสย่อยทุกคลาส สำหรับไว้เก็บวัตถุที่เป็นลูกของวัตถุนี้

2. เพิ่มเมท็อด add(<วัตถุที่เป็นลูก>) ลงในตัวต่อประสานและคลาสย่อยทุกคลาส สำหรับเพิ่มวัตถุที่เป็นลูกของวัตถุนี้

3. เพิ่มเมท็อด remove(<วัตถุที่เป็นลูก>) ลงในตัวต่อประสานและคลาสย่อยทุกคลาส สำหรับลบวัตถุที่เป็นลูกของวัตถุนี้

4. เพิ่มเมท็อด getChild()ลงในตัวต่อประสานและคลาสย่อยทุกคลาส เพื่อส่งแถวข้อมูลของวัตถุลูกไปให้วัตถุที่ร้องขอ

5. เพิ่มเมท็อดที่เป็นตัวปฏิบัติการตามที่ระบุในคุณลักษณะ ลงในตัวต่อประสานและคลาสย่อยทุกคลาส พร้อมคำอธิบายชุดคำสั่งให้เขียนชุดคำสั่งเพิ่มเติมตามต้องการ

6. เพิ่มการนำเข้าแพคเกจ java.util ลงในคลาสย่อยทุกคลาส เพื่อใช้คลาส Vector

7. เพิ่มการอิมพลีเมนต์ตัวต่อประสานที่กำหนดในคุณลักษณะให้กับคลาสย่อย

ตัวอย่างต่อไปนี้ ตัวต่อประสาน Component คือ ตัวต่อประสานที่ต้องการทำให้เป็นคอมโพสิต และคลาส Composite คือคลาสย่อยที่ต้องการทำให้เป็นคอมโพสิต ดังในรูปที่ 4.19 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.20

```
public interface Component {
}
```

```
public class Composite {
        public Composite() {
        }
}
```

รูปที่ 4.19 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบคอมโพสิต

```
public interface Component {
        public boolean add(Component c);
        public boolean remove(Component c);
        public Component[] getChild();
        public void method1();
}
```

รูปที่ 4.20 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบคอมโพสิต

```
import java.util.*;


public class Composite implements Component {
        private Vector _childList = new Vector();
        public Composite() {
        }
        public boolean add(Component c) {
                return _childList.add(c);
        }
        public boolean remove(Component c) {
                return _childList.remove(c);
        }
        public Component [] getChild() {
                return (Component [])_childList.toArray();
        }
        public void method1() {
                //TODO : Implements your operation and delete default implementation
if this is a leaf class;
                //      otherwise, please just check default implementation.
                Iterator i = _childList.iterator();
                while(i.hasNext())
                        ((Component)i.next()).method1();
        }
}
```

รูปที่ 4.20 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบคอมโพสิต (ต่อ)

4.2.9 การสร้างชุดคำสั่งของรูปแบบการออกแบบเดคอเรเตอร์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.9 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 6 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสหลักที่ต้องการทำให้เป็นเดคอเรเตอร์
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่ต้องการทำให้เป็นเดคอเรเตอร์ซึ่งมีกี่แฟ้มข้อมูลก็ได้

3. ชื่อคลาสที่ต้องการใช้เดคอเรเตอร์
4. ชื่อเมท็อดที่เป็นตัวปฏิบัติการในคลาสที่เป็นเดคอเรเตอร์
5. คุณลักษณะ Apply
6. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเขตข้อมูลแบบส่วนตัว โดยใช้ชื่อว่า _<คลาสหลักที่ต้องการใช้เดคอเรเตอร์> ลงในคลาสต้องการทำให้เป็นเดคอเรเตอร์ สำหรับไว้เก็บวัตถุที่ต้องการใช้เดคอเรเตอร์
2. เพิ่มส่วนขยายคลาสหลักที่ต้องการใช้เดคอเรเตอร์ให้กับคลาสที่ต้องการทำให้เป็นเดคอเรเตอร์
3. เพิ่มส่วนขยายคลาสที่ต้องการให้เป็นเดคอเรเตอร์ให้กับคลาสย่อย
4. เพิ่มเมท็อดเป็นตัวปฏิบัติการลงในคลาสหลักและคลาสย่อยทุกคลาสต้องการทำให้เป็นเดคอเรเตอร์
5. เพิ่มเมท็อด setComponent(<คลาสที่ต้องการใช้เดคอเรเตอร์>) ลงในคลาสที่หลักต้องการทำให้เป็นเดคอเรเตอร์ สำหรับใช้ในการตั้งค่าของวัตถุที่ต้องการใช้เดคอเรเตอร์ขณะโปแกรมทำงานแบบพลวัต

ตัวอย่างต่อไปนี้ คลาส Decorator คือ คลาสหลักที่ต้องการทำให้เป็นเดคอเรเตอร์ และคลาส ConcreteDecorator คือคลาสย่อยที่ต้องการทำให้เป็นเดคอเรเตอร์ ดังในรูปที่ 4.21 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.22

```
public class Decorator {
        public Decorator() {
        }
}
```

```
public class ConcreteDecorator {
        public ConcreteDecorator() {
        }
}
```

รูปที่ 4.21 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบเดคอเรเตอร์

```
public class Decorator extends ClassWantToUseDecorator {
        private ClassWantToUseDecorator _deco;
        public Decorator() {
        }
        public void operation() {
                //TODO : Call your operation in _deco such as _deco.operation();
                //TODO : Add your decorate behavior here.
        }
        public void setComponent(ClassWantToUseDecorator deco) {
                this._deco = deco;
        }
}
```

```
public class ConcreteDecorator extends Decorator {
        public ConcreteDecorator() {
        }
        public void operation() {
                //TODO : Call this method in parent class as super.operation();
                //TODO : Check default implementation above and add your
decorate behavior here.
        }
}
```

รูปที่ 4.22 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบเดคอเรเตอร์

4.2.10 การสร้างชุดคำสั่งของรูปแบบการออกแบบฟาซาด

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.10 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 4 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการทำให้เป็นฟาซาด
2. ชื่อเมท็อดที่เป็นศูนย์กลางการติดต่อซึ่งคุมระบบย่อย
3. คุณลักษณะ Apply

4. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อดที่เป็นศูนย์กลางการติดต่อซึ่งคุมระบบย่อยพร้อมคำอธิบายให้ผู้ใช้ลงในคลาส

ตัวอย่างต่อไปนี้ คลาส Facade คือ คลาสต้องการทำให้เป็นฟาซาด ดังในรูปที่ 4.23 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.24

```
public class Facade {
        public Facade() {
        }
        public void subSystem1() {
                //…
        }
}
```

รูปที่ 4.23 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบฟาซาด

```
public class Facade {
        public Facade() {
        }
        public void subSystem1() {
                //…
        }
        public void commandCenter() {
                //TODO : Implement connection of your sub
systems here.
        }
}
```

รูปที่ 4.24 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบฟาซาด

4.2.11 การสร้างชุดคำสั่งของรูปแบบการออกแบบไฟลเวจ

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.11 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 6 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการทำให้เป็นไฟลเวจ
2. ชื่อแฟ้มข้อมูลของคลาสย่อยของไฟลเวจ
3. ชื่อแฟ้มข้อมูลของคลาสที่ใช้ผลิตวัตถุของคลาสไฟลเวจ
4. ชื่อเมท็อดที่เป็นการทำงานของไฟลเวจ
5. คุณลักษณะ Apply
6. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเขตข้อมูลเพื่อเก็บกองรวมไฟลเวจ ลงในคลาสที่ใช้ผลิตวัตถุของคลาสไฟลเวจ ใช้สำหรับเก็บไฟลเวจที่มีการแบ่งสรรกันอยู่

2. เพิ่มเมท็อด getFlyweight(<ค่าที่ใช้ระบุไฟลเวจ>) ลงในคลาสใช้ผลิตวัตถุของคลาสไฟลเวจ เพื่อใช้ในการเรียกใช้ไฟลเวจที่แบ่งสรรกันอยู่หรือหากไม่มีจะทำการสร้างไฟลเวจใหม่

3. เพิ่มเมท็อดที่เป็นการทำงานของไฟลเวจลงในคลาสที่ต้องการทำให้เป็นไฟลเวจและคลาสย่อยไฟลเวจ

4. เพิ่มส่วนขยายคลาสที่ต้องการทำให้เป็นไฟลเวจลงในคลาสย่อยไฟลเวจ

5. เพิ่มแพคเกจ java.util ลงในคลาสที่ใช้ผลิตวัตถุของคลาสไฟลเวจ เพื่อใช้งานคลาส TreeMap

ตัวอย่างต่อไปนี้ คลาส FlyweightFactory คือ คลาสที่ใช้ผลิตวัตถุของคลาสไฟลเวจ คลาส Flyweight คือ คลาสที่ต้องการทำให้เป็นไฟลเวจ และคลาส ConcreteFlyweight คือ คลาสย่อยที่ต้องการทำให้เป็นไฟลเวจ ดังในรูปที่ 4.25 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.26

```
public class FlyweightFactory {
        public FlyweightFactory () {
        }
}
```

รูปที่ 4.25 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบไฟลเวจ

```
public class Flyweight {
        public Flyweight () {
        }
}
```

```
public class ConcreteFlyweight {
        public ConcreteFlyweight () {
        }
}
```

รูปที่ 4.25 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบไฟลเวจ (ต่อ)

```
Import java.util.*;

public class FlyweightFactory {
        private TreeMap _fwPool = new TreeMap();
        public FlyweightFactory () {
        }
        public Flyweight getFlyweight(Object key) {
                if(_fwPool.contains(key))
                        return (Flyweight)_pwPool.get(key);

                Flyweight newFlyweight = null;
                //TODO : Create and initialize your new flyweight here.

                _fwPool.add(key, newFlyweight);
                return newFlyweight;
        }
}
```

รูปที่ 4.26 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบไฟลเวจ

```
public class Flyweight {
        public Flyweight () {
        }
        public flyweightOperation() {
                //TODO : Implement your operation here.
        }
}
```

```
public class ConcreteFlyweight extends Flyweight {
        public ConcreteFlyweight () {
        }
        public flyweightOperation() {
                //TODO : Implement your operation here.
        }
}
```

รูปที่ 4.26 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบไฟลเวจ (ต่อ)

4.2.12 การสร้างชุดคำสั่งของรูปแบบการออกแบบพร็อกซี

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.12 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 6 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของตัวต่อประสานที่รับคำร้อง
2. ชื่อแฟ้มข้อมูลของคลาสที่ทำงานหลัก
3. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการทำให้เป็นพร็อกซีของคลาสที่ทำงานหลัก
4. ชื่อเมท็อดที่เป็นตัวปฏิบัติการ
5. คุณลักษณะ Apply
6. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อดที่เป็นตัวปฏิบัติการลงในตัวต่อประสานและทุกคลาสที่เกี่ยวข้อง

2. เพิ่มการอิมพลีเมนต์ตัวต่อประสานที่รับคำร้องให้กับคลาสที่ทำงานหลักและคลาสที่เป็นพร็อกซี

3. เพิ่มเขตข้อมูลส่วนตัวที่เก็บวัตถุของคลาสที่ทำงานหลักลงในคลาสที่เป็นพร็อกซี

ตัวอย่างต่อไปนี้ ตัวต่อประสาน Subject คือ ตัวต่อประสานที่รับคำร้อง คลาส RealSubject คือ คลาสที่ทำงานหลัก และคลาส Proxy คือ คลาสที่ต้องการทำให้เป็นพร็อกซีของคลาสที่ทำงานหลัก ดังในรูปที่ 4.27 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.28

```
public interface Subject {
}
```

```
public class RealSubject {
        public RealSubject () {
        }
}
```

```
public class Proxy {
        public Proxy () {
        }
}
```

รูปที่ 4.27 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบพร็อกซี


```
public interface Subject {
        public void operation();
}
```

รูปที่ 4.28 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบพร็อกซี

```
public class RealSubject implements Subject {
        public RealSubject () {
        }
        public void operation() {
                //TODO : Implement your operation here.
        }
}
```

```
public class Proxy implements Subject {
        private Subject _subj = null;
        public Proxy () {
        }
        public void operation() {
                if(_subj == null) {
                        //TODO : Create and initial your subject object here.
                }

                //TODO : Implement your proxy operation here.

                //TODO : Call this method on _subj.
        }
}
```

รูปที่ 4.28 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบพร็อกซี (ต่อ)

4.2.13 การสร้างชุดคำสั่งของรูปแบบการออกแบบเชนออฟเรสปอนสิบิลิตี

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.13 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของตัวต่อประสานที่ต้องการทำให้เป็นเชนออฟเรสปอนสิบิลิตี
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่ต้องการทำให้เป็นเชนออฟเรสปอนสิบิลิตี ซึ่งมีกี่แฟ้มข้อมูลก็ได้

3. ชื่อเมท็อดที่เป็นตัวจัดการในคลาสที่เป็นเชนออฟเรสปอนสิบิลิตี

4. คุณลักษณะ Apply

5. คุณลักษณะ FinalVersion

ในการสร้างชุดคำสั่งของส่วนประกอบซอฟต์แวร์นี้ จะทำการสร้างโครงสร้างการทำงานไว้ที่คลาสย่อยแทนที่จะเป็นคลาสหลัก เนื่องจากถ้าสร้างไว้ที่คลาสหลัก คลาสย่อยจะต้องขยายมาเท่านั้น ซึ่งในภาษาจาวา การขยายทำได้แค่จากคลาสหลักเพียงคลาสเดียว แต่การใช้งานโดยปกติของรูปแบบการออกแบบนี้คลาสย่อยจะมีคลาสหลักของตัวเองอยู่แล้ว ดังนั้นจึงไม่สะดวกที่จะทำให้มีคลาสหลักอีกคลาส จึงออกแบบให้คลาสหลักเป็นตัวต่อประสานแทนที่จะเป็นคลาสเพื่อความยืดหยุ่นในการใช้ ยังผลให้โครงสร้างการทำงานอยู่ที่คลาสย่อยดังที่กล่าว

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเขตข้อมูลแบบส่วนตัว โดยใช้ชื่อว่า _successor ลงในคลาสย่อยทุกคลาส สำหรับเก็บวัตถุที่เป็นลูกโซ่ถัดไปของวัตถุนี้

4. เพิ่มเมท็อด addToChain(<วัตถุที่เป็นโซ่ถัดไป>) ลงในตัวต่อประสานและคลาสย่อยทุกคลาส สำหรับเพิ่มวัตถุที่เป็นลูกของวัตถุนี้

5. เพิ่มเมท็อด getChain() ลงในตัวต่อประสานและคลาสย่อยทุกคลาส เพื่อส่งแถวข้อมูลของวัตถุลูกไปให้วัตถุที่ร้องขอ

6. เพิ่มเมท็อดที่เป็นตัวจัดการตามที่ระบุในคุณลักษณะ ลงในตัวต่อประสานและคลาสย่อยทุกคลาส พร้อมคำอธิบายให้เขียนชุดคำสั่งเพิ่มเติมตามต้องการ

7. เพิ่มการอิมพลีเมนต์ตัวต่อประสานที่กำหนดในคุณลักษณะให้กับคลาสย่อย

ตัวอย่างต่อไปนี้ ตัวต่อประสาน Handler คือ ตัวต่อประสานที่ต้องการทำให้เป็นเชนออฟเรสปอนสิบิลิตี คลาส ConcreteHandler คือ คลาสย่อยต้องการทำให้เป็นเชนออฟเรสปอนสิบิลิตี ดังในรูปที่ 4.29 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.30

```
public interface Handler {
}
```

รูปที่ 4.29 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบเชนออฟเรสปอนสิบิลิตี

```
public class ConcreteHandler {
        public ConcreteHandler() {
        }
}
```

รูปที่ 4.29 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบเชนออฟเรสปอนสิบิลิตี (ต่อ)

```
public interface Handler {
        public void handle();
        public void addToChain(Handler c);
        public Handler getChain();
}
```

```
public class ConcreteHandler implements Handler {
        private Handler _successor = null;
        public ConcreteHandler() {
        }
        public void handle() {
        }
        public void addToChain(Handler c) {
                //Attention : Call this method to set successor of the chain.
                _successor = c;
        }
        public Handler getChain() {
                //Attention : Call this method to get successor of the chain.
                return _successor;
        }
}
```

รูปที่ 4.30 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบเชนออฟเรสปอนสิบิลิตี

4.2.14 การสร้างชุดคำสั่งของรูปแบบการออกแบบคอมมานด์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.14 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสบัญชาการที่ต้องการทำให้เป็นศูนย์กลางคำสั่ง
2. ชื่อแฟ้มข้อมูลของคลาสที่รับคำสั่ง ซึ่งมีกี่แฟ้มข้อมูลก็ได้
3. ชื่อคำสั่ง เป็นชื่อที่ใช้ในการส่งข้อความระหว่างคลาสเพื่อระบุถึงคำสั่ง
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

การสร้างชุดคำสั่งของส่วนประกอบนี้ จะไม่สร้างคลาสหลักของคลาสที่รับคำสั่ง แต่จะใช้ตัวต่อประสาน java.awt.event.ActionListener ทำหน้าที่แทน

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเขตข้อมูลแบบส่วนตัว โดยใช้ชื่อว่า _actionList ลงในคลาสที่รับคำสั่งทุกคลาส สำหรับไว้เก็บวัตถุที่จะรับคำสั่ง

2. เพิ่มเมท็อด addActionListener(<คลาสที่จะรับคำสั่ง>) ลงในคลาสบัญชาการ สำหรับเพิ่มวัตถุที่จะรับคำสั่ง

3. เพิ่มเมท็อด removeActionListener(<คลาสที่จะรับคำสั่ง>) ลงในคลาสบัญชาการ สำหรับลบวัตถุที่จะรับคำสั่ง

4. เพิ่มเมท็อด doAction() ลงในคลาสบัญชาการ เพื่อให้ผู้ใช้ได้เรียกใช้เมื่อต้องการสั่งคำสั่งที่ตั้งไว้ไปยังคลาสที่จะรับคำสั่งทุกคลาส

5. เพิ่มการนำเข้าแพคเกจ java.awt.event ลงในคลาสที่เกี่ยวข้องทุกคลาส เพื่อใช้ตัวต่อประสาน ActionListener และคลาส ActionEvent

6. เพิ่มการอิมพลีเมนต์ตัวต่อประสาน ActionListener ให้กับคลาสที่จะรับคำสั่ง

ตัวอย่างต่อไปนี้ คลาส Invoker คือ คลาสบัญชาการ คลาส ConcreteCommand คือ คลาสที่รับคำสั่ง ดังในรูปที่ 4.31 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.32

```
public class Invoker     {
        public Invoker () {
        }
}
```

รูปที่ 4.31 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบคอมมานด์

```
public class ConcreteCommand {
        public ConcreteCommand () {
        }
}
```

รูปที่ 4.31 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบคอมมานด์ (ต่อ)

```
import java.awt.event.*;

public class Invoker {
        private Vector _actionList = new Vector();
        public Invoker () {
        }
        public void addActionListener(ActionListerner acl) {
                if(acl != null)
                        _actionList.add(acl);
        }
        public void removeActionListener(ActionListerner acl) {
                if(acl != null)
                        _actionList.remove(acl);
        }
        protected void doAction() {
                //Attention : Call this method when you want to do action.
                //You may add a new method like this method manually to do other
action you want. (just change command name in new method)
                for(int i = 0; i < _actionList.size(); i++)
                ((ActionListener)_actionList.get(i)).actionPerformed(new
ActionEvent(this, 0, "TestCommand"));
        }
}
```

รูปที่ 4.32 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบคอมมานด์

```
import java.awt.event.*;


public class ConcreteCommand implements ActionListener {
        public ConcreteCommand() {
        }
        public void actionPerformed(ActionEvent ace) {
                String command = (String)ace.getActionCommand();
                //TODO : This method will be invoked when an action occur.
        }
}
```

