บทที่ 2

ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

2.1 ทฤษฎีที่เกี่ยวข้อง

ทฤษฎีที่เกี่ยวข้องในงานวิจัยนี้ ได้แก่ ดีไซน์แพทเทิร์น (Design Patterns) และอ็อบเจกต์โมเดลที่สามารถปรับเปลี่ยนได้ (Adaptive Object Model), ยูนิตโมเดล (Unit Model) และการวัดซอฟต์แวร์เชิงวัตถุ (Object-oriented software measurement)

2.1.1 ดีไซน์แพทเทิร์น

ในกระบวนการวิศวกรรมซอฟต์แวร์ (Software Engineering) ดีไซน์แพทเทิร์นเป็นแนวทางทั่วไปที่นำมาใช้เพื่อแก้ปัญหาที่เกิดขึ้นเป็นประจำในการออกแบบซอฟต์แวร์ ดีไซน์แพทเทิร์นไม่ได้เป็นการออกแบบที่สำเร็จรูปซึ่งสามารถถูกเปลี่ยนรูปไปสู่โค้ดของโปรแกรมได้อย่างตรงๆ แต่เป็นคำอธิบายหรือตัวแบบสำหรับแก้ปัญหาซึ่งสามารถถูกนำไปใช้ในสถานการณ์ที่แตกต่างกัน โดยปกติแล้วดีไซน์แพทเทิร์นจะแสดงความสัมพันธ์และการติดต่อระหว่างคลาสหรืออ็อบเจกต์ที่มีในระบบ

ดีไซน์แพทเทิร์นถูกจำแนกได้ด้วยเกณฑ์การแบ่งที่หลากหลาย เกณฑ์พื้นฐานที่ใช้แบ่งลักษณะ คือ แบ่งตามปัญหาที่ใช้ดีไซน์แพทเทิร์นแก้ปัญหา จากเกณฑ์การแบ่งนี้ ดีไซน์แพทเทิร์นสามารถจำแนกได้หลากหลายลักษณะ ดังนี้ Fundamental patterns, Creational patterns, Structural patterns, Behavioral patterns, Concurrency patterns, Event handling patterns และ Architectural patterns

เอกสารของดีไซน์แพทเทิร์นจะมีข้อมูลที่อธิบายเกี่ยวกับปัญหาที่สนใจ สถานการณ์ที่จะนำไปใช้และแนวทางในการแก้ปัญหา ฉะนั้นผู้เขียนที่ต้องการเผยแพร่ดีไซน์แพทเทิร์นควรใช้รูปแบบในการจัดทำเอกสารเพื่ออธิบายสิ่งที่จำเป็นสำหรับนำไปใช้ให้ครบถ้วน รูปแบบที่ใช้โดยปกติ ส่วนใหญ่จะเป็นแบบเดียวกับที่ใช้โดย E. Gamma และคณะ ซึ่งจะประกอบไปด้วยส่วนต่างๆ ดังนี้

- Pattern Name and Classification: ทุกแพทเทิร์นควรมีชื่อที่อธิบายและเป็นหนึ่งเดียวไม่ซ้ำเพื่อใช้ในการอ้างถึงได้ นอกจากนี้ ควรถูกจำแนกตามหมวดหมู่ข้างต้น หมวดหมู่จะช่วยให้นำใช้ไปใช้ได้อย่างถูกต้องตามวัตถุประสงค์
- Intent : ส่วนที่อธิบายถึงเป้าหมายและเหตุผลสำหรับการนำไปใช้
- Also Known As: แพทเทิร์นสามารถมีได้มากกว่าหนึ่งชื่อ ชื่อเหล่านี้ควรจะบันทึกเป็นเอกสารไว้ในส่วนนี้

- Motivation : เป็นส่วนที่อธิบายเหตุการณ์จำลองของปัญหาและบริบทที่ควรจะนำแพทเทิร์นไปใช้
- Applicability : อธิบายถึงสถานการณ์ที่แพทเทิร์นสามารถนำไปใช้ประโยชน์ได้
- Structure : การนำเสนอแพทเทิร์นในรูปแบบภาพ กราฟฟิก เช่น แผนภาพคลาส แผนภาพซีเควนซ์
- Participants : รายชื่อคลาสและอ็อบเจกต์ทุกตัวที่ถูกใช้ในแพทเทิร์นนี้ และบทบาทของคลาสและอ็อบเจกต์ที่ใช้ในการออกแบบ
- Collaboration : อธิบายว่าคลาสและอ็อบเจกต์ทุกตัวที่ใช้ในแพทเทิร์นนี้ มีความสัมพันธ์หรือผลกระทบต่อกันอย่างไร
- Consequences : อธิบายถึงผลกระทบ ข้อดี ข้อเสียของการใช้แพทเทิร์นนี้
- Implementation : อธิบายถึงวิธีการอิมพลีเมนท์ นำเสนอวิธีการแก้ปัญหาของแพทเทิร์น และแนะนำเทคนิคที่ใช้
- Sample Code: แสดงตัวอย่างการใช้งานในรูปแบบของภาษาต่างๆที่ใช้ในการเขียนโปรแกรม
- Known Uses: ตัวอย่างของการใช้งานจริง
- Related Patterns: แพทเทิร์นที่มีความสัมพันธ์เกี่ยวข้อง หรือคล้ายกันกับแพทเทิร์นนี้

2.1.2 อ็อบเจกต์โมเดลที่ปรับเปลี่ยนได้ (Adaptive Object Model - AOM) [4]

ในการพัฒนาซอฟต์แวร์เชิงวัตถุ การออกแบบอ็อบเจกต์โมเดลเป็นขั้นตอนที่สำคัญอย่างหนึ่ง เนื่องจากอ็อบเจกต์โมเดลจะสะท้อนถึงความเข้าใจ ความหมาย ที่สอดคล้องกับธรรมชาติของธุรกิจ อย่างไรก็ตาม เนื่องจากสภาพการแข่งขันในโลกของธุรกิจ หรือด้วยเหตุผลใดๆก็ตาม ที่ส่งผลให้เกิดความจำเป็นที่จะต้องมีการเปลี่ยนแปลงเงื่อนไขหรือกฎเกณฑ์ทางธุรกิจอยู่เสมอ ความต้องการที่เปลี่ยนแปลงไป ทำให้ซอฟต์แวร์เดิมนั้นล้าสมัยได้อย่างรวดเร็ว เนื่องจากการออกแบบอ็อบเจกต์โมเดลโดยทั่วไปนั้นเป็นการออกแบบโมเดลที่มีโครงสร้างแบบคงที่ อ็อบเจกต์โมเดลที่ได้จึงมีโครงสร้างที่เปลี่ยนแปลงแก้ไขในภายหลังได้ยาก การเปลี่ยนแปลงอ็อบเจกต์โมเดลจะต้องทำการแก้ไขโปรแกรมเสมอ ทำให้ไม่สามารถตอบสนองต่อความต้องการได้ทันเวลา ดังนั้น การออกแบบให้อ็อบเจกต์โมเดลสามารถที่จะปรับเปลี่ยนได้ง่ายจะเป็นวิธีที่จะเข้ามาช่วยให้ซอฟต์แวร์นั้นสามารถรองรับความต้องการที่จะเกิดขึ้นในอนาคตได้ดีขึ้น

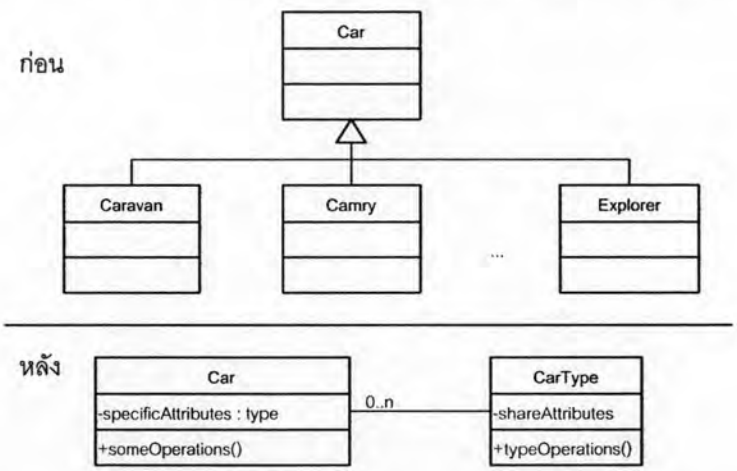
การออกแบบอ็อบเจกต์โมเดลโดยทั่วไปนั้นจะทำการออกแบบโดยการสร้างคลาสสำหรับเอนทิตี แอตทริบิวต์ เมท็อดและความสัมพันธ์ กำหนดไว้ในโมเดล แต่เอโอเอ็มนั้นจะนำเสนอสิ่งเหล่านั้นให้อยู่ในรูปแบบของเมตาดาต้า (Metadata) แล้วค่อยแปลความหมายเหล่านั้นขณะรันไทม์ หรือกล่าวได้

ว่าอ็อบเจกต์โมเดลถูกเก็บไว้ในรูปแบบของข้อมูล ดังนั้นเมื่อมีการเปลี่ยนแปลงแก้ไข อ็อบเจกต์โมเดลก็จะเปลี่ยนแปลงได้ทันที

เอโอเอ็มนั้นสร้างมาจากการใช้ดีไซน์แพทเทิร์นหลายๆชนิดมาประกอบกัน ดีไซน์แพทเทิร์นที่สำคัญประกอบด้วย ไทป์อ็อบเจกต์, พรอบเพอร์ตี, สตราทิจิ และ แอ็คเคาน์ทะบิลิตี เป็นต้น