รูปที่ 4.32 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบคอมมานด์ (ต่อ)

4.2.15 การสร้างชุดคำสั่งของรูปแบบการออกแบบอินเตอร์พรีเตอร์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.15 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของตัวต่อประสานที่ต้องการทำให้เป็นอินเตอร์พรีเตอร์
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่ต้องการทำให้เป็นอินเตอร์พรีเตอร์
3. ชื่อคลาสอรรถาธิบาย (Context)
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

ในการสร้างชุดคำสั่งของส่วนประกอบซอฟต์แวร์นี้ จะทำการสร้างโครงสร้างการทำงานไว้ที่คลาสย่อยแทนที่จะเป็นคลาสหลัก เนื่องจากถ้าสร้างไว้ที่คลาสหลัก คลาสย่อยจะต้องขยายมาเท่านั้น ซึ่งในภาษาจาวา การขยายทำได้แค่จากคลาสหลักเพียงคลาสเดียว แต่การใช้งานโดยปกติของรูปแบบการออกแบบนี้คลาสย่อยจะมีคลาสหลักของตัวเองอยู่แล้ว ดังนั้นจึงไม่สะดวกที่จะทำให้มีคลาสหลักอีกคลาส จึงออกแบบให้คลาสหลักเป็นตัวต่อประสานแทนที่จะเป็นคลาสเพื่อความยืดหยุ่นในการใช้ ยังผลให้โครงสร้างการทำงานอยู่ที่คลาสย่อยดังที่กล่าว

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเขตข้อมูลแบบส่วนตัว โดยใช้ชื่อว่า _expression ลงในคลาสย่อยทุกคลาส สำหรับไว้เก็บวัตถุนิพจน์ที่เป็นลูกของวัตถุนี้

2. เพิ่มเมท็อด add(<วัตถุที่เป็นลูก>) ลงในตัวต่อประสานและคลาสย่อยทุกคลาส สำหรับเพิ่มวัตถุนิพจน์ที่เป็นลูกของวัตถุนี้

3. เพิ่มเมท็อด remove(<วัตถุที่เป็นลูก>) ลงในตัวต่อประสานและคลาสย่อยทุกคลาส สำหรับลบวัตถุนิพจน์ที่เป็นลูกของวัตถุนี้

4. เพิ่มเมท็อด getExpression()ลงในตัวต่อประสานและคลาสย่อยทุกคลาส เพื่อส่งแถวข้อมูลของวัตถุลูกไปให้วัตถุที่ร้องขอ

5. เพิ่มเมท็อด interpret(Context) ลงในตัวต่อประสานและคลาสย่อยทุกคลาส พร้อมคำอธิบายชุดคำสั่งให้เขียนชุดคำสั่งเพิ่มเติมตามต้องการ

6. เพิ่มการนำเข้าแพคเกจ java.util ลงในคลาสย่อยทุกคลาส เพื่อใช้คลาส Vector

7. เพิ่มการอิมพลีเมนต์ตัวต่อประสานที่กำหนดในคุณลักษณะให้กับคลาสย่อย

ตัวอย่างต่อไปนี้ ตัวต่อประสาน Expression คือ ตัวต่อประสานที่ต้องการทำให้เป็นอินเตอร์พรีเตอร์ และคลาส ConcreteExpression คือ คลาสย่อยที่ต้องการทำให้เป็นอินเตอร์พรีเตอร์ ดังในรูปที่ 4.33 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.34

```
public interface Expression {
}
```

```
public class ConcreteExpression {
        public ConcreteExpression () {
        }
}
```

รูปที่ 4.33 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบอินเตอร์พรีเตอร์

```
public interface Expression {
        public boolean add(Expression c);
        public boolean remove(Expression c);
        public Expression[] getExpression();
        public void interpret(Context c);
}
```

รูปที่ 4.34 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบอินเตอร์พรีเตอร์

```
import java.util.*;


public class ConcreteExpression implements Expression {
        private Vector _expression = new Vector();
        public ConcreteExpression () {
        }
        public boolean add(Expression c) {
                return _expression.add(c);
        }
        public boolean remove(Expression c) {
                return _expression.remove(c);
        }
        public Expression [] getExpression () {
                return (Expression [])_expression.toArray();
        }
        public void interpret(Context c) {
                //TODO : Implements your interpretation and delete default
implementation if this is a leaf class;
                //      otherwise, please just check default implementation.
                Iterator i = _expression.iterator();
                while(i.hasNext())
                    ((Expression)i.next()).interpret(c);
        }
}
```

รูปที่ 4.34 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบอินเตอร์พรีเตอร์ (ต่อ)

4.2.16 การสร้างชุดคำสั่งของรูปแบบการออกแบบอิเทอเรเตอร์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.16 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 4 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่เก็บค่าที่ต้องการทำให้มีอิเทอเรเตอร์
2. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการให้เป็นอิเทอเรเตอร์
3. คุณลักษณะ Apply

4. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อด iterator () ลงในคลาสที่เก็บค่า สำหรับส่งวัตถุที่เป็นอิเทอเรเตอร์ให้คลาสที่เรียกใช้

2. เพิ่มเขตข้อมูลแบบส่วนตัวของคลาสที่เก็บค่า โดยใช้ชื่อว่า collection ลงในคลาสที่ต้องการทำให้เป็นอิเทอเรเตอร์ เพื่อให้คลาสที่ต้องการทำให้เป็นอิเทอเรเตอร์อ้างถึงได้

3. เพิ่มการอิมพลีเมนต์ตัวต่อประสาน Iterator ให้กับคลาสที่ต้องการให้เป็นอิเทอเรเตอร์

4. เพิ่มเมท็อด next () เมท็อด hasNext () และเมท็อด remove () ลงในคลาสที่ต้องการให้เป็นอิเทอเรเตอร์ ตามตัวต่อประสาน Iterator ในข้อ 2 เพื่อใช้ในการทำงานอิเทอเรเตอร์

5. เปลี่ยนตัวสร้างสาธารณะทั้งหมดในคลาสที่ต้องการให้เป็นอิเทอเรเตอร์ให้เป็นแบบอารักขา แล้วเพิ่มตัวสร้างสาธารณะที่รับตัวแปรเสริม (parameter) เป็นคลาสที่เก็บค่าที่ระบุไว้ในคุณลักษณะ

6. เพิ่มการนำเข้าแพคเกจ java.util ลงในคลาสต้องการให้เป็นอิเทอเรเตอร์ เพื่อใช้ตัวต่อประสาน Iterator

ตัวอย่างต่อไปนี้ คลาส Aggregate คือ คลาสที่เก็บค่าที่ต้องการทำให้มีอิเทอเรเตอร์ และคลาส MyIterator คือ คลาสต้องการทำให้เป็นอิเทอเรเตอร์ ดังในรูปที่ 4.35 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนา แก้ไขชุดคำสั่งด้วยตัวพิมพ์ตัวเอียงดังในรูปที่ 4.36

```
public class MyIterator {
        public MyIterator() {
        }
}
```

```
public class Aggregate {
        public Aggregate() {
        }
}
```

รูปที่ 4.35 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบอิเทอเรเตอร์

```
import java.util.*;


public class MyIterator implements Iterator {
        private Aggregate collection;
        protected MyIterator() {
        }
        public MyIterator(Aggregate collection) {
                this.collection = collection;
        }
        public Object next() {
                //TODO : increase index and return the next element.
        }
        public boolean hasNext() {
                //TODO : test that the collection has next element or not?
        }
        public void remove() {
                throw new UnsupportedOperationException()
                ///TODO Optional : remove UnsupportedOperationException
from here and implements your own method
                //What is this method should do?
                //Removes from the underlying collection the last element
returned by the iterator (optional operation).
                //This method can be called only once per call to next.
                //The behavior of an iterator is unspecified if the underlying
collection is modified
                //while the iteration is in progress in any way other than by
calling this method.
                //If it cannot perform, please throw IllegalStateException.
        }
}
```

รูปที่ 4.36 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบอิเทอเรเตอร์

```
public class Aggregate {
        public Aggregate() {
        }
        public Iterator iterator() {
                return new MyIterator(this);
        }
}
```

รูปที่ 4.36 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบอิเทอเรเตอร์ (ต่อ)

4.2.17 การสร้างชุดคำสั่งของรูปแบบการออกแบบเมดิเอเตอร์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.17 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 4 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการให้เป็นเมดิเอเตอร์
2. ชื่อแฟ้มข้อมูลของตัวต่อประสานของผู้ร่วมงาน
3. ชื่อแฟ้มข้อมูลของคลาสที่เป็นผู้ร่วมงาน
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

การสร้างชุดคำสั่งของส่วนประกอบนี้ จะไม่สร้างคลาสหลักของคลาสเมดิเอเตอร์ แต่จะใช้ตัวต่อประสาน java.awt.event.ActionListener ทำหน้าที่แทน

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มการนำเข้าแพคเกจ java.awt.event ลงในคลาสและตัวต่อประสานที่เกี่ยวข้องทั้งหมด เพื่อใช้ตัวต่อประสาน ActionListener และคลาส ActionEvent
2. เพิ่มการอิมพลีเมนต์ตัวต่อประสาน ActionListener ให้กับคลาสที่ต้องการให้เป็นเมดิเอเตอร์
3. เพิ่มการอิมพลีเมนต์ตัวต่อประสานของผู้ร่วมงานให้กับคลาสผู้ร่วมงาน
4. เพิ่มเมท็อด setMediator(<Mediator>) และ notifyMediator() ลงในตัวต่อประสานของผู้ร่วมงาน
5. เพิ่มเมท็อด actionPeformed(ActionEvent) ตามตัวต่อประสาน ActionListener ลงในคลาสที่ต้องการให้เป็นเมดิเอเตอร์
6. เพิ่มเขตข้อมูลแบบส่วนตัวเพื่อเก็บวัตถุที่เป็นเมดิเอเตอร์ลงในคลาสผู้ร่วมงาน

7. เพิ่มเมท็อด setMediator(<Mediator>) ลงในคลาสผู้ร่วมงาน เพื่อใช้ตั้งค่าวัตถุที่จะเป็นเมดิเอเตอร์

8. เพิ่มเมท็อด notifyMediator() ลงในคลาสผู้ร่วมงาน เพื่อให้คลาสผู้ร่วมงานใช้สำหรับแจ้งเมดิเอเตอร์

ตัวอย่างต่อไปนี้ คลาส ConcreteMediator คือ คลาสที่ต้องการให้เป็นเมดิเอเตอร์ ตัวต่อประสาน Colleague คือตัวต่อประสานของผู้ร่วมงาน และคลาส ConcreteColleague คือ คลาสที่เป็นผู้ร่วมงาน ดังรูปที่ 4.37 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.38

```
public class Mediator  {
        public Mediator() {
        }
}
```

```
public interface Colleague  {
}
```

```
public class ConcreteColleague  {
        public ConcreteColleague() {
        }
}
```

รูปที่ 4.37 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบเมดิเอเตอร์

```
import java.awt.event.*;
public class Mediator implements ActionListener {
        public Mediator () {
        }
        public void actionPerformed(ActionEvent ace) {
                Colleague colleague = (Colleague)getSource();
                //TODO : This method will be invoked when colleagues notify.
        }
}
```

รูปที่ 4.38 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบเมดิเอเตอร์

```
import java.awt.event.*;


public interface Colleague  {
        public void setMediator(ActionListener medi);
        public void notifyMediator();
}
```

```
import java.awt.event.*;


public class ConcreteColleague implement Colleague {
        private ActionListener _mediator = null;
        public ConcreteColleague() {
        }
        public void setMediator(ActionListener medi) {
                this._mediator = medi;
        }
        public void notifyMediator() {
                //Attention : Call this method when you want to notify mediator.
                _mediator.actionPerformed(new ActionEvent(this, 0,
getClass().getName()));
        }
}
```

รูปที่ 4.38 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบเมดิเอเตอร์ (ต่อ)

4.2.18 การสร้างชุดคำสั่งของรูปแบบการออกแบบมีเมนโต

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.18 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 3 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่ต้องการทำให้มีมีเมนโต
2. คุณลักษณะ Apply
3. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อด createMemento () ลงในคลาสที่ต้องการ สำหรับสร้างมีเมนโต

2. เพิ่มเมท็อด setMemento (<มีเมนโต>) ลงในคลาสที่ต้องการ สำหรับการเอาค่าในมีเมนโตกลับคืนสู่วัตถุ

3. เพิ่มคลาสภายใน ชื่อ Memento เพิ่มทำหน้าที่เป็นมีเมนโต โดยมีตัวสร้างและเมท็อดอื่นๆ เป็นแบบส่วนตัวทั้งหมด เพื่อไม่ให้คลาสภายนอกสามารถสร้างขึ้นหรือเข้าถึงได้ มีเพียงคลาสที่ต้องการทำให้มีมีเมนโตเท่านั้นที่เข้าถึงได้

4. เพิ่มการอิมพลีเมนต์ตัวต่อประสาน Cloneable ให้กับคลาสที่ต้องการ

ตัวอย่างต่อไปนี้ คลาส Originator คือ คลาสที่ต้องการทำให้มีมีเมนโต ดังในรูปที่ 4.39 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.40

```
public class Originator  {
        public Originator() {
        }
}
```

รูปที่ 4.39 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบมีเมนโต

```
public class Originator implements Cloneable {
    public Originator() {
    }
    public Object createMemento() {
        try {
            return new Memento((Originator)this.clone());
        }catch(CloneNotSupportedException e) {e.printStackTrace();}
        return null;
    }
    public void setMemento(Object obj) {
        if(obj instanceof Memento) {
            Originator save = ((Memento)obj).state;

            //TODO: Restore fields which you need.
```

รูปที่ 4.40 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบมีเมนโต

```
                    //Example: this.fieldname = save.fieldname;
                }
        }
        public final class Memento {
                private final Originator state;
                private Memento(Originator state) {
                        this.state = state;
                }
        }
}
```

รูปที่ 4.40 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบมีเมนโต (ต่อ)

4.2.19 การสร้างชุดคำสั่งของรูปแบบการออกแบบออบเซิร์ฟเวอร์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.19 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 4 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสที่จะถูกสังเกตการณ์
2. ชื่อแฟ้มข้อมูลของคลาสที่เป็นผู้สังเกตการณ์
3. คุณลักษณะ Apply
4. คุณลักษณะ FinalVersion

การสร้างชุดคำสั่งของส่วนประกอบนี้ จะไม่สร้างคลาสหลักของคลาสทั้งสอง แต่จะใช้คลาส java.util.Observerable ทำหน้าที่แทนคลาส Subject และใช้ตัวต่อประสาน java.util.Observer ทำหน้าที่แทนคลาส Observer ซึ่งคลาสและตัวต่อประสานเหล่านี้ ภาษาจาวาได้เตรียมไว้ให้สำหรับการทำงานด้วยรูปแบบออบเซิร์ฟเวอร์ โดยมีเมท็อดที่ทำงานตามรูปแบบการออกแบบนี้ครบชุด ส่วนรายชื่อเมท็อดเหล่านี้สามารถใช้เครื่องมือช่วยการพัฒนาแบบเห็นภาพดูได้ทันทีขณะเขียนชุดคำสั่ง

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มส่วนขยาย Observable ลงในคลาสที่จะถูกสังเกตการณ์
2. เพิ่มการอิมพลีเมนต์ Observer ลงในคลาสที่เป็นผู้สังเกตการณ์
3. เพิ่มเมท็อด update () ลงในคลาสที่เป็นผู้สังเกตการณ์ตามตัวต่อประสาน Observer

4. เพิ่มการนำเข้าแพคเกจ java.util ลงในคลาสทุกคลาสที่เกี่ยวข้อง เพื่อใช้ตัวต่อประสาน Observer และคลาส Observable ซึ่งเป็น API ซึ่งมีอยู่ในภาษาจาวาในการทำรูปแบบการออกแบบออบเซิร์ฟเวอร์

ตัวอย่างต่อไปนี้ คลาส ConcreteSubject คือ คลาสที่จะถูกสังเกตการณ์ และคลาส ConcreteObserver คือ คลาสที่เป็นผู้สังเกตการณ์ ดังในรูปที่ 4.41 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.42

```
public class ConcreteObserver  {
        public ConcreteObserver () {
        }
}
```

```
public class ConcreteSubject  {
        public ConcreteSubject() {
        }
}
```

รูปที่ 4.41 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบออบเซิร์ฟเวอร์

```
import java.util.*;
public class ConcreteObserver implements Observer {
        public ConcreteObserver() {
        }
        public void update(Observable o, Object arg) {
                //TODO : Implements what to do when observed object is
changed.
                //This method is called whenever the observed object is changed.
                //An application calls an Observable object's notifyObservers
method to have all the object's observers notified of the change.
        }
}
```

รูปที่ 4.42 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบออบเซิร์ฟเวอร์

```
import java.util.*;

public class ConcreteSubject extends Observable {
        public ConcreteSubject () {
        }
}
```

รูปที่ 4.42 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบออบเซิร์ฟเวอร์ (ต่อ)

4.2.20 การสร้างชุดคำสั่งของรูปแบบการออกแบบสเตท

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.20 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของตัวต่อประสานที่จะทำให้มีสเตท
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่จะทำให้มีสเตท
3. ชื่อเมท็อดในการปฏิบัติการเมื่อถึงสเตทที่ต้องการ
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

คลาส Context ในรูปจะเป็นส่วนของคลาสที่เรียกใช้ ซึ่งผู้ใช้ต้องเขียนเอง

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อดในการปฏิบัติการที่ระบุในคุณลักษณะ ลงในตัวต่อประสานและคลาสย่อย
2. เพิ่มการอิมพลีเมนต์ตัวต่อประสานที่ระบุไว้ในคุณลักษณะ

ตัวอย่างต่อไปนี้ ตัวต่อประสาน State คือ ตัวต่อประสานที่จะทำให้มีสเตท และคลาส ConcreteState คือ คลาสย่อยที่จะทำให้มีสเตท ดังในรูปที่ 4.43 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.44

```
public interface State  {
}
```

รูปที่ 4.43 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบสเตท

```
public class ConcreteState  {
        public ConcreteState() {
        }
}
```

รูปที่ 4.43 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบสเตท (ต่อ)

```
public interface State {
        public Object stateOperationMethod();
}
```

```
public class ConcreteState implements State {
        public ConcreteState() {
        }
        public Object stateOperationMethod() {
                //TODO : Implement your state operation.
        }
}
```

รูปที่ 4.44 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบสเตท

4.2.21 การสร้างชุดคำสั่งของรูปแบบการออกแบบสเทรทิจี

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.21 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 5 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของตัวต่อประสานที่จะทำให้มีสเทรทิจี
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่จะทำให้มีสเทรทิจี
3. ชื่อเมท็อดสเทรทิจีที่ต้องการ
4. คุณลักษณะ Apply
5. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อดสเทรทิจีที่ระบุในคุณลักษณะ ลงในตัวต่อประสานและคลาสย่อย
2. เพิ่มการอิมพลีเมนต์ตัวต่อประสานที่ระบุไว้ในคุณลักษณะ

ตัวอย่างต่อไปนี้ ตัวต่อประสาน Strategy คือ ตัวต่อประสานที่จะทำให้มีสเทรทิจี และคลาส ConcreteStrategy คือ คลาสย่อยที่จะทำให้มีสเทรทิจี ดังในรูปที่ 4.45 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.46

```
public interface Strategy {
}
```

```
public class ConcreteStrategy {
        public ConcreteStrategy () {
        }
}
```

รูปที่ 4.45 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบสเทรทิจี

```
public interface Strategy {
        public void strategyMethod();
}
```

```
public class ConcreteStrategy implements Strategy {
        public ConcreteStrategy () {
        }
        public void strategyMethod() {
                //TODO : Implement your strategy method.
        }
}
```

รูปที่ 4.46 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบสเทรทิจี

4.2.22 การสร้างชุดคำสั่งของรูปแบบการออกแบบเทมเพลตเมท็อด

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.22 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 6 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของคลาสหลักที่จะทำให้มีเทมเพลตเมท็อด
2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่จะทำให้มีเทมเพลตเมท็อด
3. ชื่อเมท็อดที่เป็นเทมเพลตเมท็อดที่ต้องการ
4. ชื่อเมท็อดที่เป็นเมท็อดปฏิบัติการที่ต้องการ
5. คุณลักษณะ Apply
6. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อดที่เป็นเทมเพลตเมท็อดที่ระบุในคุณลักษณะ ลงในคลาสหลัก
2. เพิ่มเมท็อดที่เป็นเมท็อดปฏิบัติการแบบปกป้องที่ระบุในคุณลักษณะ ลงในคลาสหลักแบบนามธรรม และแบบสาธารณะลงในคลาสย่อย
3. เพิ่มการขยายคลาสหลักที่ระบุไว้ในคุณลักษณะลงไปในคลาสย่อย
4. เปลี่ยนคลาสหลักให้เป็นคลาสนามธรรม

ตัวอย่างต่อไปนี้ คลาส AbstractClass คือ คลาสหลักที่จะทำให้มีเทมเพลตเมท็อด และคลาส ConcreteClass คือ คลาสย่อยที่จะทำให้มีเทมเพลตเมท็อด ดังในรูปที่ 4.47 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.48

```
public class AbstarctClass  {
        public AbstarctClass() {
        }
}
```

```
public class ConcreteClass {
        public ConcreteClass() {
        }
}
```

รูปที่ 4.47 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบเทมเพลตเมท็อด

```
abstract public class AbstractClass {
        public AbstractClass() {
        }
        public Object templateMethod() {
                //TODO : Implement your template method here by use
operation methods you have defined.
        }
        abstract protected void operationMethod1();
        abstract protected void operationMethod2();
}
```

```
public class ConcreteClass extends AbstarctClass {
        public ConcreteClass() {
        }
        protected void operationMethod1() {
                //TODO : Implements your operation method that let
subclass to customize template method.
        }
        protected void operationMethod2() {
                //TODO : Implements your operation method that let
subclass to customize template method.
        }
}
```

รูปที่ 4.48 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบเทมเพลตเมท็อด

4.2.23 การสร้างชุดคำสั่งของรูปแบบการออกแบบวิสิเตอร์

ตามโครงสร้างของรูปแบบการออกนี้ในรูปที่ 2.23 ส่วนประกอบซอฟต์แวร์นี้มีคุณลักษณะอยู่ 6 คุณลักษณะ คือ

1. ชื่อแฟ้มข้อมูลของตัวต่อประสานของวิสิเตอร์

2. ชื่อแฟ้มข้อมูลของคลาสย่อยที่ต้องการให้เป็นวิสิเตอร์
3. ชื่อแฟ้มข้อมูลของตัวต่อประสานของโครงสร้างคลาสที่ต้องการเพิ่มการปฏิบัติการ
4. ชื่อแฟ้มข้อมูลของคลาสย่อยของโครงสร้างคลาสที่ต้องการเพิ่มการปฏิบัติการ
5. คุณลักษณะ Apply
6. คุณลักษณะ FinalVersion

ส่วนประกอบซอฟต์แวร์นี้จะทำการสร้างชุดคำสั่งโดย

1. เพิ่มเมท็อด visit<ชื่อคลาสย่อยในโครงสร้างคลาส> สำหรับแต่ล่ะคลาสย่อยลงในตัวต่อประสานและคลาสย่อยวิสิเตอร์เพื่อใช้เป็นเมท็อดการปฏิบัติการที่เพิ่มขึ้นตามที่รูปแบบการออกแบบวิสิเตอร์ต้องการ

2. เพิ่มเมท็อด accept(Visitor) ลงในตัวต่อประสานและคลาสย่อยของโครงสร้างคลาสที่ต้องการเพิ่มการปฏิบัติการ เพื่อรับคลาสวิสิเตอร์สำหรับการปฏิบัติการเพิ่มเติม

3. เพิ่มการอิมพลีเมนต์ตัวต่อประสานของวิสิเตอร์ ให้กับคลาสย่อยวิสิเตอร์

4. เพิ่มการอิมพลีเมนต์ตัวต่อประสานของโครงสร้างคลาส ที่ต้องการเพิ่มการปฏิบัติการให้กับคลาสย่อย แต่ในสภาพปกติมักจะมีอยู่แล้ว แต่ถ้าหากยังไม่มีส่วนประกอบซอฟต์แวร์จะทำการเพิ่มให้

ตัวอย่างต่อไปนี้ ตัวต่อประสาน Visitor คือ ตัวต่อประสานของวิสิเตอร์ คลาส ConcreteVisitor คือ คลาสย่อยที่ต้องการให้เป็นวิสิเตอร์ ตัวต่อประสาน Element คือ ตัวต่อประสานของโครงสร้างคลาสที่ต้องการเพิ่มการปฏิบัติการ และคลาส ConcreteElementA คือ คลาสย่อยของโครงสร้างคลาสที่ต้องการเพิ่มการปฏิบัติการ ดังในรูปที่ 4.49 และทำการสร้างชุดคำสั่งเพิ่มเติมด้วยตัวพิมพ์ตัวหนาดังในรูปที่ 4.50

```
public interface Visitor {
}
```

```
public class ConcreteVisitor {
        public ConcreteVisitor() {
        }
}
```

รูปที่ 4.49 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบวิสิเตอร์

```
public interface Element {
        public void operation();
}
```

```
public class ConcreteElementA implements Element {
        public ConcreteElementA () {
        }
        public void operation() {
                //…
        }
}
```

รูปที่ 4.49 ตัวอย่างชุดคำสั่งก่อนเพิ่มเติมรูปแบบการออกแบบวิสิเตอร์ (ต่อ)

```
public interface Visitor {
        public void visitConcreteElementA (ConcreteElementA obj);
}
```

```
public class ConcreteVisitor implements Visitor {
        public ConcreteVisitor() {
        }
        public void visitConcreteElementA (ConcreteElementA obj) {
                //TODO : Implement your visitor operation here.
        }
}
```

รูปที่ 4.50 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบวิสิเตอร์

```
public interface Element {
        public void operation();
        public void acceptVisitor(Visitor v);
}
```

```
public class ConcreteElementA implements Element {
        public ConcreteElementA () {
        }
        public void operation() {
                //…
        }
        public void acceptVisitor(Visitor v) {
                v.visitConcreteElementA(this);
        }
}
```

รูปที่ 4.50 ตัวอย่างชุดคำสั่งหลังเพิ่มเติมรูปแบบการออกแบบวิสิเตอร์ (ต่อ)

บทที่ 5

การทดสอบส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

5.1 สภาพแวดล้อมที่ใช้ในการทดสอบ

การทดสอบต้นแบบส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ผู้ทดสอบทำงานบนเครื่องคอมพิวเตอร์ที่มีหน่วยประมวลผลกลางเป็น AMD Athlon XP 1700+ โดยทำงานที่ 1633 เมกกะเฮิร์ทซ์ หน่วยความจำขนาด 512 เมกกะไบต์ ติดตั้งระบบปฏิบัติการวินโดวส์เอ็กซ์พี (Windows XP) ใช้เครื่องมือช่วยการพัฒนาแบบเห็นภาพเจบิวเดอร์ รุ่นที่ 6 (JBuilder 6) และใช้เครื่องเสมือนของจาวา รุ่นที่ 1.4.1

5.2 การทดสอบการทำงานของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ และผลการทดสอบ

การทดสอบการทำงานของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ทดสอบโดยการติดตั้งร่วมกับเครื่องมือช่วยการพัฒนาแบบเห็นภาพ แล้วทดสอบการสร้างชุดคำสั่งว่าเป็นไปตามที่กำหนดในหัวข้อ 4.2 หรือไม่ โดยการทดสอบจะประกอบด้วย

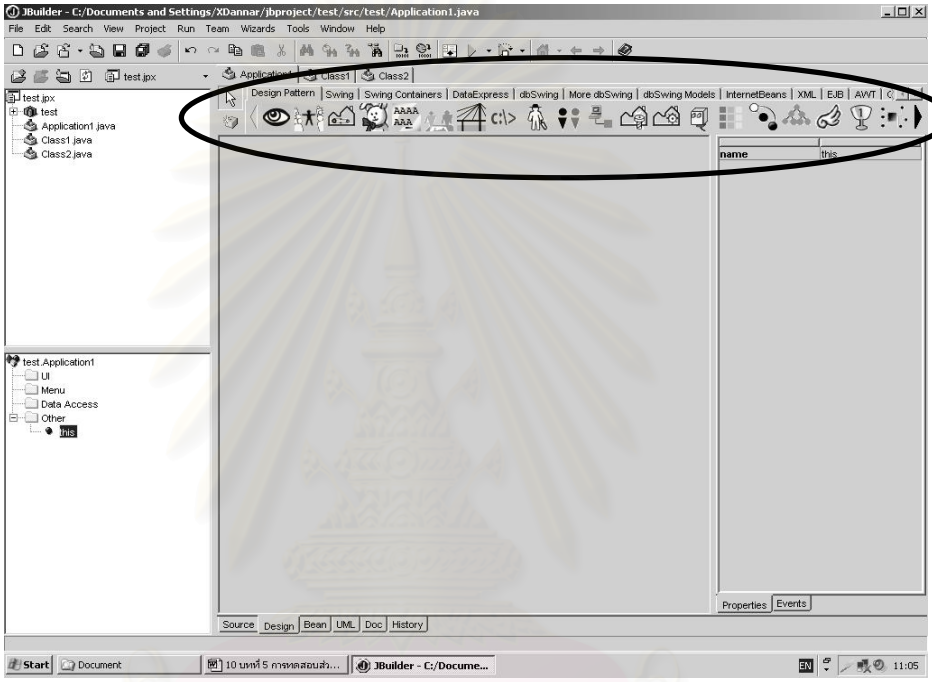
1. การทดสอบการติดตั้งส่วนประกอบซอฟต์แวร์ลงในเครื่องมือช่วยการพัฒนาแบบเห็นภาพ
2. การทดสอบการสร้างชุดคำสั่งเมื่อกำหนดคุณลักษณะต่างๆ ให้กับส่วนประกอบซอฟต์แวร์
3. การทดสอบการสร้างชุดคำสั่งในการพัฒนาโปรแกรมประยุกต์

5.2.1 การทดสอบการติดตั้งส่วนประกอบซอฟต์แวร์ลงในเครื่องมือช่วยการพัฒนาแบบเห็นภาพ

ส่วนประกอบซอฟต์แวร์ทั้งหมด 23 ส่วนประกอบซอฟต์แวร์ก่อนการนำไปใช้งานจริงจะต้องนำมารวมกันเป็นแฟ้มข้อมูลเดียวด้วยแฟ้มรวมข้อมูลจาวา (Java Archive File หรือ JAR) โดยใช้เครื่องมือการรวมแฟ้มข้อมูลในชุดเครื่องมือช่วยการพัฒนาจาวา เมื่อได้แฟ้มรวมข้อมูลจาวาแล้ว ก็จะทำการติดตั้งลงในเครื่องมือช่วยการพัฒนาแบบเห็นภาพต่อไป

การติดตั้งลงในเครื่องมือช่วยการพัฒนาแบบเห็นภาพ ซึ่งในที่นี้ใช้เจบิวเดอร์ รุ่นที่ 6 สามารถทำได้ตามภาคผนวก ก

เมื่อทำการติดตั้งส่วนประกอบซอฟต์แวร์ ลงในเครื่องมือช่วยการพัฒนาแบบเห็นแบบภาพแล้ว สัญรูปของส่วนประกอบซอฟต์แวร์ทั้ง 23 รูปจะปรากฏในกล่องเครื่องมือของเครื่องมือช่วยการพัฒนาแบบเห็นภาพดังจุดที่วงในรูปที่ 5.1 โดยสัญรูปแต่ละรูป แสดงถึงรูปแบบการออกแบบแต่ละรูปแบบ ความหมายของสัญรูปดูได้ในภาคผนวก ก



รูปที่ 5.1 กล่องเครื่องมือในเครื่องมือช่วยการพัฒนาแบบเห็นภาพที่ได้ทำการติดตั้งส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบลงไปแล้ว

5.2.2 การทดสอบการสร้างชุดคำสั่งเมื่อกำหนดคุณลักษณะต่าง ๆ ให้กับส่วนประกอบซอฟต์แวร์

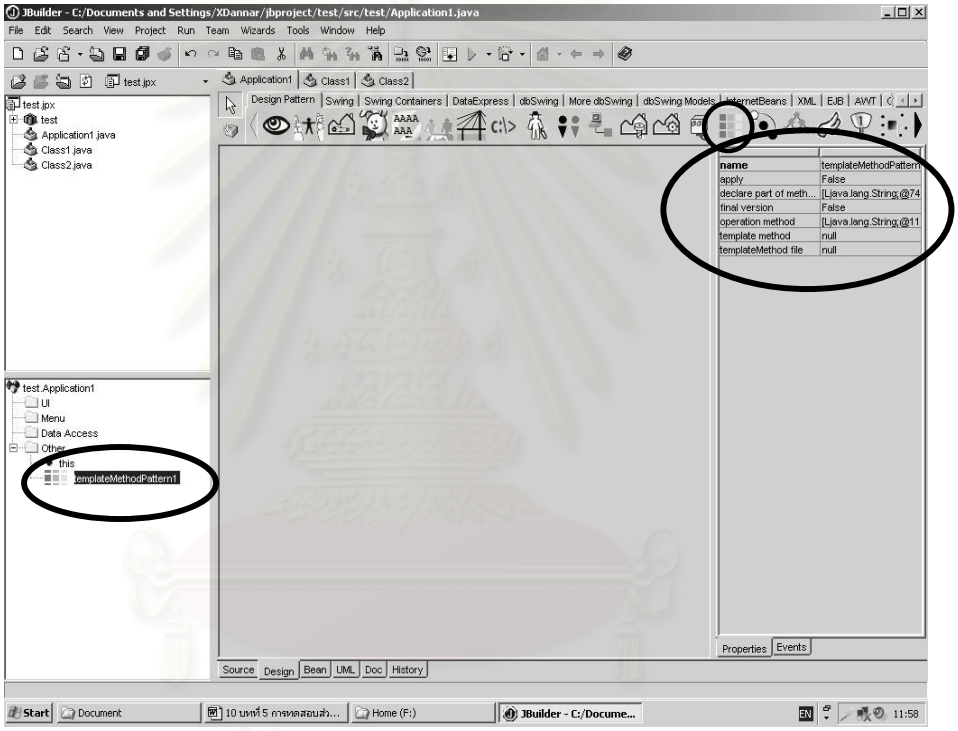
ในส่วนนี้จะทำการทดสอบการสร้าง เปลี่ยนแปลง และลบชุดคำสั่งของส่วนประกอบซอฟต์แวร์โดยใช้ส่วนของโปรแกรมในการทดสอบ การทดสอบนี้จะเป็นการทดสอบทุกเส้นทาง (White box test) โดยทดสอบทั้งหมด 3 กรณี ดังนี้

5.2.2.1 กรณีที่หนึ่ง เป็นการทดสอบการใช้งานรูปแบบการออกแบบเพียงแบบเดียวในโปรแกรม โดยใช้ส่วนประกอบซอฟต์แวร์เทมเพลตเมท็อด การใช้งานส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบสามารถอ่านเพิ่มเติมได้ที่ภาคผนวก ก

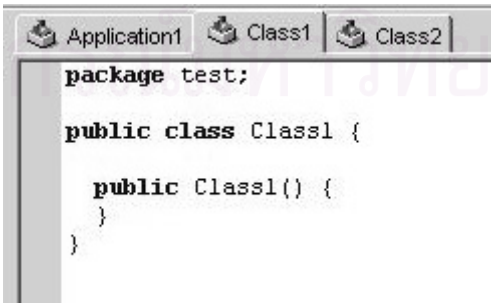
ขั้นตอนการทดสอบ ดังนี้

1. ลากส่วนประกอบซอฟต์แวร์จากกล่องเครื่องมือแล้วปล่อยลงบนฟอร์มของโปรแกรม

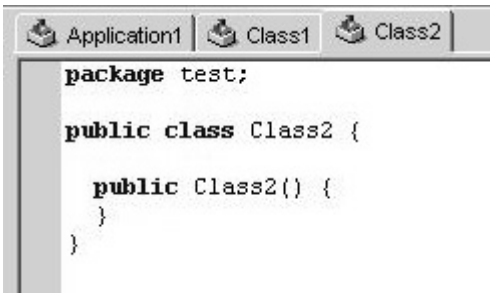
จากรูปที่ 5.2 จะเห็นรายการส่วนประกอบซอฟต์แวร์ที่โปรแกรมใช้ในหน้าต่างด้านซ้ายมือ จะมีรายการของ TemplateMethodPattern เมื่อเลือกส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบในรายการนั้น หน้าต่างคุณลักษณะจะแสดงคุณลักษณะที่อนุญาตให้แก้ไขได้ของส่วนประกอบซอฟต์แวร์ทางด้านขวามือ จากนั้นทำการสร้างคลาสว่างชื่อ Class1 และ Class2 ขึ้นดังรูปที่ 5.3 และ 5.4



รูปที่ 5.2 ผลการทดสอบการทดสอบลากแล้วปล่อยส่วนประกอบซอฟต์แวร์บนฟอร์ม

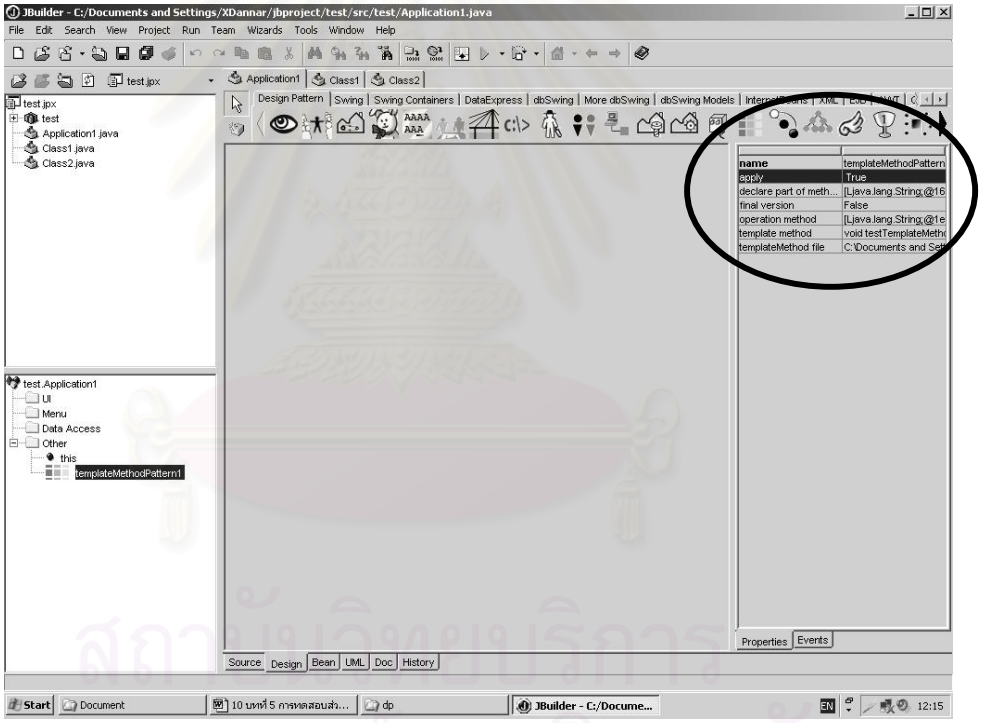


รูปที่ 5.3 ชุดคำสั่งในคลาส Class1 ก่อนการกำหนดคุณลักษณะ Apply ให้เป็นจริง



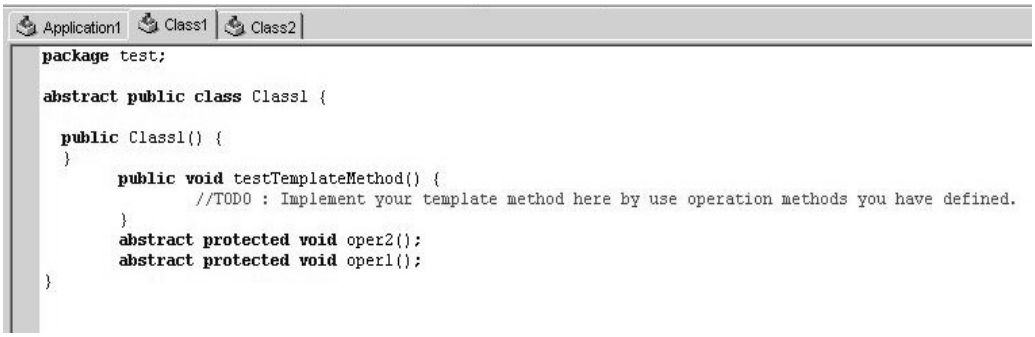
รูปที่ 5.4 ชุดคำสั่งในคลาส Class2 ก่อนการกำหนดคุณลักษณะ Apply ให้เป็นจริง