- ไทป์อ็อบเจกต์

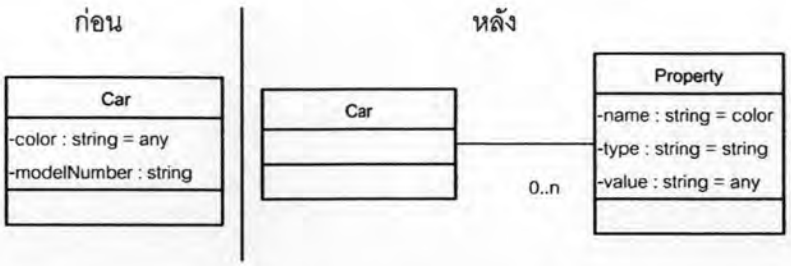
  โดยปกติภาษาเชิงวัตถุจะกำหนดโครงสร้างของโปรแกรมในรูปแบบกลุ่มของคลาส คลาสจะเป็นตัวกำหนดโครงสร้างและพฤติกรรมของอ็อบเจกต์ คลาสแต่ละคลาสจะแทนประเภทของ อ็อบเจกต์ที่แตกต่างกันไป ดังนั้นหากต้องการเพิ่มคลาสใหม่เข้าไปในระบบจำเป็นที่จะต้องเขียนโปรแกรมเพิ่มเติม ดังนั้นในระบบที่มีขนาดใหญ่มากปัญหาจะเกิดขึ้นเนื่องจากเราจะไม่สามารถรู้ได้ว่าจะมีคลาสใหม่เพิ่มขึ้นเมื่อใด ดังนั้นจึงได้มีการกำหนดวิธีการแก้ปัญหานี้ขึ้นมาคือ การแบ่งคลาสเดิมออกเป็นสองคลาส โดยที่คลาสหนึ่งเป็นตัวแทนของเอนทีตี (Entities) และอีกคลาสหนึ่งเป็นตัวแทนของประเภทเอนทีตี (EntityTypes) ยกตัวอย่างเช่น คลาสแม่ที่ชื่อว่า Car และกลุ่มของคลาสลูกเช่น Caravan, Camry, Explorer ซึ่งสามารถออกแบบใหม่ให้มีสองคลาสได้ดังนี้คือ Car และ CarType ดังแสดงในรูปที่ 2.1



รูปที่ 2.1 ไทป์อ็อบเจกต์ [3]

- พรอบเพอร์ตี

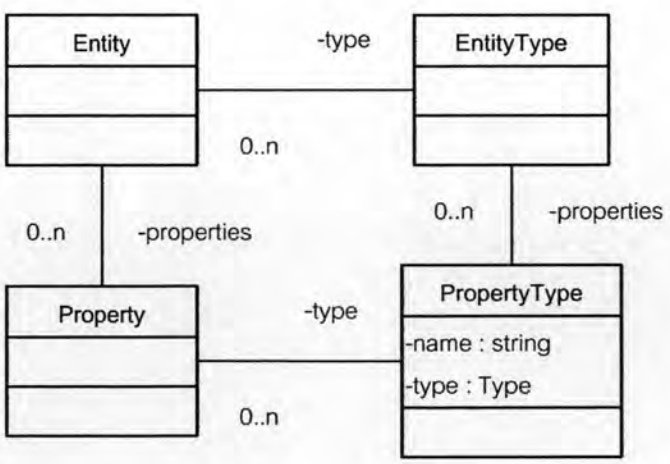
แอตทริบิวต์ของอ็อบเจกต์โดยปกติจะกำหนดให้เป็นตัวแปรแบบอินสแตนท์ (Instance Variable) การที่จะทำให้แอตทริบิวต์ของคลาสนั้นเปลี่ยนแปลงได้ เช่น การเพิ่มลดจำนวนแอตทริบิวต์ จะไม่สามารถทำได้ ดังนั้นจึงได้กำหนดรูปแบบการแก้ปัญหานี้ขึ้นมาโดยการสร้างคลาสชื่อว่า Property ซึ่งเปรียบเหมือนคลาสที่อธิบายว่าคลาส Car จะประกอบด้วยแอตทริบิวต์อะไรบ้าง ภายในคลาส Property มีแอตทริบิวต์ซึ่งประกอบด้วย name, type และ value เพื่ออธิบายแอตทริบิวต์ของคลาสที่สนใจ ดังรูปที่ 2.2



รูปที่ 2.2 พรอบเพอร์ตี [4]