2. ทำการเปลี่ยนแปลงคุณลักษณะแฟ้มข้อมูลให้เป็นแฟ้มข้อมูลของชุดคำสั่งแฟ้มใดแฟ้มหนึ่งที่อยู่ในโปรแกรมที่กำลังพัฒนา หลังจากนั้นทำการเปลี่ยนแปลงคุณลักษณะ Apply ให้เป็นจริง

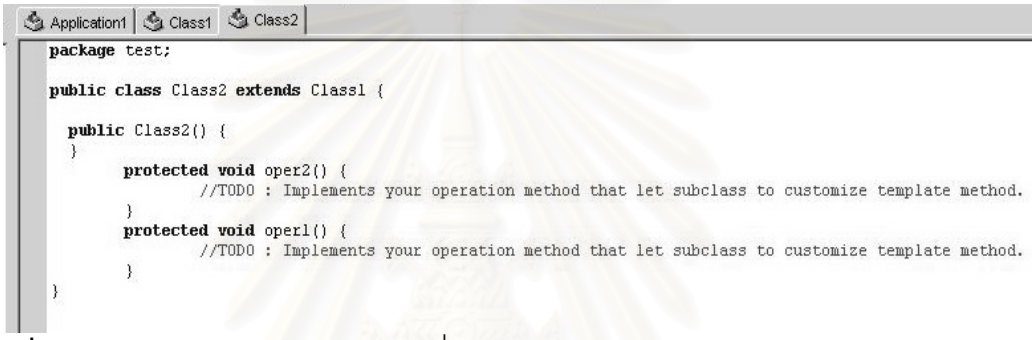


รูปที่ 5.5 การกำหนดคุณลักษณะต่างๆ

จากรูปที่ 5.5 ได้ทำการกำหนดคุณลักษณะต่างๆ ลงในหน้าต่างคุณลักษณะด้านขวา โดยกำหนดให้ Class1 เป็นคลาสหลักที่จะทำให้มีเทมเพลตเมท็อด และ Class2 เป็นคลาสย่อยที่จะทำให้มีเทมเพลตเมท็อด กำหนดชื่อเมท็อดต่างๆ ที่จำเป็น แล้วกำหนดให้คุณลักษณะ Apply เป็นจริง ซึ่งจะทำให้ชุดคำสั่งเปลี่ยนไปตามรูปที่ 5.6 และ 5.7 โดยส่วนประกอบซอฟต์แวร์จะทำการสร้างชุดคำสั่งตามที่ระบุไว้ในหัวข้อ 4.2



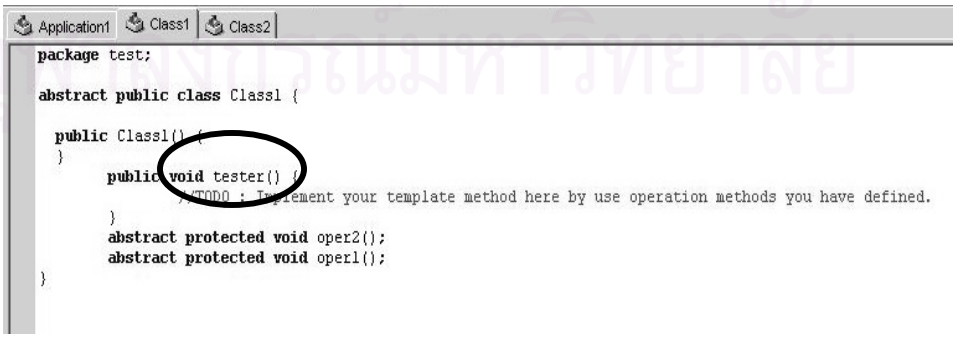
รูปที่ 5.6 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class1 หลังการกำหนดคุณลักษณะ Apply ให้เป็นจริง



รูปที่ 5.7 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class2 หลังการกำหนดคุณลักษณะ Apply ให้เป็นจริง

จากรูปที่ 5.6 และ 5.7 ส่วนประกอบซอฟต์แวร์จะทำการสร้างชุดคำสั่งเพิ่มเติมตามที่ระบุไว้ในหัวข้อ 4.2 อย่างถูกต้อง

3. ทำการเปลี่ยนแปลงคุณลักษณะชื่อเมท็อด ดูการเปลี่ยนแปลงของชุดคำสั่งที่ส่วนประกอบซอฟต์แวร์สร้างขึ้น

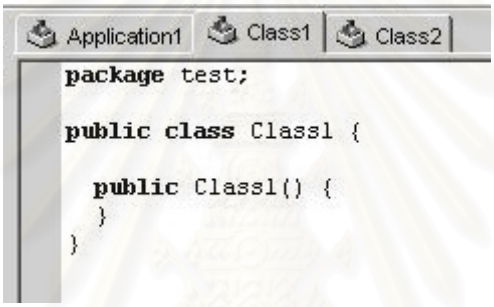


รูปที่ 5.8 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class1 หลังการกำหนดคุณลักษณะใหม่

จากรูปที่ 5.8 จะเห็นว่าชื่อเมท็อดเปลี่ยนไป ซึ่งจากเดิมคือ testTemplateMethod() ในรูปที่ 5.7 เป็นชื่อเมท็อดที่ใส่ลงไปใหม่คือ tester() ตามเงื่อนไขของการออกแบบที่ระบุไว้ในหัวข้อที่ 3.2 ในบทที่ 3

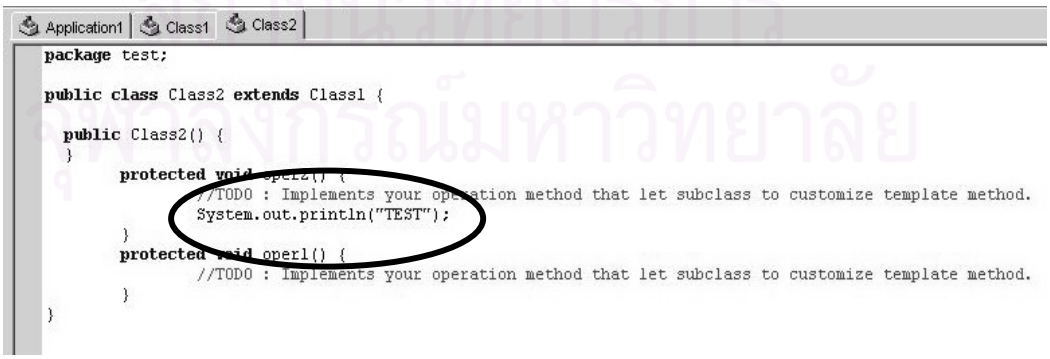
4. ทำการเปลี่ยนแปลงคุณลักษณะ Apply ให้เป็นเท็จ โดยไม่ต้องทำการแก้ไขหรือเพิ่มเติมชุดคำสั่งที่ส่วนประกอบซอฟต์แวร์สร้างขึ้น

จากรูปที่ 5.9 จะเห็นว่าชุดคำสั่งที่ส่วนประกอบซอฟต์แวร์สร้างขึ้นหายไป กลับไปเป็นเหมือนเดิมเหมือนในรูปที่ 5.3 ก่อนการกำหนดคุณลักษณะ Apply ให้เป็นจริง ซึ่งจะเกิดกับ Class2 เช่นกัน



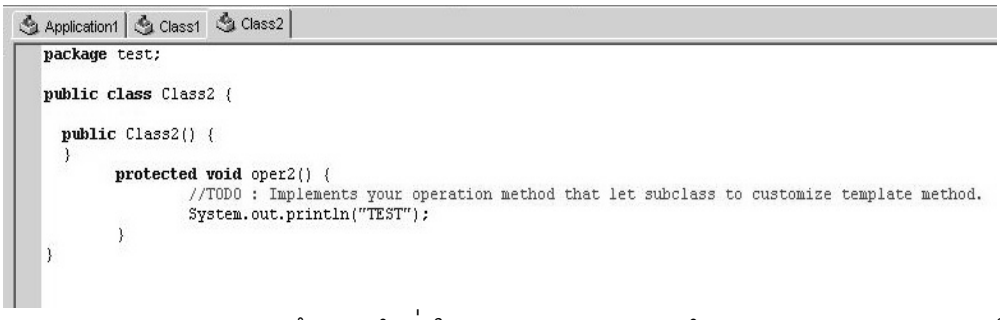
รูปที่ 5.9 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class1 หลังการกำหนดคุณลักษณะ Apply ให้เป็นเท็จ

5. จากข้อ 3 ทำการเพิ่มเติมชุดคำสั่ง System.out.println(“TEST”); ลงไป แล้วทำการเปลี่ยนแปลงคุณลักษณะ Apply ให้เป็นเท็จ



รูปที่ 5.10 ชุดคำสั่งในคลาส Class2 หลังการเขียนชุดคำสั่งเพิ่ม

จากนี้ทำการกำหนดคุณลักษณะ Apply ให้เป็นเท็จ



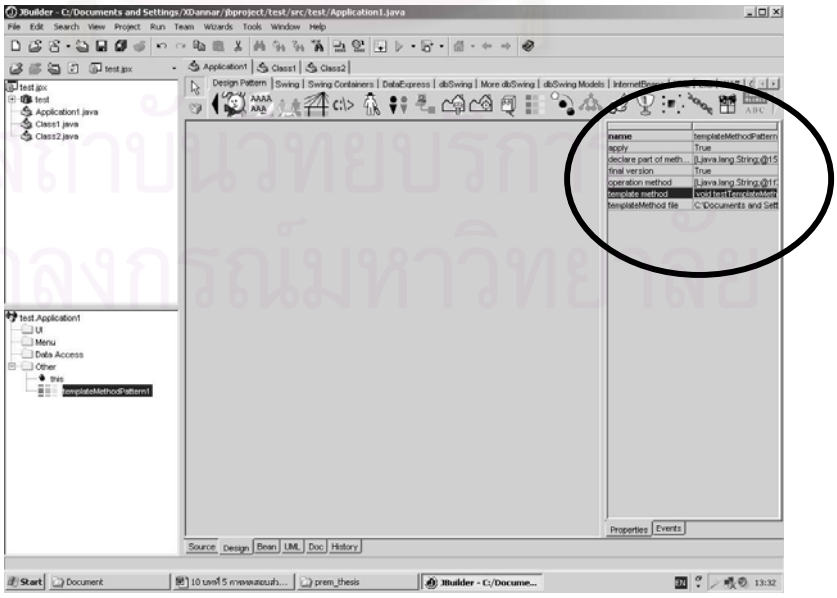
รูปที่ 5.11 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class2 หลังกำหนดคุณลักษณะ Apply ให้เป็นเท็จ

จากรูปที่ 5.11 เห็นได้ว่าชุดคำสั่งทั้งหมดที่ส่วนประกอบซอฟต์แวร์สร้างขึ้นถูกลบออก ยกเว้นส่วนของชุดคำสั่งในเมท็อดที่มีการแก้ไขแล้วจะยังคงอยู่ ทั้งนี้ เพื่อป้องกันการกำหนดคุณลักษณะ Apply ให้เป็นเท็จโดยไม่ตั้งใจ ตามเงื่อนไขของการออกแบบที่ระบุไว้ในหัวข้อที่ 3.2 ในบทที่ 3

6. ทำการเปลี่ยนแปลงคุณลักษณะ FinalVersion ให้เป็นจริงแล้วลองเปลี่ยนแปลงคุณลักษณะอื่นๆ

จากรูปที่ 5.12 จะไม่สามารถกำหนดคุณลักษณะใดๆ ใหม่ได้เลย ตามเงื่อนไขของการออกแบบที่ระบุไว้ในหัวข้อที่ 3.2 ในบทที่ 3

สรุปผลการทดสอบ ส่วนประกอบซอฟต์แวร์สามารถสร้าง เปลี่ยนแปลง ลบ ชุดคำสั่งได้อย่างถูกต้องตามชุดคำสั่งในหัวข้อ 4.2



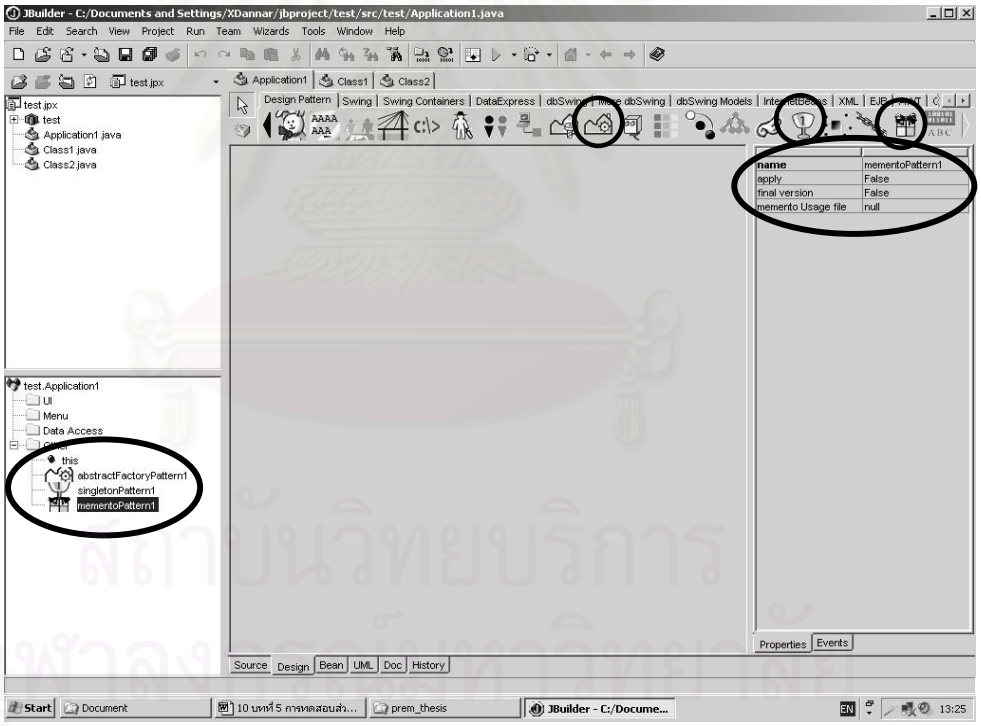
รูปที่ 5.12 ผลการทดสอบหลังกำหนดคุณลักษณะ FinalVersion ให้เป็นจริง

5.2.2.2 กรณีที่สอง เป็นการสร้างคลาสที่ใช้สร้างวัตถุอื่นๆ โดยให้วัตถุของคลาสที่ใช้สร้างนี้เป็นวัตถุตัวกลางเพียงตัวเดียวใช้ทั่วไปทั้งโปรแกรม และให้คลาสนี้สามารถเก็บข้อมูลภายในไว้ได้เมื่อเปิดโปรแกรมใหม่สามารถทำงานต่อได้ การทำงานดังนี้ใช้รูปแบบการออกแบบแอ็บสแทรคท์แฟคทอรี รูปแบบการออกแบบซิงเกิลตอน และรูปแบบการออกแบบมีเมนโต

ขั้นตอนการทดสอบ ดังนี้

1. ลากส่วนประกอบซอฟต์แวร์ทั้งหมดที่ต้องการใช้จากกล่องเครื่องมือแล้วปล่อยลงบนฟอร์มของโปรแกรม

จากรูปที่ 5.13 จะเห็นรายการส่วนประกอบซอฟต์แวร์ที่โปรแกรมใช้ในหน้าต่างด้านซ้ายมือ จะมีรายการของ AbstractFactoryPattern SingletonPattern และ MemmentoPattern เมื่อเลือกส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบในรายการนั้น หน้าต่างคุณลักษณะจะแสดงคุณลักษณะที่อนุญาตให้แก้ไขได้ของส่วนประกอบซอฟต์แวร์ทางด้านขวามือ



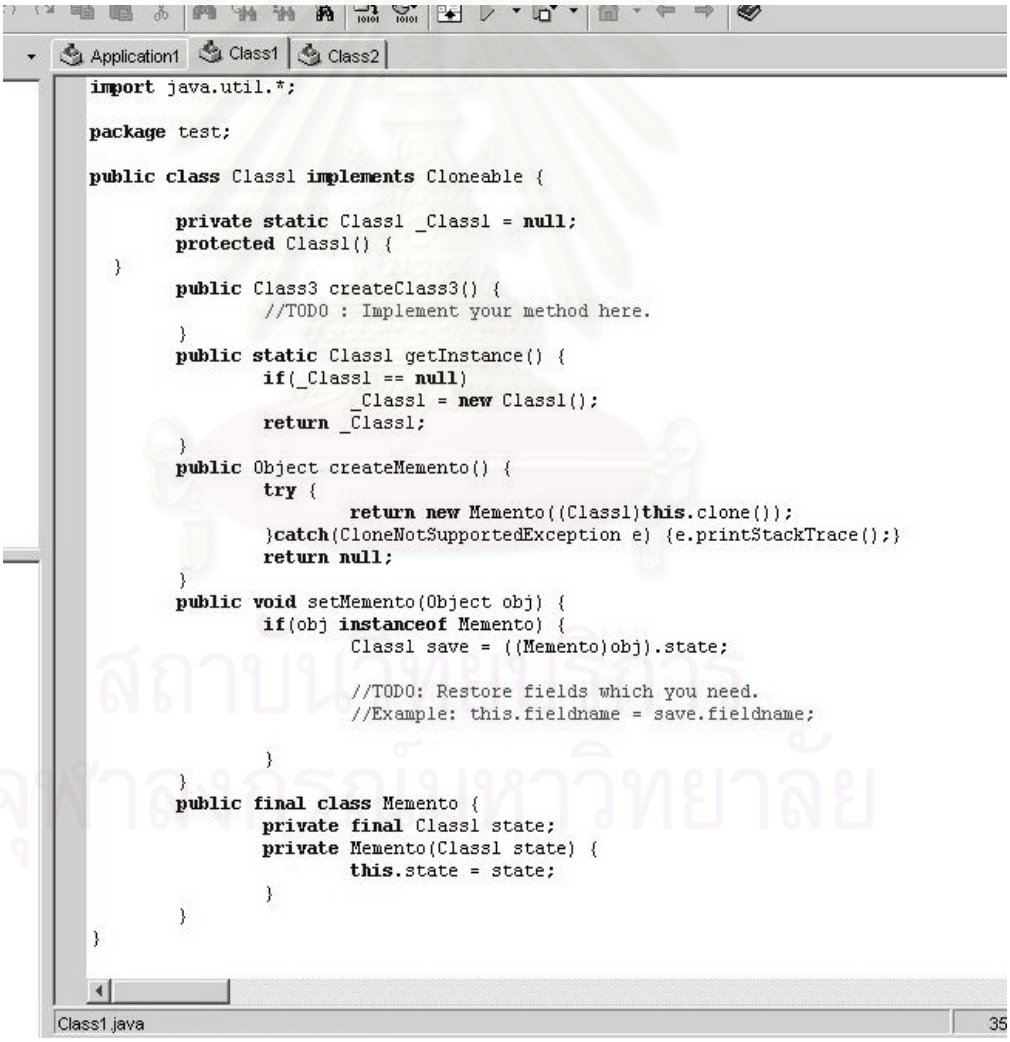
รูปที่ 5.13 ผลการทดสอบการทดสอบลากแล้วปล่อยส่วนประกอบซอฟต์แวร์แอ็บสแทรคท์แฟตทอรี ซิงเกิลตอน และ มีเมนโตลงบนฟอร์ม

จากนี้ ทำการสร้างแฟ้มข้อมูลของคลาส Class1 และ Class2 โดยให้เป็นคลาสว่างเหมือนดังรูปที่ 5.3 และ 5.4

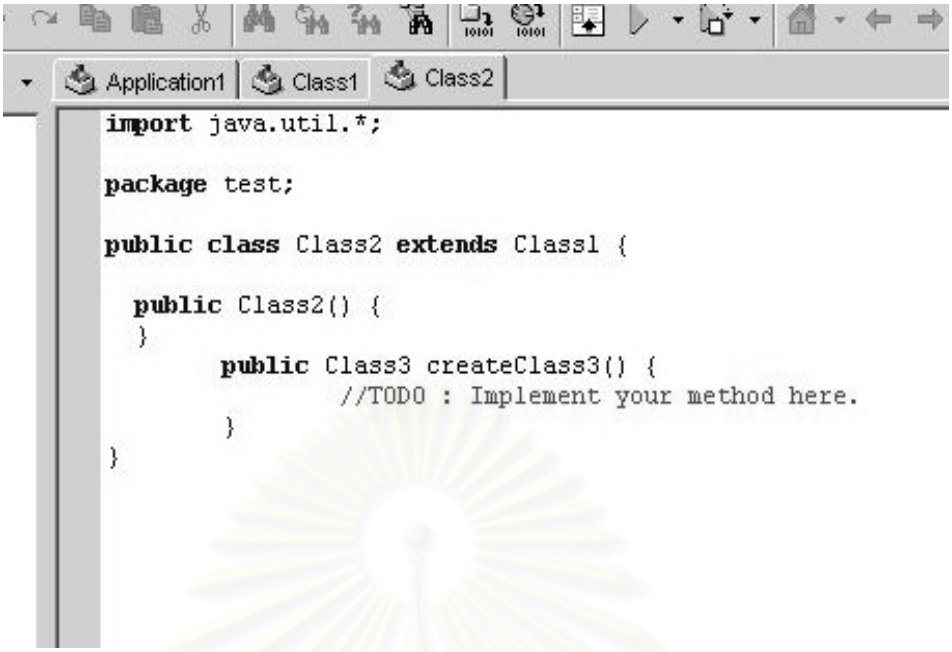
2. ทำการเปลี่ยนแปลงคุณลักษณะต่างๆ และเปลี่ยนแปลงคุณลักษณะ Apply ของทุกส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบที่เลือกมาใช้ให้เป็นจริง

ทำการกำหนดคุณลักษณะต่างๆ ลงในหน้าต่างคุณลักษณะ โดยกำหนดให้ Class1 เป็นคลาสหลักที่จะทำให้เป็นแอ็บสแทรคท์แฟคทอรี และ Class2 เป็นคลาสย่อยที่จะทำให้เป็นแอ็บสแทรคท์แฟคทอรี กำหนดให้คลาส Class1 มีมีเมนโตและเป็นซิงเกิลตอน กำหนดชื่อเมท็อดต่างๆ แล้วกำหนดให้คุณลักษณะ Apply ของทุกรูปแบบการออกแบบให้เป็นจริง ซึ่งจะทำให้ชุดคำสั่งเปลี่ยนไปตามรูปที่ 5.14 และ 5.15 โดยส่วนประกอบซอฟต์แวร์จะทำการสร้างชุดคำสั่งตามที่ระบุไว้ในหัวข้อ 4.2

จากรูปที่ 5.14 และ 5.15 ส่วนประกอบซอฟต์แวร์ทั้งสามทำการสร้างชุดคำสั่งเพิ่มเติมตามที่ระบุไว้ในหัวข้อ 4.2 อย่างถูกต้อง



รูปที่ 5.14 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class1 หลังการกำหนดคุณลักษณะต่างๆ และกำหนดคุณลักษณะ Apply ให้เป็นจริง



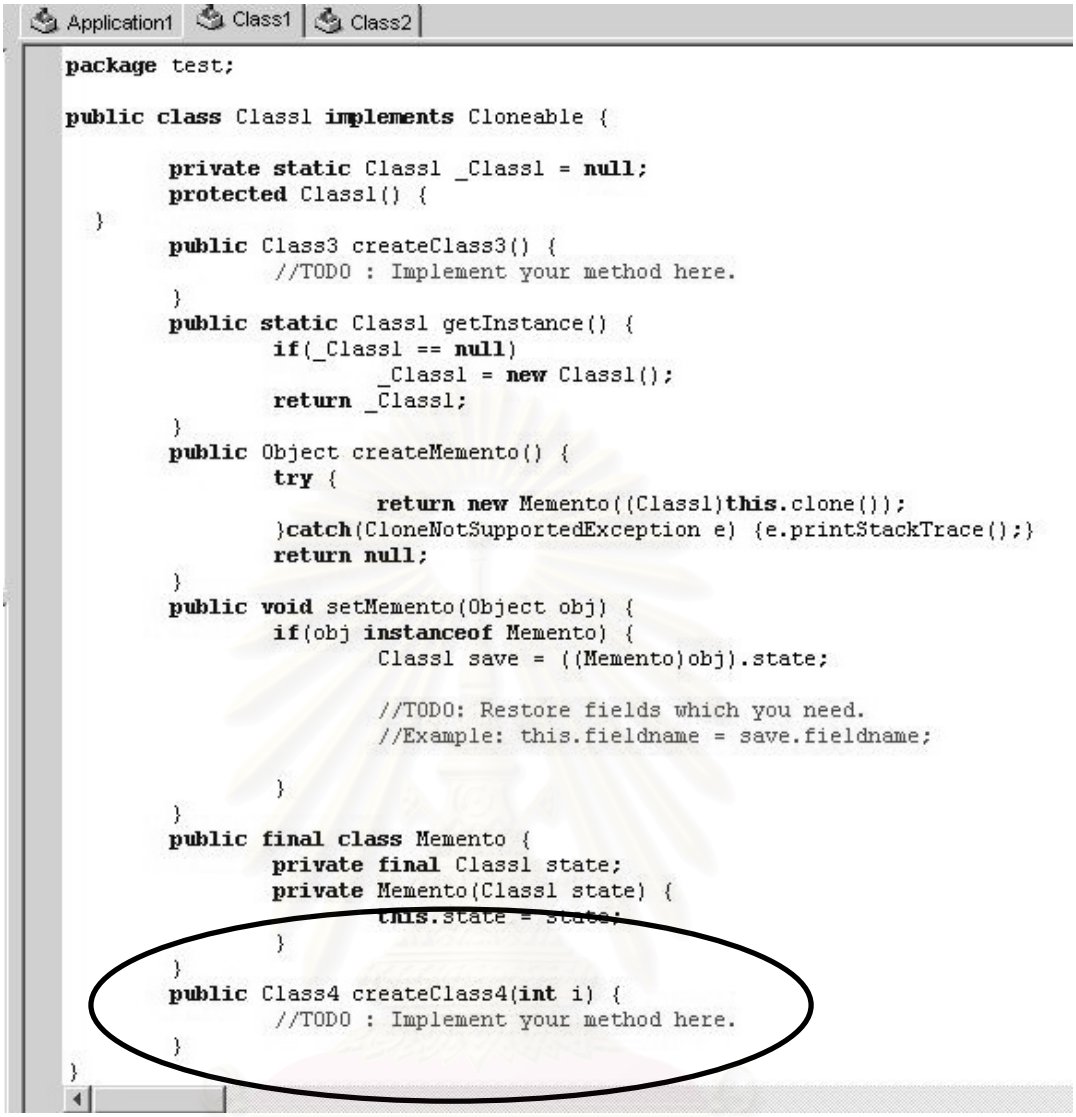
รูปที่ 5.15 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class2 หลังการกำหนดคุณลักษณะต่างๆ และกำหนดคุณลักษณะ Apply ให้เป็นจริง

3. ทำการเปลี่ยนแปลงคุณลักษณะต่างๆ ดูการเปลี่ยนแปลงของชุดคำสั่งที่ส่วนประกอบซอฟต์แวร์สร้างขึ้น

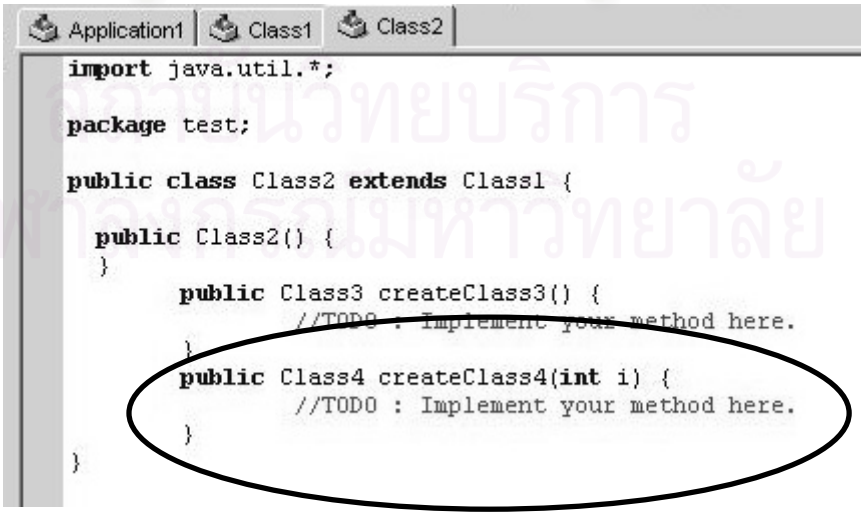
ในที่นี้ ทำการแก้ไขคุณลักษณะชื่อเมท็อดในการผลิตของส่วนประกอบซอฟต์แวร์แอ็บสแทรคท์แฟคทอรี โดยเพิ่มเมท็อดในการผลิตที่ชื่อ Class4 createClass4(int i) ลงไป ซึ่งจะทำให้มีการสร้างชุดคำสั่งขึ้นใหม่ดังในรูปที่ 5.16 และ 5.17 โดยส่วนประกอบซอฟต์แวร์จะทำการสร้างชุดคำสั่งตามที่ระบุไว้ในหัวข้อ 4.2

จากรูปที่ 5.16 และ 5.17 ส่วนประกอบซอฟต์แวร์ทำการสร้างชุดคำสั่งเพิ่มเติมตามที่ระบุไว้ในหัวข้อ 4.2 อย่างถูกต้อง โดยการสร้างโครงสร้างเมท็อดในการผลิต พร้อมคำอธิบายชุดคำสั่งเพื่อให้ผู้ใช้เขียนชุดคำสั่งเพิ่มเองต่อไป

สาเหตุที่ต้องมีการทดสอบในข้อนี้ เพื่อให้แน่ใจว่าส่วนประกอบซอฟต์แวร์นี้ สามารถเพิ่มเติมหรือเปลี่ยนแปลงชุดคำสั่งได้ เมื่อคุณลักษณะต่างๆ มีการเปลี่ยนแปลง



รูปที่ 5.16 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class1 หลังการกำหนดคุณลักษณะใหม่

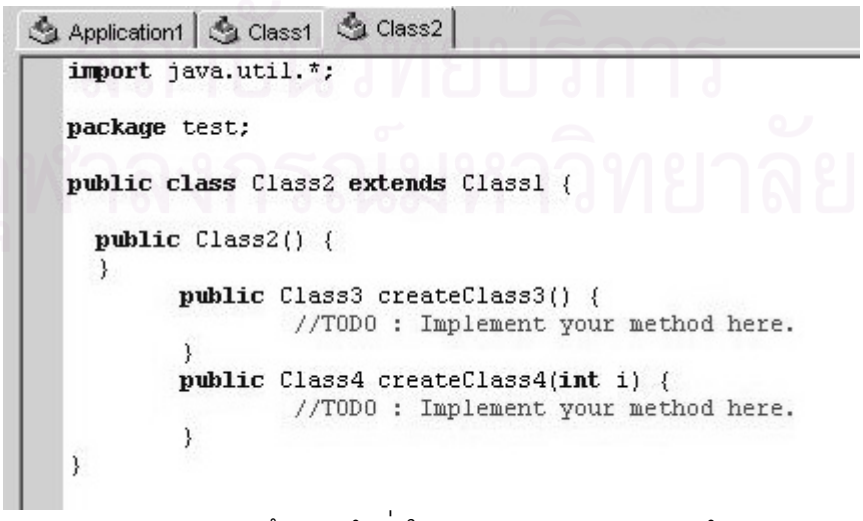


รูปที่ 5.17 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class2 หลังการกำหนดคุณลักษณะใหม่

4. ทำการเปลี่ยนแปลงคุณลักษณะ Apply ของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบที่เลือกมาใช้บางส่วนประกอบให้เป็นเท็จ

ทำการแก้ไขคุณลักษณะ Apply ของรูปแบบการออกแบบมีเมนโตให้เป็นเท็จ ซึ่งจะทำให้ชุดคำสั่งเปลี่ยนไปตามรูปที่ 5.18 และ 5.19 โดยส่วนประกอบซอฟต์แวร์จะลบชุดคำสั่งของรูปแบบการออกแบบมีเมนโตทิ้งตามที่ระบุไว้ในหัวข้อ 4.2

รูปที่ 5.18 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class1 หลังการกำหนดคุณลักษณะใหม่



รูปที่ 5.19 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class2 หลังการกำหนดคุณลักษณะใหม่

จากรูปที่ 5.18 และ 5.19 คลาส Class1 ไม่มีชุดคำสั่งของรูปแบบการออกแบบมีเมนโตแล้ว ส่วนคลาส Class2 ยังคงเดิมเนื่องจากรูปแบบการออกแบบมีเมนโตไม่ได้มีผลอะไรกับคลาสนี้ ส่วนประกอบซอฟต์แวร์ทำการลบชุดคำสั่งตามที่ระบุไว้ในหัวข้อ 4.2 อย่างถูกต้อง

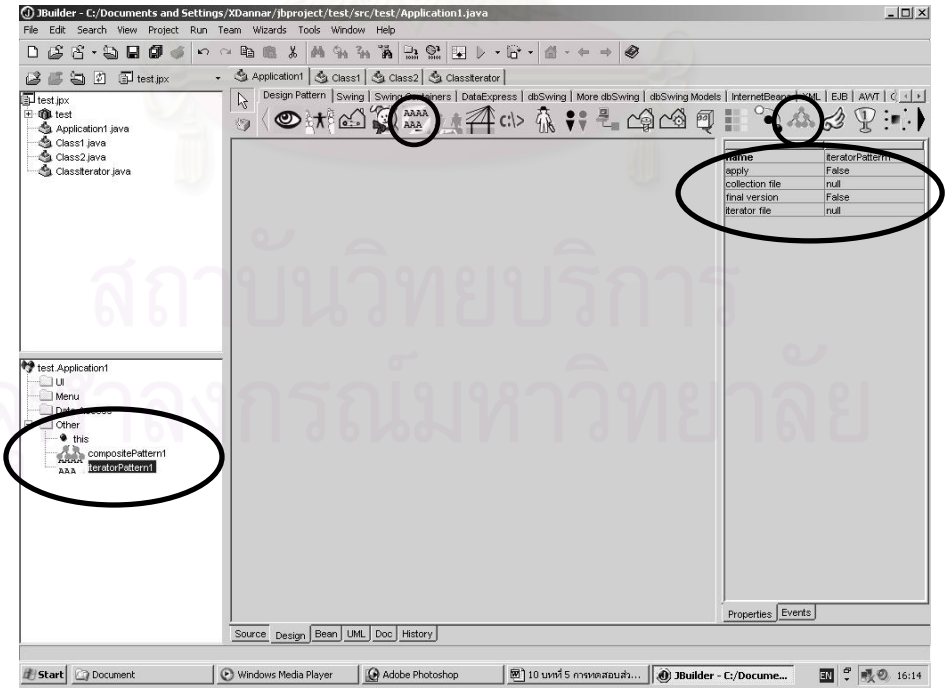
5.2.2.3 กรณีที่สาม เป็นการสร้างคลาสที่เป็นโครงสร้างข้อมูล และมีเมท็อดที่ใช้เข้าถึงโครงสร้างข้อมูลนั้นแบบเรียงลำดับ การทำงานดังนี้จะใช้รูปแบบการออกแบบคอมโพสิต และรูปแบบการอิเทอร์เรเตอร์

ขั้นตอนการทดสอบ ดังนี้

1. ลากส่วนประกอบซอฟต์แวร์ทั้งหมดที่ต้องการใช้จากกล่องเครื่องมือ แล้วปล่อยลงบนฟอร์มของโปรแกรม

จากรูปที่ 5.20 จะเห็นรายการส่วนประกอบซอฟต์แวร์ที่โปรแกรมใช้ในหน้าต่างด้านซ้ายมือ จะมีรายการของ CompositePattern และ IteratorPattern เมื่อเลือกส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบในรายการนั้น หน้าต่างคุณลักษณะจะแสดงคุณลักษณะที่อนุญาตให้แก้ไขได้ของส่วนประกอบซอฟต์แวร์ทางด้านขวามือ

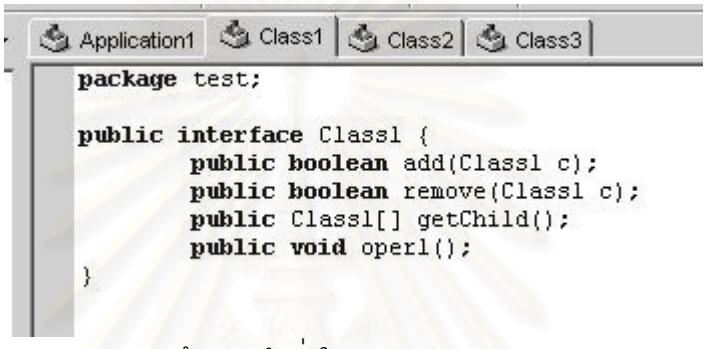
จากนี้ ทำการสร้างแฟ้มข้อมูลของคลาส Class2 และ Class3 โดยให้เป็นคลาสว่างเหมือนดังรูปที่ 5.3 และ 5.4 และสร้างแฟ้มข้อมูลของตัวต่อประสาน Class1 เป็นตัวต่อประสานว่างเช่นกัน