จากรูปที่ 2.2 คลาส Car ก่อนใช้ดีไซน์แพทเทิร์น ประกอบด้วยแอตทริบิวต์ชื่อ color โดยมีค่าเริ่มต้นเท่ากับ any และแอตทริบิวต์ชื่อ modelNumber เมื่อแก้ปัญหาโดยใช้ พรอบเพอร์ตีแพทเทิร์น ทำให้แต่ละแอตทริบิวต์ของคลาส Car จะถูกแปลงให้อยู่ในรูปของความสัมพันธ์กับคลาส Property แบบ 0..n ดังตัวอย่างจะเห็นได้ว่าแอตทริบิวต์เดิมที่ชื่อ color จะกลายเป็นอินสแตนท์ของคลาส Property โดยมีค่าแอตทริบิวต์ของ name เท่ากับ color, type เท่ากับ string และ value เท่ากับ any

ในการออกแบบเอโอเอ็มโดยปกติแล้วนั้นมักจะต้องทำการประยุกต์ใช้รูปแบบของ Type Object และ Property เข้าด้วยกันคือ เริ่มแรกจะต้องทำการประยุกต์ใช้ไทป์อ็อบเจกต์แพทเทิร์นเพื่อแยกประเภทของคลาสก่อน จากนั้นจึงใช้พรอบเพอร์ตีแพทเทิร์น เพื่อแยกแอตทริบิวต์ของเอนทิตีออกมา แต่เนื่องจากคลาส Property และ EntityType เองนั้นก็มี แอตทริบิวต์ที่เป็น Type อยู่ภายใน จึงต้องประยุกต์ใช้ไทป์อ็อบเจกต์แพทเทิร์นกับคลาส Property อีกครั้งหนึ่งเพื่อแยก Type ออกมาจึงได้คลาสที่ชื่อว่า PropertyType อีกหนึ่งคลาส เรียกรูปแบบความสัมพันธ์นี้ว่าเป็นแบบไทป์สแควร์ (Type Square) แพทเทิร์นนี้มักถูกนำมาใช้เป็นแพทเทิร์นหลักในการออกแบบสำหรับเอโอเอ็ม ดังรูปที่ 2.3



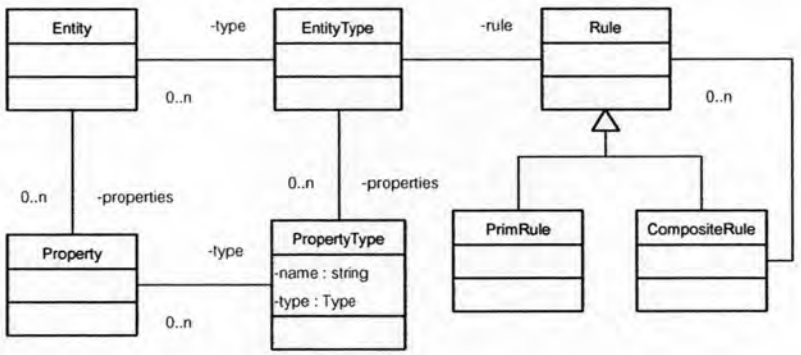
รูปที่ 2.3 ไทป์สแควร์ [4]

- สตราทิจิ

เพื่อให้อ็อบเจกต์นั้นสามารถที่จะปรับเปลี่ยนพฤติกรรม อัลกอริทึม หรือเพิ่มลดเมท็อดได้ เอโอเอ็มจึงได้นำเอาสตราทิจิแพทเทิร์นมาใช้ โดยกำหนดอินเตอร์เฟสมาตรฐานของพฤติกรรมเงื่อนไข หรือ กฎเกณฑ์ของอ็อบเจกต์ ซึ่งในที่นี้ก็คือคลาส Rule และมีความสัมพันธ์กับคลาส EntityType ที่ทำหน้าที่เป็นบริบท (Context) ซึ่งจะเลือกใช้อ็อบเจกต์ของคลาส Rule ที่เหมาะสมและสอดคล้องกับบริบท โดยเรียกใช้งานผ่านทางเมท็อดที่กำหนดให้เป็นมาตรฐาน ดังแสดงในรูปที่ 2.4

คลาส PrimRule จะเป็นคลาสลูกของคลาส Rule ที่อิมพลิเมนต์อินเตอร์เฟสมาตรฐาน เมื่อต้องการเพิ่มเมท็อด สามารถทำได้โดยนำเอาคลาสที่อิมพลิเมนต์หรือสืบทอดมาจากคลาส Rule นำมาติดตั้งหรือวางไว้ในที่ๆกำหนดไว้และอาศัยวิธีการโหลดคลาสขณะรันไทม์เข้าไป ก็จะทำให้โปรแกรมแบบเอโอเอ็มสามารถเรียกใช้อ็อบเจกต์ของคลาสใหม่ได้ทันที

และในกรณีที่พฤติกรรมหรือเงื่อนไขของธุรกิจนั้นมีความซับซ้อน สามารถที่จะออกแบบให้หลายๆสตราทิจิมาประกอบกันให้ซับซ้อนตามที่ต้องการได้โดยการนำเอาดีไซน์แพทเทิร์นของคอมโพสิทมาประยุกต์ใช้