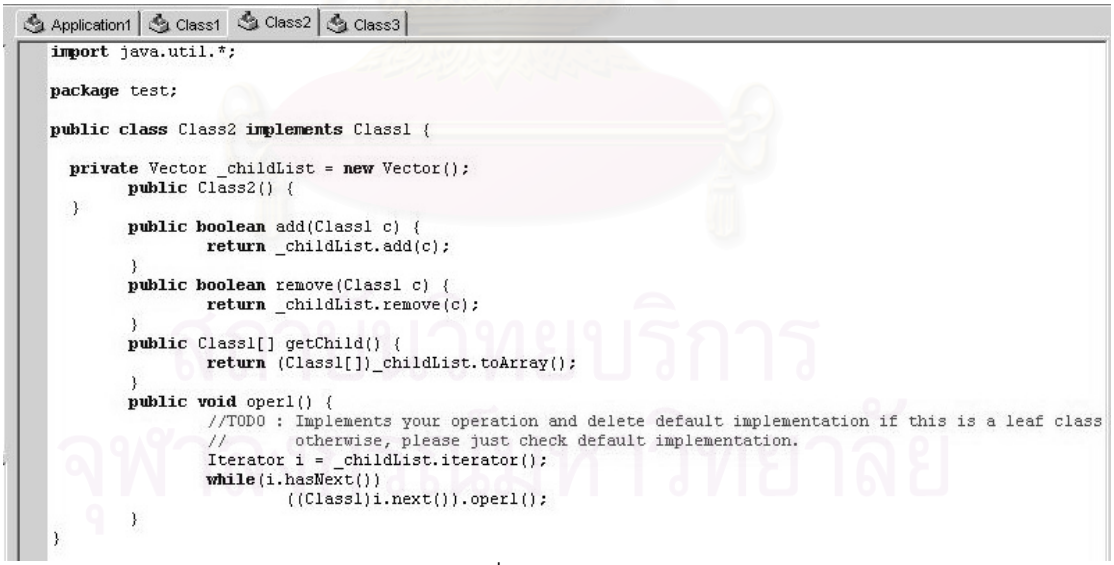
รูปที่ 5.20 ผลการทดสอบการทดสอบลากแล้วปล่อยส่วนประกอบซอฟต์แวร์คอมโพสิตและอิเทอร์เรเตอร์ลงบนฟอร์ม

2. ทำการเปลี่ยนแปลงคุณลักษณะต่างๆ และทำการเปลี่ยนแปลงคุณลักษณะ Apply ของทุกส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบที่เลือกมาใช้ให้เป็นจริง

กำหนดคุณลักษณะต่างๆ ลงในหน้าต่างคุณลักษณะ โดยกำหนดให้ Class1 เป็นตัวต่อประสานที่จะทำให้เป็นคอมโพสิต และ Class2 เป็นคลาสย่อยที่จะทำให้เป็นคอมโพสิต กำหนดให้คลาส Class2 มีอิเทอเรเตอร์ กำหนดให้ Class3 เป็นอิเทอเรเตอร์ กำหนดชื่อเมท็อดต่างๆ แล้วกำหนดให้คุณลักษณะ Apply ของทุกรูปแบบการออกแบบให้เป็นจริง ซึ่งจะทำให้ชุดคำสั่งเปลี่ยนไปตามรูปที่ 5.21 5.22 และ 5.23 โดยส่วนประกอบซอฟต์แวร์จะทำการสร้างชุดคำสั่งตามที่ระบุไว้ในหัวข้อ 4.2

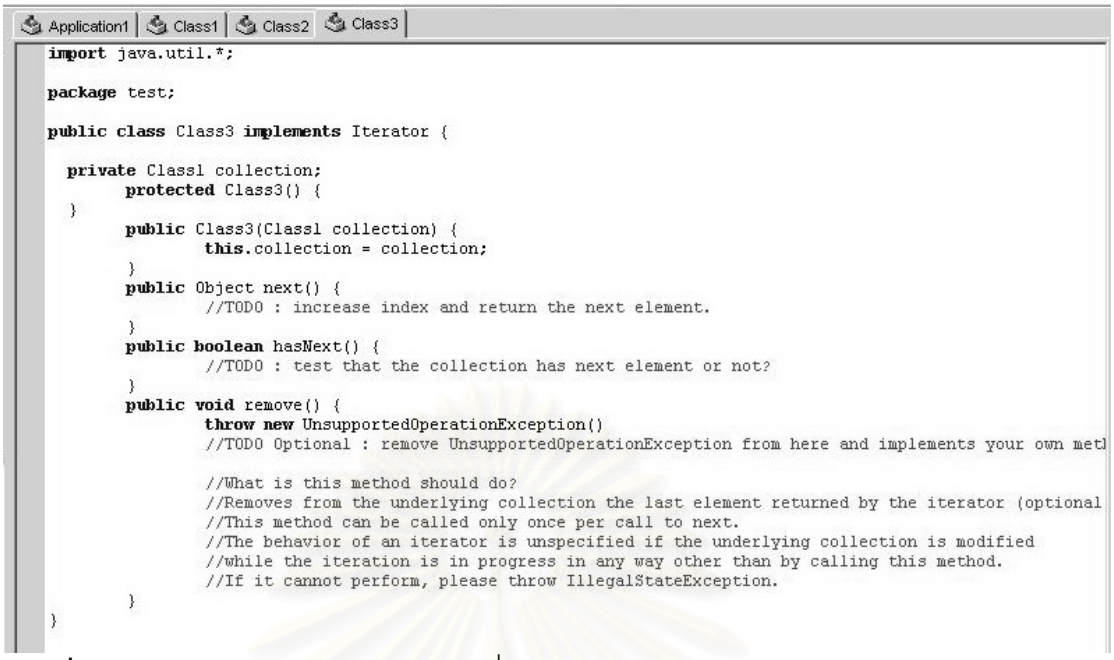


รูปที่ 5.21 ผลการทดสอบการสร้างชุดคำสั่งในตัวต่อประสานทดสอบ Class1 หลังการกำหนดคุณลักษณะต่างๆ และกำหนดคุณลักษณะ Apply ให้เป็นจริง



รูปที่ 5.22 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class2 หลังการกำหนดคุณลักษณะต่างๆ และกำหนดคุณลักษณะ Apply ให้เป็นจริง

จากรูปที่ 5.21 5.22 และ 5.23 ส่วนประกอบซอฟต์แวร์ทำการสร้างชุดคำสั่งตามที่ระบุไว้ในหัวข้อ 4.2 อย่างถูกต้อง



รูปที่ 5.23 ผลการทดสอบการสร้างชุดคำสั่งในคลาส Class3 หลังการกำหนดคุณลักษณะต่างๆ และกำหนดคุณลักษณะ Apply ให้เป็นจริง

5.2.3 การทดสอบการสร้างชุดคำสั่งในการพัฒนาโปรแกรมประยุกต์

ในส่วนนี้จะทำการทดสอบการสร้างชุดคำสั่งของรูปแบบการออกแบบเพื่อช่วยในการพัฒนาโปรแกรมประยุกต์ ทั้งหมด 3 โปรแกรม ดังนี้

5.2.3.1 โปรแกรมที่หนึ่ง เป็นโปรแกรมจัดการข้อมูลพนักงานซึ่งได้พัฒนาเอาไว้แล้ว โดยมีคลาสพนักงานดังตัวอย่างในรูปที่ 5.24 ซึ่งใช้เก็บค่าต่างๆ ของพนักงาน

ปัญหาที่เกิดขึ้นคือ ผู้ดูแลระบบต้องการฟังก์ชันเพิ่มเงินค่าขายให้กับพนักงานฝ่ายขายทุกคนเป็นกรณีพิเศษขึ้นในอัตราตายตัวและอัตราร้อยละ ซึ่งการปฏิบัติการนี้ไม่ใช่ปฏิบัติการที่จะกระทำเป็นปกติของระบบแต่เป็นการใช้งานชั่วคราว และไม่ต้องการเปลี่ยนโครงสร้างเดิมของโปรแกรมให้มีการปฏิบัติการชั่วคราวเหล่านี้ในคลาสพนักงาน ต้องการเพียงเพิ่มการปฏิบัติการชั่วคราวกับระบบที่มีอยู่แล้วเท่านั้น รวมถึงการเพิ่มเติมปฏิบัติการในลักษณะนี้มีอยู่บ่อยครั้งและการเพิ่มเติมแต่ละครั้งไม่เหมือนกันเลย นอกจากนี้การกระทำแต่ละครั้งเป็นการกระทำชั่วคราวทั้งสิ้น

แนวทางการแก้ปัญหาการทำงานลักษณะนี้จะใช้รูปแบบการออกแบบวิสิเตอร์ ซึ่งใช้เพิ่มการปฏิบัติการใหม่ลงบนโครงสร้างของวัตถุเดิมโดยไม่มีการเปลี่ยนแปลงโครงสร้างของวัตถุเดิมที่มีอยู่

Application1 | SaleEmployee

```
package test5332;

import java.io.Serializable;

class SaleEmployee implements Employee, Serializable {
  public SaleEmployee() {
  }
  private int salary;
  private String name;
  private int commisstion;
  public int getSalary() {
    return salary;
  }
  public void setSalary(int salary) {
    this.salary = salary;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public int getCommisstion() {
    return commisstion;
  }
  public void setCommisstion(int commisstion) {
    this.commisstion = commisstion;
  }
}
```

รูปที่ 5.24 ชุดคำสั่งในคลาส SaleEmployee ที่มีอยู่แล้ว

ใช้ส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบวิสิเตอร์ โดยกำหนดคุณลักษณะแฟ้มข้อมูลต่างๆ ให้สร้างตัวต่อประสานขึ้นมาตัวต่อประสานหนึ่งโดยใช้ชื่อว่า MyVisitor และคลาส FixPayVisitor และ PercentPayVisitor เป็นคลาสที่ทำหน้าที่ในการปฏิบัติการนี้ โดยกำหนดคุณลักษณะแฟ้มข้อมูลของตัวต่อประสานของโครงสร้างคลาส ที่ต้องการเพิ่มการปฏิบัติการเป็น Employee และแฟ้มข้อมูลของคลาสย่อยของโครงสร้างคลาสที่ต้องการเพิ่มการปฏิบัติการเป็น SaleEmployee แล้วกำหนดให้คุณลักษณะ Apply และ FinalVersion เป็นจริง ผลทดสอบการช่วยสร้างชุดคำสั่งดังรูปที่ 5.25 5.26 และ 5.27

จากนั้น เขียนชุดคำสั่งเพิ่มเติมเพื่อทำตามที่โปรแกรมต้องการในคลาส FixPayVisitor ให้เพิ่มคนละ 1000 บาท และ PercentPayVisitor ให้เพิ่มคนละร้อยละ 20 ดังรูปที่ 5.28 และ 5.29

SaleEmployee

```
package test5332;

import java.io.Serializable;

class SaleEmployee implements Employee, Serializable {
  public SaleEmployee() {
  }
  private int salary;
  private String name;
  private int commisstion;
  public int getSalary() {
    return salary;
  }
  public void setSalary(int salary) {
    this.salary = salary;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public int getCommisstion() {
    return commisstion;
  }
  public void setCommisstion(int commisstion) {
    this.commisstion = commisstion;
  }
        public void acceptVisitor(MyVisitor v) {
                v.visitSaleEmployee(this);
        }
}
```

รูปที่ 5.25 ผลการทดสอบการสร้างชุดคำสั่งในคลาส SaleEmployee

Application1 | FixPayVisitor | PercentPayVisitor | SaleEmployee

```
package test5332;

public class FixPayVisitor implements MyVisitor {

  public FixPayVisitor() {
  }
        public void visitSaleEmployee(SaleEmployee obj) {
                //TODO : Implement your visitor operation here.
        }
}
```

รูปที่ 5.26 ผลการทดสอบการสร้างชุดคำสั่งในคลาส FixPayVisitor

Application1 | FixPayVisitor | PercentPayVisitor | SaleEmployee

```
package test5332;

public class PercentPayVisitor implements MyVisitor {

  public PercentPayVisitor() {
  }
        public void visitSaleEmployee(SaleEmployee obj) {
                //TODO : Implement your visitor operation here.
        }
}
```

รูปที่ 5.27 ผลการทดสอบการสร้างชุดคำสั่งในคลาส PercentPayVisitor

Application1 | FixPayVisitor | MyVisitor | PercentPayVisitor | SaleEmployee

```
package test5332;

public class FixPayVisitor implements MyVisitor {

  public FixPayVisitor() {
  }
        public void visitSaleEmployee(SaleEmployee obj) {
            obj.setCommisstion(obj.getCommisstion() + 1000);
        }
}
```

รูปที่ 5.28 ชุดคำสั่งในคลาส FixPayVisitor หลังทำการเขียนชุดคำสั่งเพิ่ม

Application1 | FixPayVisitor | MyVisitor | PercentPayVisitor | SaleEmployee

```
package test5332;

public class PercentPayVisitor implements MyVisitor {

  public PercentPayVisitor() {
  }
        public void visitSaleEmployee(SaleEmployee obj) {
            obj.setCommisstion((int)(obj.getCommisstion() * 1.2));
        }
}
```

รูปที่ 5.29 ชุดคำสั่งในคลาส PercentPayVisitor หลังทำการเขียนชุดคำสั่งเพิ่ม

เมื่อเขียนคลาสตามที่ระบบต้องการเสร็จสิ้นแล้ว ขั้นตอนต่อไปคือการเขียนโปรแกรมเพื่อทดสอบการทำงานของคลาสดังกล่าว โดยจะทำการสร้างวัตถุของ SaleEmployee จำนวนหนึ่งขึ้นมา แล้วทำการปฏิบัติการตามที่ระบบต้องการเพิ่มเติมตามเมท็อดในวิสิเตอร์ทั้งสองคลาส แล้วดูผลการคำนวณจากโปรแกรมทดสอบ โปรแกรมทดสอบเขียนได้ดังรูปที่ 5.30

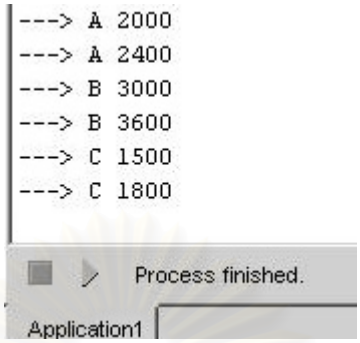
```
SaleEmployee[] s = new SaleEmployee[3];
for(int i = 0; i < s.length; i++)
  s[i] = new SaleEmployee();
s[0].setName("A");
s[0].setCommisstion(1000);
s[1].setName("B");
s[1].setCommisstion(2000);
s[2].setName("C");
s[2].setCommisstion(500);

MyVisitor[] v = new MyVisitor[2];
v[0] = new FixPayVisitor();
v[1] = new PercentPayVisitor();

for(int i = 0; i < s.length; i++)
  for(int j = 0; j < v.length; j++) {
    s[i].acceptVisitor(v[j]);
    System.out.println("---> " + s[i].getName() + " " + s[i].getCommisstion());
  }
```

รูปที่ 5.30 ชุดคำสั่งที่ใช้ในการทดสอบคลาส FixPayVisitor และ PercentPayVisitor

หลังจากนั้นทำการแปลโปรแกรมและดำเนินงาน ได้ผลการทดสอบถูกต้องดังแสดงในรูปที่ 5.31



รูปที่ 5.31 ผลการทดสอบคลาส FixPayVisitor และ PercentPayVisitor

5.2.3.2 โปรแกรมที่สอง เป็นโปรแกรมเกมเขาวงกตที่มีการสร้างหีบสมบัติมากมายในเส้นทางเพื่อให้ผู้เล่นได้เก็บในระหว่างทาง โดยหีบสมบัติแต่ละหีบจะมีข้อมูลภายในบางข้อมูลที่คล้ายๆ กัน เช่น ของที่อยู่ในหีบซึ่งอาจจะเหมือนกันหลายหีบสำหรับของที่หาง่าย หรือหีบเปล่าไม่มีของ จะมีเพียงบางหีบซึ่งมีของที่หายากหรือมีสัตว์ประหลาดอยู่ รวมถึงลักษณะของหีบซึ่งอาจจะเป็นหีบไม้หรือหีบเหล็กเหมือนๆ กันด้วย

ปัญหาที่เกิดขึ้นคือ การสร้างวัตถุของหีบซึ่งมีข้อมูลภายในหลายอย่างนั้นแต่ละครั้งเสียทรัพยากรมาก และต้องสร้างวัตถุชนิดนี้จำนวนมากที่มีลักษณะคล้ายๆ กันด้วย

แนวทางการแก้ปัญหาการทำงานลักษณะนี้ จะเขียนโปรแกรมโดยมีการสร้างคลาสที่เป็นผู้สร้างหีบสมบัติเป็นคลาสกลางให้ตัวโปรแกรมส่วนดำเนินเกมเรียกใช้เพื่อสร้างหีบไปวางจุดต่างๆ ให้เป็นรูปแบบการออกแบบบิวเดอร์เพื่อใช้ในการสร้างส่วนประกอบของหีบสมบัติที่มีหลายส่วนได้สะดวกการ และให้คลาสของหีบสมบัติใช้รูปแบบการออกแบบโปรโตไทพ์ในการคัดลอกวัตถุของหีบที่เหมือนๆ กันเพื่อประหยัดเวลาในการสร้างใหม่

เขียนชุดคำสั่ง โดยสร้างคลาสขึ้นมาคลาสหนึ่งโดยใช้ชื่อว่า MyObjectBuilder เป็นคลาสที่ใช้สร้างหีบและทำให้คลาสนี้เป็นไปตามรูปแบบการออกแบบบิวเดอร์ หลังจากนี้สร้างคลาส MyObject เป็นคลาสของหีบโดยกำหนดเขตข้อมูลและเมท็อดต่างๆ ลงไป โดยทำให้คลาสนี้เป็นคลาสที่จะสร้างโดย MyObjectBuilder โดยเขียนชุดคำสั่งดังรูปที่ 5.32 หลังจากนั้นนำส่วนประกอบซอฟต์แวร์มาใช้โดยทำให้คลาสนี้เป็นรูปแบบการออกแบบโปรโตไทพ์

Application1 | MyObjectBuilder | MyObject

```
package test;

public class MyObject {
  public MyObject(String p1, int p2, Object p3, Object p4) {
    property1 = p1;
    property2 = p2;
    property3 = p3;
    property4 = p4;
  }
  private String property1;
  private int property2;
  private Object property3;
  private Object property4;
  public String getProperty1() {
    return property1;
  }
  public int getProperty2() {
    return property2;
  }
  public Object getProperty3() {
    return property3;
  }
  public Object getProperty4() {
    return property4;
  }
}
```

รูปที่ 5.32 ชุดคำสั่งในคลาส MyObject

จากนั้น ทำการเปลี่ยนคุณลักษณะของแฟ้มข้อมูลชุดคำสั่งไปที่แฟ้มข้อมูลของคลาส MyObjectBuilder ที่สร้างขึ้นมา เพื่อทำงานเป็นวัตถุผู้สร้างหีบสมบัติที่ใช้สำหรับสร้างหีบสมบัติที่ชื่อว่า MyObject กำหนดคุณลักษณะเมท็อดต่างๆ ของบิวเดอร์ แล้วกำหนดคุณลักษณะ Apply และ FinalVersion ของทั้งสองส่วนประกอบซอฟต์แวร์ให้เป็นจริง ผลการทดสอบการช่วยสร้างชุดคำสั่งดังรูปที่ 5.33 และ 5.34

ถัดไปทำการเขียนชุดคำสั่งเพิ่มเติมให้กับเมท็อดของรูปแบบการออกแบบบิวเดอร์เพื่อทำงานตามวัตถุประสงค์ของโปรแกรมลงในคลาส MyObjectBuilder ได้ดังรูปที่ 5.35