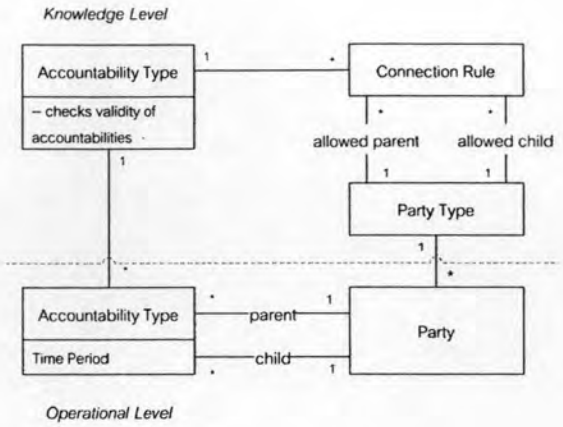


รูปที่ 2.4 ไทป์สแควร์และสตราทิจิ [4]

- แอ็คเคาน์ทะบิลิตี

แพทเทิร์นนี้ถูกนำมาใช้ในการออกแบบให้อ็อบเจกต์สามารถปรับเปลี่ยนความสัมพันธ์ระหว่างคลาสแต่ละประเภทได้ โดยอ็อบเจกต์ของคลาส Accountability นั้นจะเป็นตัวแทนความสัมพันธ์ระหว่างอ็อบเจกต์ของคลาส Party ซึ่งประเภทของความสัมพันธ์นั้นจะอธิบายได้ด้วยอ็อบเจกต์ของคลาส AccountabilityType

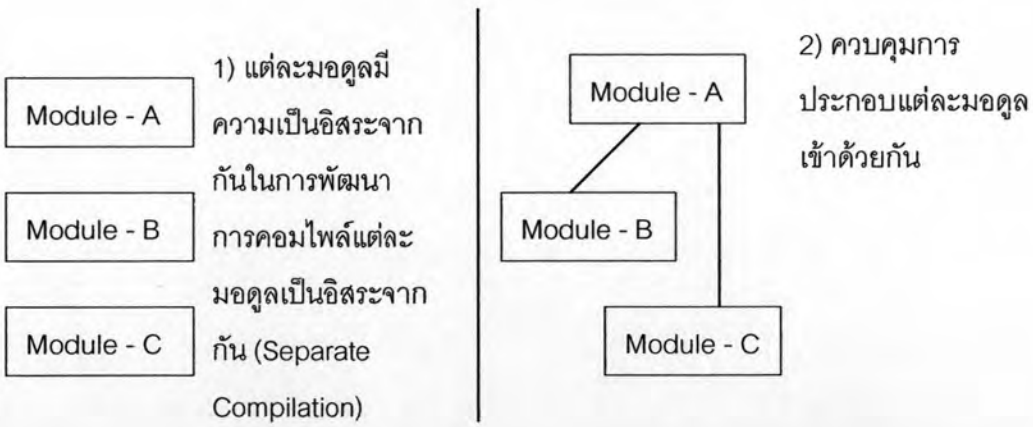
นอกจากนั้นยังมีคลาส ConnectionRule ซึ่งอ็อบเจกต์ของคลาสนี้จะเป็นตัวกำหนดว่าความสัมพันธ์ที่จะเกิดขึ้นระหว่างอ็อบเจกต์ของคลาส Party นั้นจะต้องเป็น Party ประเภทใดและอ็อบเจกต์ใดเป็นแม่หรือลูก ดังรูปที่ 2.5



รูปที่ 2.5 แอ็คเคาน์ทะบิลิตี [5]

2.1.3 ยูนิตโมเดล (Unit Model)

ในงานวิจัยของ Matthew Flatt [6] นั้นได้กล่าวถึงการพัฒนาซอฟต์แวร์ที่เป็นมอดูลหรือคอมโพเนนต์นั้นควรจะสามารถทำกระบวนการที่เรียกว่า assembly-line programming ได้ ซึ่งหมายถึง (1) มอดูลนั้นจะต้องสามารถแยกขั้นตอนของการคอมไพล์ออกมาเพื่อสนับสนุนให้การพัฒนาแต่ละมอดูลแยกเป็นอิสระจากกัน และ (2) ควรจะต้องมีภาษาสนับสนุนให้สามารถเชื่อมต่อ (Link) แต่ละมอดูลเข้าด้วยกันได้เพื่อให้นักพัฒนาสามารถทำการควบคุมการประกอบเข้าด้วยกันได้ดังรูปที่ 2.6