เมื่อเขียนคลาสที่ต้องการเสร็จสิ้นแล้ว ต่อไปคือการเขียนโปรแกรมเพื่อทดสอบคลาสดังกล่าว โดยจะให้วัตถุผู้สร้างหีบสมบัติ MyObjectBuilder ทำการสร้างวัตถุหีบสมบัติ MyObject ที่มีโครงสร้างภายในต่างกันขึ้นสองวัตถุโดยกำหนดโครงสร้างทีละส่วน จากนั้นให้วัตถุย่อยคัดลอกตัวเอง แล้วดูโครงสร้างภายในว่าเหมือนกันหรือไม่ โปรแกรมทดสอบเขียนได้ดังรูปที่ 5.36

หลังจากนั้นทำการแปลโปรแกรมและดำเนินงาน ได้ผลการทดสอบถูกต้องดังแสดงในรูปที่ 5.37

Application1 | MyObjectBuilder | MyObject

```
package test;

public class MyObject implements Cloneable {
  public MyObject(String p1, int p2, Object p3, Object p4) {
    property1 = p1;
    property2 = p2;
    property3 = p3;
    property4 = p4;
  }
  private String property1;
  private int property2;
  private Object property3;
  private Object property4;
  public String getProperty1() {
    return property1;
  }
  public int getProperty2() {
    return property2;
  }
  public Object getProperty3() {
    return property3;
  }
  public Object getProperty4() {
    return property4;
  }
        public MyObject copy() {
                try {
                        return (MyObject)this.clone();
                }catch(CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return null;
        }
}
```

รูปที่ 5.33 ผลการทดสอบการสร้างชุดคำสั่งในคลาส MyObject

Application1 | MyObjectBuilder | MyObject

```
package test;

public class MyObjectBuilder {

  public MyObjectBuilder() {
  }
        public MyObject getMyObject() {
                //TODO : Implement method to return product here.
        }
        public void makeProperty4() {
                //TODO : Implement your method here.
        }
        public void makeProperty3(Object obj) {
                //TODO : Implement your method here.
        }
        public void makeProperty2() {
                //TODO : Implement your method here.
        }
        public void makeProperty1(String s) {
                //TODO : Implement your method here.
        }
}
```

รูปที่ 5.34 ผลการทดสอบการสร้างชุดคำสั่งในคลาส MyObjectBuilder

Application1 | MyObjectBuilder | MyObject

```
package test;

import java.util.Calendar;

public class MyObjectBuilder {
  private String p1 = null;
  private static int p2 = 0;
  private Object p3 = null, p4 = null;
  public MyObjectBuilder() {
  }
        public MyObject getMyObject() {
                MyObject m = new MyObject(p1, p2, p3, p4);
                p2 = 0; p1 = null;
                p3 = p4 = null;
                return m;
        }
        public void makeProperty4() {
                p4 = this;
        }
        public void makeProperty3(Object obj) {
                p3 = obj;
        }
        public void makeProperty2() {
                p2 = Calendar.getInstance().get(Calendar.MILLISECOND);
        }
        public void makeProperty1(String s) {
                p1 = s;
        }
}
```

รูปที่ 5.35 ชุดคำสั่งในคลาส MyObjectBuilder หลังทำการเขียนชุดคำสั่งเพิ่ม

```
MyObject[] mo = new MyObject[4];
MyObjectBuilder mob = new MyObjectBuilder();
mob.makeProperty1("Test");
mob.makeProperty2();
mob.makeProperty3(new Vector());
mob.makeProperty4();
mo[0] = mob.getMyObject();

mob.makeProperty1("Hello");
mob.makeProperty2();
mob.makeProperty3(new Exception());
mob.makeProperty4();
mo[1] = mob.getMyObject();

mo[2] = mo[0].copy();
mo[3] = mo[1].copy();

for(int i = 0; i < mo.length; i++) {
  System.out.println("---> " + mo[i].getProperty1() + " " + mo[i].getProperty2() + " " +
                     mo[i].getProperty3().getClass().getName() + " " +
                     mo[i].getProperty4().getClass().getName());
}
```

รูปที่ 5.36 ชุดคำสั่งที่ใช้ในการทดสอบคลาส MyObjectBuilder และ MyObject

```
---> Test 758 java.util.Vector test.MyObjectBuilder
---> Hello 768 java.lang.Exception test.MyObjectBuilder
---> Test 758 java.util.Vector test.MyObjectBuilder
---> Hello 768 java.lang.Exception test.MyObjectBuilder

Process finished.
Application1
```

รูปที่ 5.37 ผลการทดสอบคลาส MyObjectBuilder และ MyObject

5.2.3.3 โปรแกรมที่สาม เป็นโปรแกรมโปรแกรมสัญญาณไฟจราจร จะประกอบด้วยสามส่วนคือ ส่วนแรกจะใช้ในการกดเพื่อให้สัญญาณไฟ ส่วนที่สองเป็นการแจ้งข้อความสำหรับคนข้ามถนน ส่วนที่สามเป็นสัญญาณไฟที่แสดงให้แก่ผู้ขับรถ มีลักษณะการทำงานดังนี้ ส่วนแรกใช้สั่งการเพื่อให้สัญญาณไฟ โดยจะมีสัญญาณสี 3 สี คือ สีแดง สีเหลือง สีเขียว ส่วนที่สองเป็นข้อความแจ้งสำหรับคนข้ามถนนโดยจะสัมพันธ์กับการกดให้สัญญาณไฟ เมื่อสัญญาณไฟแดงสามารถข้ามถนนได้จะแสดงข้อความบอกคนข้าม เมื่อสัญญาณไฟเหลืองหรือสัญญาณไฟสีเขียวจะให้คนหยุดรอข้ามถนนจะแสดงข้อความให้หยุด ส่วนที่สามจะเป็นการแสดงสัญญาณไฟจราจรให้แก่ผู้ขับรถจะสัมพันธ์กับการกดให้สัญญาณไฟในส่วนแรก

ปัญหาที่เกิดขึ้นคือ การออกแบบโปรแกรมบ่อยครั้งที่ต้องการแสดงผลข้อมูลหลายรูปแบบในเวลาเดียวกันและการแสดงผลในรูปแบบต่างๆ ต้องสัมพันธ์กับการเปลี่ยนแปลงของข้อมูลด้วย ซึ่งในการเขียนโปรแกรมอาจจะใช้วิธีเขียนชุดคำสั่งเพื่อเช็คการเปลี่ยนแปลงข้อมูล แล้วทำการสั่งให้ปรับปรุงการแสดงผลทั้งหมด ซึ่งจะต้องระวังในการเขียนโปรแกรมให้ปรับปรุงการแสดงผลให้ครบ และยุ่งยากในการแก้ไขโปรแกรมเมื่อมีการเพิ่มหรือลดจำนวนรูปแบบการแสดงผล

แนวทางการแก้ปัญหาการทำงานลักษณะนี้ เขียนโปรแกรมโดยใช้รูปแบบการออกแบบออบเซิร์ฟเวอร์เพื่อให้ส่วนแสดงผลในส่วนที่สองและสามเฝ้าดูการสั่งการในส่วนแรก รวมถึงเปลี่ยนแปลงตามเมื่อส่วนแรกเปลี่ยน

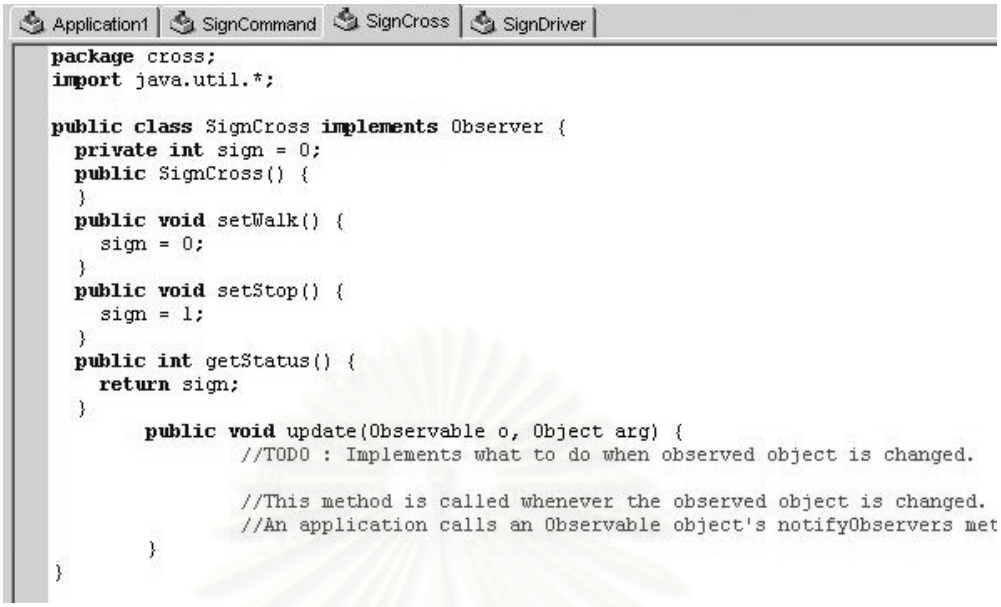
เขียนชุดคำสั่ง โดยสร้างคลาสขึ้นมาและกำหนดเมท็อดที่ต้องการลงไป คลาสที่หนึ่งใช้ชื่อว่า SignCommand ใช้ในการสั่งสัญญาณ คลาสที่สองใช้ชื่อว่า SignCross ใช้แสดงสัญญาณให้คนข้าม และคลาสที่สามชื่อว่า SignDriver ใช้แสดงสัญญาณไฟให้คนขับ โดยกำหนดคุณลักษณะส่วนประกอบซอฟต์แวร์ให้คลาสที่หนึ่งเป็นคลาสที่ถูกสังเกตการณ์ และคลาสที่สองและสามเป็นคลาสผู้เฝ้าสังเกตการณ์ โดยส่วนประกอบซอฟต์แวร์จะช่วยสร้างชุดคำสั่งดังรูปที่ 5.38 5.39 และ 5.40

Application1 | SignCommand | SignCross | SignDriver

```
package cross;
import java.util.*;

public class SignCommand extends Observable {
  private int sign;
  public SignCommand() {
  }
  public void setRed() {
    sign = 0;
  }
  public void setYellow() {
    sign = 1;
  }
  public void setGreen() {
    sign = 2;
  }
  public int getStatus() {
    return sign;
  }
}
```

รูปที่ 5.38 ผลการทดสอบการสร้างชุดคำสั่งในคลาส SignCommand


Application1 | SignCommand | SignCross | SignDriver

```
package cross;
import java.util.*;

public class SignCross implements Observer {
  private int sign = 0;
  public SignCross() {
  }
  public void setWalk() {
    sign = 0;
  }
  public void setStop() {
    sign = 1;
  }
  public int getStatus() {
    return sign;
  }
        public void update(Observable o, Object arg) {
                //TODO : Implements what to do when observed object is changed.

                //This method is called whenever the observed object is changed.
                //An application calls an Observable object's notifyObservers met
        }
}
```

รูปที่ 5.39 ผลการทดสอบการสร้างชุดคำสั่งในคลาส SignCross


Application1 | SignCommand | SignCross | SignDriver

```
package cross;
import java.util.*;

public class SignCross implements Observer {
  private int sign = 0;
  public SignCross() {
  }
  public void setWalk() {
    sign = 0;
  }
  public void setStop() {
    sign = 1;
  }
  public int getStatus() {
    return sign;
  }
        public void update(Observable o, Object arg) {
                //TODO : Implements what to do when observed object is changed.

                //This method is called whenever the observed object is changed.
                //An application calls an Observable object's notifyObservers met
        }
}
```

รูปที่ 5.40 ผลการทดสอบการสร้างชุดคำสั่งในคลาส SignDriver

จากนั้นเขียนโปรแกรมเพิ่มเติมตามข้อกำหนดได้ดังรูปที่ 5.41 5.42 และ 5.43

Application1 | SignCommand | SignCross | SignDriver

```
package cross;
import java.util.*;

public class SignCommand extends Observable {
  private int sign;
  public SignCommand() {
  }
  public void setRed() {
    sign = 0;
    System.out.println("\r\nCommand red sign");
    this.setChanged();
    this.notifyObservers();
  }
  public void setYellow() {
    sign = 1;
    System.out.println("\r\nCommand yellow sign");
    this.setChanged();
    this.notifyObservers();
  }
  public void setGreen() {
    sign = 2;
    System.out.println("\r\nCommand green sign");
    this.setChanged();
    this.notifyObservers();
  }
  public int getStatus() {
    return sign;
  }
}
```

รูปที่ 5.41 ชุดคำสั่งในคลาส SignCommand หลังทำการเขียนชุดคำสั่งเพิ่ม

Application1 | SignCommand | SignCross | SignDriver

```
package cross;
import java.util.*;

public class SignCross implements Observer {
  private int sign = 0;
  public SignCross() {
  }
  public void setWalk() {
    sign = 0;
    System.out.println("Can walk");
  }
  public void setStop() {
    sign = 1;
    System.out.println("Stop walk");
  }
  public int getStatus() {
    return sign;
  }
      public void update(Observable o, Object arg) {
              //TODO : Implements what to do when observ
              int status = ((SignCommand)o).getStatus();
              if(status == 0) //red
                setWalk();
              else if(status > 0) //yellow or green
                setStop();
      }
}
```

รูปที่ 5.42 ชุดคำสั่งในคลาส SignCross หลังทำการเขียนชุดคำสั่งเพิ่ม

Application1 | SignCommand | SignCross | SignDriver

```
package cross;
import java.util.*;

public class SignDriver implements Observer {
  private int sign;
  public SignDriver() {
  }
  public void setRed() {
    sign = 0;
    System.out.println("Now red sign");
  }
  public void setYellow() {
    sign = 1;
    System.out.println("Now yellow sign");
  }
  public void setGreen() {
    sign = 2;
    System.out.println("Now green sign");
  }
  public int getStatus() {
    return sign;
  }
        public void update(Observable o, Object arg) {
                //TODO : Implements what to do when observed obj
                int status = ((SignCommand)o).getStatus();
                if(status == 0) //red
                  setRed();
                else if(status == 1) //yellow
                  setYellow();
                else if(status == 2) //green
                  setGreen();
        }
}
```

รูปที่ 5.43 ชุดคำสั่งในคลาส SignDriver หลังทำการเขียนชุดคำสั่งเพิ่ม

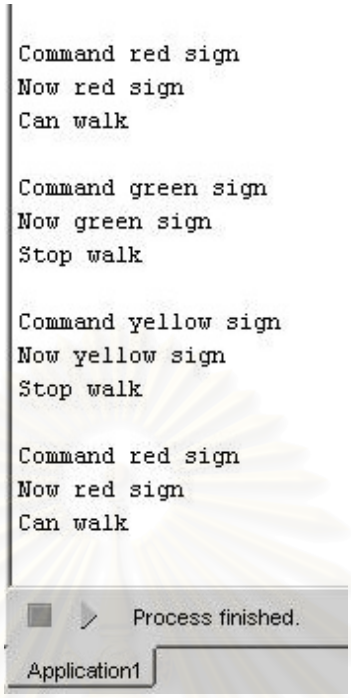
ถัดไปเขียนโปรแกรมเพื่อทดสอบคลาสดังกล่าว โดยสร้างวัตถุของคลาสที่สามขึ้นมา แล้วทดสอบเปลี่ยนแปลงสัญญาณไฟ โปรแกรมทดสอบเขียนได้ดังรูปที่ 5.44

หลังจากนั้นทำการแปลโปรแกรมและดำเนินงาน ได้ผลการทดสอบถูกต้องดังแสดงในรูปที่ 5.45

```
SignCommand comm = new SignCommand();
SignCross cross = new SignCross();
SignDriver drive = new SignDriver();
comm.addObserver(cross);
comm.addObserver(drive);

comm.setRed();
comm.setGreen();
comm.setYellow();
comm.setRed();
```

รูปที่ 5.44 ชุดคำสั่งที่ใช้ในการทดสอบคลาส SignCommand SignCross และ SignDriver



รูปที่ 5.45 ผลการทดสอบคลาส SignCommand SignCross และ SignDriver

บทที่ 6

สรุปผลการวิจัยและข้อเสนอแนะ

ในบทนี้กล่าวถึงผลสรุปของงานวิจัย ปัญหาและข้อจำกัดต่างๆ ของงานวิจัย รวมทั้งข้อเสนอแนะที่สามารถนำไปปรับปรุงและพัฒนาได้ต่อไปในอนาคต โดยมีรายละเอียดดังต่อไปนี้

6.1 สรุปผลการวิจัย

งานวิจัยนี้ได้ทำการออกแบบ และพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบด้วยจาวาบีน และสร้างเครื่องมือช่วยการพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ โดยผลที่ได้รับคือ ส่วนประกอบซอฟต์แวร์ที่สร้างชุดคำสั่งสำหรับรูปแบบการออกแบบทั้ง 23 ชนิด ตามหนังสือ Design Patterns: Element of Reusable Object-Oriented Software [1] และสามารถนำไปใช้กับเครื่องมือช่วยการพัฒนาแบบเห็นภาพที่เป็นที่นิยมในปัจจุบันได้ ซึ่งจะช่วยให้การพัฒนาโปรแกรมเชิงวัตถุเพื่อแก้ไขปัญหาที่พบบ่อยตามรูปแบบการออกแบบ ทำให้สามารถนำแนวคิดของรูปแบบการออกแบบกลับมาใช้ใหม่ได้ โดยไม่ต้องเขียนชุดคำสั่งของรูปแบบการออกแบบใหม่ทุกครั้งที่พัฒนาโปรแกรม แต่เป็นเพียงการใช้ส่วนประกอบซอฟต์แวร์ซึ่งช่วยในการสร้างชุดคำสั่งของรูปแบบการออกแบบเท่านั้น

หลังจากพัฒนาส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ ได้ทดสอบการใช้ส่วนประกอบซอฟต์แวร์นี้ในการพัฒนาซอฟต์แวร์จำนวน 3 ซอฟต์แวร์ ผลการทดสอบพบว่า ส่วนประกอบซอฟต์แวร์นี้สามารถช่วยในการเขียนโครงชุดคำสั่งของรูปแบบการออกแบบซึ่งเป็นโครงสร้างหลักของโปรแกรมได้อย่างถูกต้อง

6.2 ข้อจำกัดของงานวิจัย

6.2.1 รูปแบบการออกแบบบางรูปแบบสามารถเขียนแบบที่เป็นกลางได้หลายแบบ ดังนั้นถึงแม้ว่าผู้วิจัยจะเลือกรูปแบบของชุดคำสั่งที่เป็นกลางแล้ว แต่ก็ยังสามารถเขียนได้อีกหลายแบบ ซึ่งเป็นการจำกัดผู้ใช้ให้ต้องใช้รูปแบบการเขียนชุดคำสั่งเพียงแบบเดียว รวมถึงโปรแกรมที่ต้องการการเขียนเฉพาะตัว ไม่สามารถใช้ชุดคำสั่งที่เป็นกลางของรูปแบบการออกแบบได้ อาจนำส่วนประกอบซอฟต์แวร์นี้ไปใช้งานไม่ได้

6.2.2 การใช้งานของรูปแบบการออกแบบมากกว่าหนึ่งรูปแบบผสานกัน อาจยังมีปัญหาในการใช้งานอยู่บ้าง เนื่องจากบางครั้งชุดคำสั่งต้องเป็นคลาสและบางครั้งเป็นตัวต่อประสาน

6.2.3 รูปแบบการออกแบบบางรูปแบบมีความหลากหลายที่สูงมาก เช่น รูปแบบการออกแบบฟาซาด การสร้างชุดคำสั่งที่เป็นกลางจึงเป็นชุดคำสั่งที่ช่วยการพัฒนาโปรแกรมได้ไม่มาก

6.2.4 ปัจจุบันได้มีการคิดค้นรูปแบบการออกแบบที่พบได้บ่อยๆ มากขึ้น รูปแบบการออกแบบ 23 รูปแบบในวิทยานิพนธ์นี้จึงอาจไม่เพียงพอ

6.3 **ข้อเสนอแนะ**

ควรเพิ่มเติมรูปแบบชุดคำสั่งที่สามารถสร้างได้ให้มีหลายแบบ เช่น สร้างแบบขยายคลาสหรือใช้ตัวต่อประสาน ให้ผู้ใช้เลือกใช้ได้ รวมถึงเพิ่มเติมส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบใหม่ๆ ในปัจจุบัน โดยสร้างคลาส DesignPattern และ DesignPatternBeanInfo ดังรูปที่ 3.2 เพิ่มเติมตามจำนวนรูปแบบการออกแบบที่ต้องการเพิ่มเติม

รายการอ้างอิง

1. Erich Gamma. Design Patterns, Elements of Reusable Object-Oriented Software. (n.p.): Addison-Wesley, 1995.
2. Sherman R. Alpert. The Design Patterns, Smalltalk Companion. (n.p.): Addison-Wesley, 1998.
3. James W. Cooper. The Design Patterns, Java Companion. (n.p.), 1998.
4. Mark Grand. Patterns in Java Volumn 1, A Catalog of Reusable Design Patterns Illustrated with UML. Wiley, 1998.
5. Jukka Viljamaa. Tool Supporting the Use of Design Patterns in Frameworks. Report, Department of Computer Science, University of Helsinki, 1997.
6. Theo dirk Maijler, Serge Demeyer, Robert Engel. Making Design Patterns Explicit in FACE. Washington University, 1996.
7. Sun Microsystems. JavaBean™ [Online]. Available from: http://java.sun.com/docs/books/tutorial/javabeans/ [Dec 1998].
8. Sun Microsystems. Java™ 2 SDK, Standard Edition Documentation V 1.4.1, 2002.
9. Sun Microsystems. JavaBeans Specification[Online]. Available from: http://java.sun.com/beans/ [Dec 1998]
10. นิเวศน์ จรัสดำรง. โครงร่างและแบบอย่างการออกแบบเพื่อพัฒนาโปรแกรมประยุกต์.วิทยานิพนธ์ปริญญามหาบัณฑิต ภาควิชาวิศวกรรมคอมพิวเตอร์ บัณฑิตวิทยาลัย จุฬาลงกรณ์มหาวิทยาลัย, 2541.
11. ดุลยรัตน์ กรณฑ์แสง. การสร้างสิ่งแวดล้อมสำหรับพัฒนาโปรแกรมด้วยแผนภาพเอนทิตีและความสัมพันธ์ด้วยจาวาบีน. โครงร่างวิทยานิพนธ์ ภาควิชาวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย, 2541.