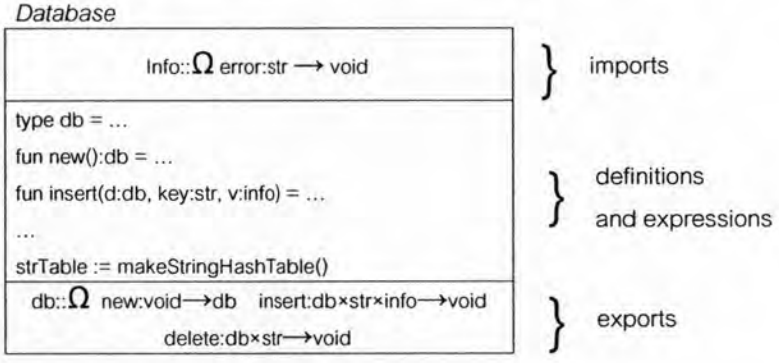
รูปที่ 2.6 กระบวนการหลักในการพัฒนาซอฟต์แวร์แบบมอดูล

Matthew Flatt ได้นำเสนอภาษาสำหรับการพัฒนาซอฟต์แวร์มอดูลที่เรียกว่า โปรแกรมยูนิต (Program Units) เพื่อสนับสนุนการทำ assembly-line programming ยูนิตนั้นเปรียบเสมือนหน่วยโปรแกรมย่อยที่เป็นตัวแทนของมอดูล ทำหน้าที่ในการอธิบายโครงสร้างของแต่ละมอดูล

ยูนิตแบ่งออกเป็นสองประเภทคือ อะตอมยูนิต (Atomic Unit) และ คอมพาวด์ยูนิต (Compound Unit) รูปที่ 2.7 แสดงตัวอย่างของอะตอมยูนิต ซึ่งประกอบด้วยสามส่วนคือ

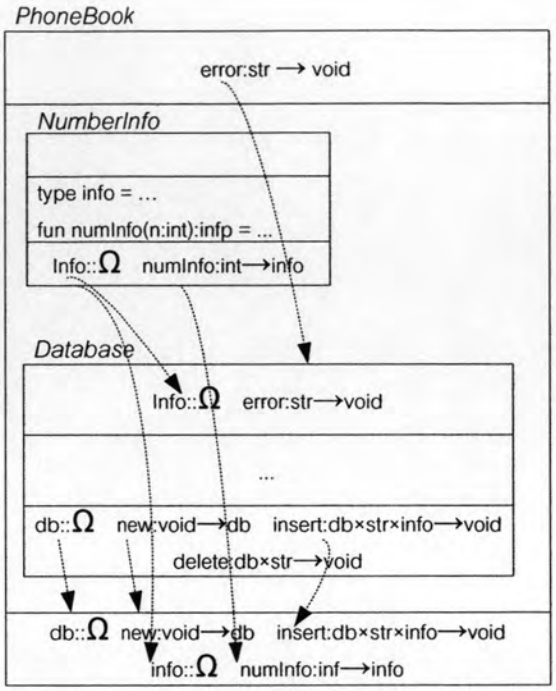
- เซตของอิมพอร์ต (Import) – เป็นส่วนที่อธิบายว่ายูนิตทำการอิมพอร์ตฟังก์ชัน หรือ ชนิดของข้อมูลใดเข้ามาใช้ภายในยูนิต ซึ่งฟังก์ชันที่อิมพอร์ตนั้นเป็นฟังก์ชันของยูนิตอื่น นั่นคือ ยูนิตมีการขึ้นต่อกัน (Dependency) กับยูนิตอื่น
- เซตของเอ็กซ์พอร์ต (Export) – เป็นส่วนที่อธิบายว่า ยูนิตทำการเอ็กซ์พอร์ตฟังก์ชันใดเพื่อให้ยูนิตอื่นสามารถนำไปใช้ได้บ้าง
- เซตของนิยาม (Definition) - เป็นส่วนที่อธิบายการกำหนดค่าเริ่มต้นให้กับตัวแปรหรือฟังก์ชันที่มีภายในยูนิต

ตัวอย่างยูนิตที่ชื่อ Database ในรูปที่ 2.7 กำหนดให้มีการอิมพอร์ต ประเภทข้อมูลที่ชื่อว่า info โดยมีสัญลักษณ์ Ω เป็นตัวบอกว่า Info คือประเภทข้อมูลชนิดหนึ่ง และฟังก์ชันที่จัดการเกี่ยวกับข้อผิดพลาดในการทำงานเข้ามาใช้ภายในยูนิต ส่วนของนิยามภายในยูนิต ทำการกำหนดค่าเริ่มต้นให้กับตัวแปรชื่อ db กำหนดฟังก์ชัน new, insert และค่าให้กับ strTable ด้วยค่าที่ได้จากฟังก์ชัน makeStringHashTable และส่วนสุดท้ายคือ ส่วนของเอ็กซ์พอร์ต ได้กำหนดให้ยูนิตอื่นสามารถนำประเภทข้อมูลที่ชื่อ db รวมทั้งฟังก์ชัน new, insert และ delete ไปใช้ได้