ภาคผนวก

ภาคผนวก ก

การใช้งานส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

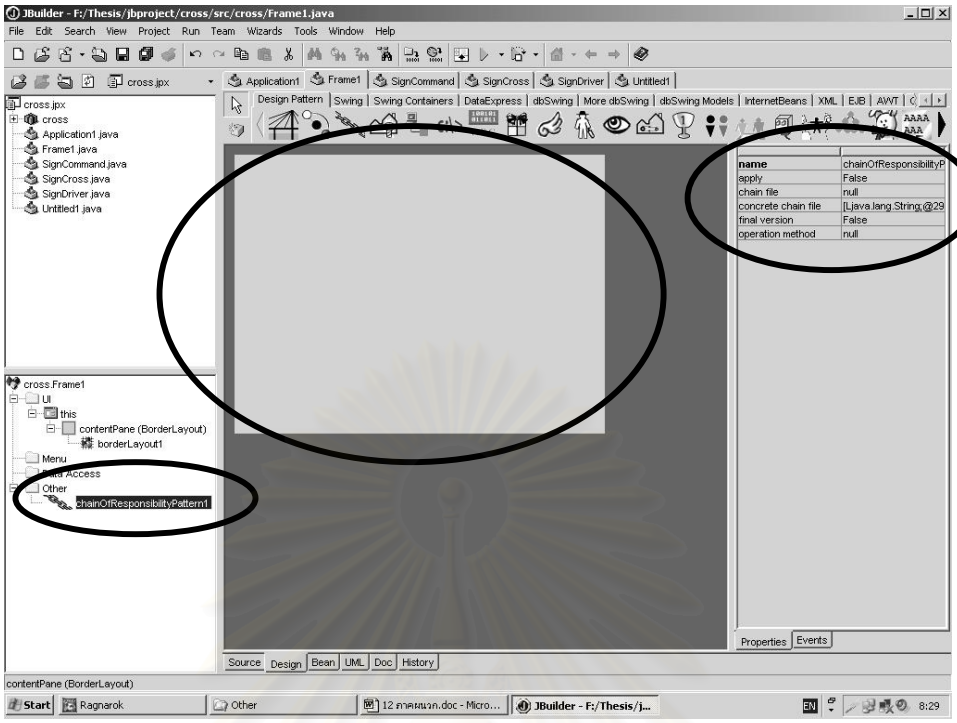
ในการนำส่วนประกอบซอฟต์แวร์ทั้งหมด 23 ส่วนประกอบซอฟต์แวร์ไปใช้งานนั้น จะต้องนำมารวมกันเป็นแฟ้มข้อมูลเดียวด้วยแฟ้มรวมข้อมูลจาวา (Java Archive File หรือ JAR) ซึ่งผู้วิจัยของใช้ชื่อแฟ้มข้อมูลว่า designpattern.jar จากนั้นจะต้องทำการติดตั้งลงในเครื่องมือช่วยการพัฒนาแบบเห็นภาพ

การติดตั้งลงในเครื่องมือช่วยการพัฒนาแบบเห็นภาพ ซึ่งในที่นี้ใช้เจบิวเดอร์ รุ่นที่ 6 ทำได้โดย เข้าถึงรายการ Tools → Configure Libraries เพื่อสร้างคลัง (Library) ขึ้นใหม่ แล้วเลือกแฟ้มรวมข้อมูลจาวาของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบให้กับคลังใหม่นี้ ซึ่งในที่นี้คือ designpattern.jar เพื่อให้เครื่องมือช่วยการพัฒนาแบบเห็นภาพรู้จักแพกเกจนี้ ถัดไปให้เข้าถึงรายการ Tools → Configure Palette เลือกไปที่คลังใหม่ที่ได้สร้างไว้ แล้วทำการใส่บีนของรูปแบบการออกแบบทั้ง 23 รูปแบบให้แสดงรายชื่อบนกล่องเครื่องมือในเครื่องมือช่วยการพัฒนาแบบเห็นภาพ ดังรูปที่ ก.1

รูปที่ ก.1 ลักษณะของกล่องเครื่องมือ

การใช้งานเมื่อติดตั้งเสร็จสิ้นแล้ว ทำได้โดยคลิก (Click) ที่รูปของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบที่ต้องการตามตารางที่ 3.1 แล้วคลิกอีกครั้งบนฟอร์มของโปรแกรมตรงกลาง (ดูรูปที่ ก.2 ประกอบ) หลังจากนั้นจะปรากฏรายชื่อของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบที่เลือกใช้ในโปรแกรมแล้วทางหน้าต่างด้านซ้ายล่าง และเมื่อเลือกรายชื่อของส่วนประกอบซอฟต์แวร์ทางหน้าต่างด้านซ้ายล่างแล้ว จะปรากฏคุณลักษณะที่แก้ไขได้ทางหน้าต่างด้านขวา หลังจากนั้น ผู้ใช้สามารถทำการแก้ไขคุณลักษณะตามต้องการได้ ซึ่งจะนำไปสู่การสร้างชุดคำสั่งของรูปแบบการออกแบบลงในโปรแกรมที่กำลังพัฒนาต่อไป

อนึ่ง การใช้งานอาจแตกต่างกันไปขึ้นกับ ชนิด และ รุ่น ของเครื่องมือช่วยการพัฒนาแบบเห็นภาพที่ใช้งาน



รูปที่ ก.2 ลักษณะของการใช้งานส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

ภาคผนวก ข

รายชื่อเมท็อดที่ใช้ในการสร้าง เปลี่ยนแปลง ลบ ชุดคำสั่ง

เมท็อดเหล่านี้อยู่ในคลาส SourceCodeManipulation ดังที่แสดงในรูปที่ 3.3 โดย เป็นเมท็อดแบบอารักขา และส่งคืนค่าเป็นแบบบูล ซึ่งเป็นการระบุว่าเมท็อดได้ทำการเปลี่ยนแปลงชุดคำสั่งหรือไม่

รายชื่อเมท็อดมีดังต่อไปนี้

replaceSourceCode(String sdata, String oldSrc, String newSrc, String className)

ใช้แทนที่ชุดคำสั่ง sdata ในคลาส className ที่มีส่วนชุดคำสั่ง oldSrc ด้วย ส่วนชุดคำสั่ง newSrc

addImplements(String sdata, String impInterface, String className) throws ParseException

ใช้เพิ่มชุดคำสั่งการอิมพลีเมนต์ implements <impInterface> ในชุดคำสั่ง sdata ของคลาส className

deleteImplements(String sdata, String impInterface, String className) throws ParseException

ใช้ลบชุดคำสั่งการอิมพลีเมนต์ implements <impInterface> ในชุดคำสั่ง sdata ของคลาส className

addExtends(String sdata, String extToAdd, String className) throws ParseException

ใช้เพิ่มชุดคำสั่งการขยาย extends <extToAdd> ในชุดคำสั่ง sdata ของคลาส className

deleteExtends(String sdata, String extToAdd, String className) throws ParseException

ใช้ลบชุดคำสั่งการขยาย extends <extToAdd> ในชุดคำสั่ง sdata ของคลาส className

addJavaLib(String sdata, String libToAdd)

ใช้เพิ่มชุดคำสั่งการนำเข้า import <libToAdd>; ในชุดคำสั่ง sdata

addField(String sdata, String fieldToAdd, String className)

ใช้เพิ่มชุดคำสั่งการประกาศเขตข้อมูล fieldToAdd ในชุดคำสั่ง sdata ของคลาส className

deleteField(String sdata, String fieldToDel, String className)

ใช้ลบชุดคำสั่งการประกาศเขตข้อมูล fieldToDel ในชุดคำสั่ง sdata ของคลาส className

addMethod(String sdata, String methodToAdd, String className)

ใช้เพิ่มชุดคำสั่งการประกาศเมท็อด methodToAdd ในชุดคำสั่ง sdata ของคลาส className

addMethod(String sdata, String methodToAdd, String headerForCheck, String className)

ใช้เพิ่มชุดคำสั่งการประกาศเมท็อด methodToAdd โดยตรวจสอบกับส่วนหัวของเมท็อดในชุดคำสั่งเดิมก่อนว่ามีอยู่แล้วหรือไม่ ในชุดคำสั่ง sdata ของคลาส className

deleteMethod(String sdata, String headerForCheck, byte type, String className) throws ParseException

ใช้ลบชุดคำสั่งการประกาศเมท็อดโดยตรวจสอบกับส่วนหัวของเมท็อด headerForCheck ว่าตรงกันหรือไม่เท่านั้น หากตรงกันจะลบโดยไม่สนใจส่วนตัวของเมท็อด ในชุดคำสั่ง sdata ของคลาส className

deleteMethod(String sdata, String methodToDel, String className)

ใช้ลบชุดคำสั่งการประกาศเมท็อดโดยตรวจสอบทุกประโยคในเมท็อด methodToDel ในชุดคำสั่ง sdata ของคลาส className

addInnerClass(String sdata, String classToAdd, String className)

ใช้เพิ่มชุดคำสั่งคลาสภายใน classToAdd ในชุดคำสั่ง sdata ของคลาส className

deleteInnerClass(String sdata, String classToDel, String className)

ใช้ลบชุดคำสั่งคลาสภายใน classToAdd ในชุดคำสั่ง sdata ของคลาส className

makeClassAbstract(String sdata, String className) throws ParseException

ใช้เปลี่ยนแปลงชุดคำสั่งคลาส className ในชุดคำสั่ง sdata ให้เป็นคลาสนามธรรม

unmakeClassAbstract(String sdata, String className) throws ParseException

ใช้เปลี่ยนแปลงชุดคำสั่งคลาส className ในชุดคำสั่ง sdata ซึ่งเป็นคลาสนามธรรมให้เป็นคลาสปกติ

การเรียกใช้เมท็อดเหล่านี้จะกระทำโดยเมท็อด adjustStructure() และเมท็อด deleteStructure() ของคลาส DesignPattern ในรูปที่ 3.3 ซึ่งเป็นคลาสของส่วนประกอบซอฟต์แวร์ในวิทยานิพนธ์นี้ เมื่อทำการเพิ่มเติม เปลี่ยนแปลง หรือลบชุดคำสั่งด้วยเมท็อดดังกล่าวแล้ว จะใช้เมท็อด protected String getNewSourceCode() ในการขอชุดคำสั่งซึ่งแก้ไขแล้วกลับคืน

นอกจากนี้ ยังมีเมท็อดที่สำคัญอีกหนึ่งเมท็อดซึ่งจะใช้ในการตรวจสอบความถูกต้องของชุดคำสั่งภาษาจาวาเบื้องต้นก่อนทำการเพิ่มเติม เปลี่ยนแปลง หรือลบชุดคำสั่ง

validateSourceCode(String srcCode, String className, byte type) throws IOException, ParseException

ภาคผนวก ค

ตัวอย่างชุดคำสั่งของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบ

ชุดคำสั่งต่อไปนี้เป็นชุดคำสั่งของส่วนประกอบซอฟต์แวร์ของรูปแบบการออกแบบมีเมนโต

```
package designpattern;

import java.beans.PropertyVetoException;

public final class MementoPattern extends DesignPatternComponent {
        private String fileName = null;
        private boolean apply = false;
        private boolean finalVersion = false;

        private String className = null;
        private SingleFileHandle pattFile = null;

        public MementoPattern() {
                pattFile = new SingleFileHandle(this);
        }

        protected void adjustStructure() throws Exception {
                boolean dirty = false;
                String sdata = pattFile.getValidSourceCode();

                { //add implements Cloneable
                        if(addImplements(sdata, "Cloneable", className)) {
                                dirty = true;
                                sdata = getNewSourceCode();
                        }
                }

                { //add method createMemento(), check for repeat add
                final String createMementoBody = INDENT + "public Object createMemento() {\r\n" +
INDENT + INDENT + "try {\r\n" +
INDENT + INDENT + INDENT + "return new Memento((" + className + ")this.clone());\r\n" +
INDENT + INDENT + "}catch(CloneNotSupportedException e) {e.printStackTrace();}\r\n" +
INDENT + INDENT + "return null;\r\n" +
INDENT + "}\r\n";
```

```
                        if(addMethod(sdata, createMementoBody, className)) {
                                dirty = true;
                                sdata = getNewSourceCode();
                        }
                }

                { //add method setMemento(Memento m), check for repeat add
                        final String setMementoHeader = "public void setMemento(Object obj)";
                        final String setMementoBody = INDENT + setMementoHeader + " {\r\n" +
INDENT + INDENT + "if(obj instanceof Memento) {\r\n" +
INDENT + INDENT + INDENT + className + " save = ((Memento)obj).state;\r\n\r\n" +
INDENT + INDENT + INDENT + "//TODO: Restore fields which you need.\r\n" +
INDENT + INDENT + INDENT + "//Example: this.fieldname = save.fieldname;\r\n\r\n" +
INDENT + INDENT + "}\r\n" +
INDENT + "}\r\n";
                        if(addMethod(sdata, setMementoBody, setMementoHeader, className)) {
                                dirty = true;
                                sdata = getNewSourceCode();
                        }
                }

                { //add inner class Memento
                        final String memento = INDENT + "public final class Memento {\r\n" +
INDENT + INDENT + "private final " + className + " state;\r\n" +
INDENT + INDENT + "private Memento(" + className + " state) {\r\n" +
INDENT + INDENT + INDENT + "this.state = state;\r\n" +
INDENT + INDENT + "}\r\n\t}\r\n";
                        if(addInnerClass(sdata, memento, className)) {
                                dirty = true;
                                sdata = getNewSourceCode();
                        }
                }

                if(dirty) {
                        pattFile.saveFile(sdata);
                }
        }

        protected void deleteStructure() throws Exception {
                boolean dirty = false;
                String sdata = pattFile.getValidSourceCode();
```

```
{ //delete method createMemento(), if it same as I've added(ignore comment), delete it
final String createMementoBody = INDENT + "public Object createMemento() {\r\n" +
INDENT + INDENT + "try {\r\n" +
INDENT + INDENT + INDENT + "return new Memento((" + className + ")this.clone());\r\n" +
INDENT + INDENT + "}catch(CloneNotSupportedException e) {e.printStackTrace();}\r\n" +
INDENT + INDENT + "return null;\r\n" +
INDENT + "}\r\n";
if(deleteMethod(sdata, createMementoBody, className)) {
dirty = true;
sdata = getNewSourceCode();
}
}

{ //delete method setMemento(Memento m)
final String setMementoHeader = "public void setMemento(Object obj)";
if(deleteMethod(sdata, setMementoHeader, HEADER, className)) {
dirty = true;
sdata = getNewSourceCode();
}
}

{ //delete inner class Memento
final String memento = INDENT + "public final class Memento {\r\n" +
INDENT + INDENT + "private final " + className + " state;\r\n" +
INDENT + INDENT + "private Memento(" + className + " state) {\r\n" +
INDENT + INDENT + INDENT + "this.state = state;\r\n" +
INDENT + INDENT + "}\r\n" +
INDENT + "}\r\n";
if(deleteInnerClass(sdata, memento, className)) {
dirty = true;
sdata = getNewSourceCode();
}
}

if(dirty) { //delete implements Cloneable, if and only if other components of this pattern -
also delete
if(deleteImplements(sdata, "Cloneable", className)) {
//dirty = true; //true already
sdata = getNewSourceCode();
}
```

```
		}

		if(dirty) {
			pattFile.saveFile(sdata);
		}
	}

	protected boolean isPropertiesNotNull() {
		return (className != null) && !className.equals("");
	}

	//Bean Interface
	public void setFileName(String fileName) throws Exception { //fileName
		if(finalVersion)
			return;

		String newVal = fileName.trim();
		String className = FileManipulation.getNoExtensionFileName(newVal);
		fireVetoableChange("className", null, className);

		if(apply && isPropertiesNotNull()) {
			applyEngine(false, null);
			this.className = className;
			this.fileName = newVal;
			pattFile.setupFile(newVal, className);
			applyEngine(true, null);
		}else {
			this.className = className;
			this.fileName = newVal;
			pattFile.setupFile(newVal, className);
		}
	}
	public String getFileName() {
		return fileName;
	}

	public void setApply(boolean apply) throws PropertyVetoException { //apply
		if(finalVersion)
			return;

		if(apply || this.apply) {
```

```
                        fireVetoableChange("apply", new Boolean(this.apply), new Boolean(apply));
                        this.apply = apply;
                }
        }
        public boolean getApply() {
                return apply;
        }

        public void setFinalVersion(boolean finalVersion) { //finalVersion
                this.finalVersion = finalVersion;
        }
        public boolean getFinalVersion() {
                return finalVersion;
        }
}
```

ประวัติผู้เขียนวิทยานิพนธ์

นายกิรพัฒน์ วิเชียรกิจ เกิดเมื่อวันที่ 24 เมษายน พ.ศ. 2521 สำเร็จการศึกษาหลักสูตรบริหารธุรกิจบัณฑิต สาขาวิชาระบบสารสนเทศเพื่อการจัดการ จากคณะพาณิชยศาสตร์และการบัญชี มหาวิทยาลัยธรรมศาสตร์ เมื่อปีการศึกษา 2541 และเข้าศึกษาต่อในหลักสูตรวิทยาศาสตร์มหาบัณฑิต ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย เมื่อปีการศึกษา 2542