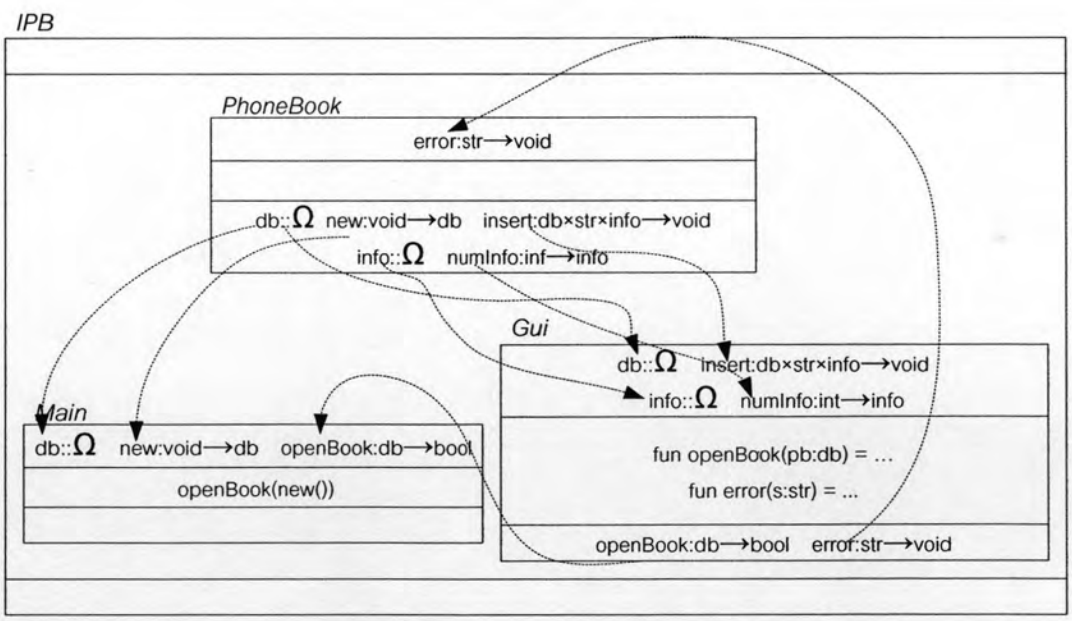
รูปที่ 2.7 ตัวอย่างของอะตอมยูนิต [6]

คอมพาวด์ยูนิต คือ ยูนิตที่ประกอบกันขึ้นจากอะตอมยูนิต ดังรูปที่ 2.8 แสดงตัวอย่าง คอมพาวด์ยูนิต ชื่อ PhoneBook ซึ่งเกิดจากการลิงค์ระหว่างสองยูนิต และทำการอิมพอร์ตฟังก์ชันชื่อ error ซึ่งทำหน้าที่จัดการเกี่ยวกับข้อผิดพลาดต่างๆเข้ามาในยูนิต ภายในยูนิต PhoneBook ประกอบด้วยยูนิต NumberInfo และ Database โดยที่ยูนิต PhoneBook ทำการซ่อนฟังก์ชัน delete ของ Database ไว้ และทำการเอ็กซ์พอร์ตฟังก์ชันของยูนิตอื่นออกมาอีกครั้ง



รูปที่ 2.8 ตัวอย่างของคอมพาวด์ยูนิต [6]

รูปที่ 2.9 แสดงตัวอย่างของการออกแบบโปรแกรมที่สมบูรณ์ ซึ่งประกอบขึ้นจากยูนิตต่างๆเข้าด้วยกัน ภายในประกอบด้วยยูนิตชื่อ PhoneBook, Main และ Gui



รูปที่ 2.9 ตัวอย่างโปรแกรมที่สมบูรณ์ [6]

Matthew Flatt กำหนดให้ยูนิตนั้นต้องมีคุณสมบัติที่สนับสนุนให้การพัฒนาแต่ละมอดูลเป็นอิสระจากกันได้ดังนี้ คือ

- การซ่อนรายละเอียด (Encapsulation) – ยูนิตจะซ่อนรายละเอียดของส่วนประกอบของโปรแกรมไว้ภายใน ยูนิตจะเชื่อมต่อกันได้โดยผ่านทางอินเตอร์เฟสกำหนดไว้เท่านั้น
- มีความเป็นอิสระในการคอมไพล์ (Separate compilation) - อินเตอร์เฟสของยูนิตจะให้ข้อมูลที่เพียงพอต่อการคอมไพล์ได้โดยอิสระ ซึ่งจะทำให้แต่ละยูนิตเป็นอิสระจากกัน

นอกจากนั้นเพื่อสนับสนุนกระบวนการประกอบแต่ละส่วนโปรแกรมเข้าด้วยกันได้ จะต้องมีกลไกดังนี้

- แต่ละยูนิตสามารถถูกนำกลับมาใช้ใหม่ หรือ ถูกแทนที่ได้ (Individual reuse and replace) – นั่นหมายความว่า การเชื่อมต่อกันระหว่างยูนิตนั้น จะต้องเชื่อมต่อกันที่ภายนอกยูนิต ไม่ใช่ภายในตัวยูนิตเอง นอกจากนั้น ภาษาจะต้องสนับสนุนให้ยูนิตสามารถมีได้หลายๆอินสแตนท์ในบริบทที่ต่างกัน
- สามารถทำการลิงค์หลายๆยูนิตเข้าด้วยกันได้ (Hierarchical structuring) เพื่อสร้างยูนิตที่มีขนาดใหญ่ขึ้น และสามารถเลือกที่จะซ่อนรายละเอียดบางอย่างไว้ได้
- สนับสนุนให้สามารถทำการเชื่อมต่อยูนิตได้ใหม่ (Dynamic linking) และสามารถเรียกใช้งานผ่านอินเตอร์เฟสที่กำหนดไว้อย่างชัดเจนได้

เปรียบเทียบคอมโพเนนต์กับภาษาจาวา - แพ็กเกจ (Package) หนึ่งแพ็กเกจในภาษาจาวานั้นถูกนำมาเทียบได้กับคอมโพเนนต์หนึ่ง แต่แพ็กเกจนั้นมีข้อจำกัดที่สำคัญก็คือ ไม่สามารถที่จะแยกการขึ้นต่อกันจากแพ็กเกจอื่นได้เลย คลาสในแพ็กเกจจะผูกติดกับแพ็กเกจที่เรียกใช้อย่างถาวร ไม่สามารถที่จะแยกออกจากกันได้ ด้วยเหตุนี้ จึงยากที่จะแทนที่แพ็กเกจเดิมด้วยแพ็กเกจใหม่ที่มีฟังก์ชันการทำงานเดียวกันได้อย่างเสมอเหมือน และการแยกทีมพัฒนาแต่ละคอมโพเนนต์อิสระจากกันอาจจะก่อให้เกิดแพ็กเกจที่ซ้อนทับด้วยชื่อเดียวกัน นอกจากนั้นยังไม่สามารถคอมไพล์และทดสอบแต่ละคอมโพเนนต์ได้โดยอิสระเพราะต้องนำมารวมกันจึงจะสามารถทำได้

ความแตกต่างระหว่างยูนิตโมเดลและอ็อบเจกต์โมเดล – ยูนิตโมเดล เป็นโมเดลที่ใช้ในการออกแบบคอมโพเนนต์ซอฟต์แวร์ ซึ่งอยู่ในระดับสูงกว่า (High Level) อ็อบเจกต์โมเดล การอิมพลีเมนท์ตามข้อกำหนดของยูนิตซึ่งจะได้คอมโพเนนต์ออกมา สามารถที่จะอิมพลีเมนท์ด้วยภาษาเชิงวัตถุซึ่งหมายความว่า ภายในคอมโพเนนต์อาจจะมีฟังก์ชันการทำงานที่ประกอบขึ้นมาจากหลายคลาสได้ ดังนั้น ยูนิตโมเดลนั้นจะสามารถเปรียบเทียบได้ที่ระดับแพ็กเกจ เช่น ในภาษาจาวา เป็นต้น สิ่งที่แตกต่างกันระหว่างแพ็กเกจและยูนิตโมเดล ก็คือ คุณสมบัติดังที่กล่าวไปแล้วของยูนิตโมเดล ซึ่งเปรียบเทียบได้ดังตารางที่ 1

ตารางที่ 1 เปรียบเทียบความแตกต่างระหว่างแพ็กเกจและยูนิตโมเดล

คุณสมบัติ	ยูนิตโมเดล	แพ็กเกจ
การซ่อนรายละเอียด	ยูนิตโมเดลสามารถที่จะกำหนดให้ซ่อนฟังก์ชันการทำงานได้ตามที่ต้องการ	ไม่สามารถทำได้ เปิดเผยทุกคลาสและเมท็อดที่เป็น public เนื่องจากไม่ได้มีข้อกำหนดให้กับแพ็กเกจเช่นเดียวกับยูนิตโมเดล
อิสระในการคอมไพล์	สามารถแยกคอมไฟล์แต่ละคอมโพเนนต์ให้เป็นอิสระจากกันได้	ไม่สามารถทำได้ ต้องอาศัยแพ็กเกจของซอฟต์แวร์ที่สมบูรณ์ในการคอมไพล์

2.1.4 ภาษาและเครื่องมือที่ใช้ในการพัฒนายูนิตโมเดล

Jiazzi[7] เป็นภาษาที่ใช้ในสร้างคอมโพเนนต์หรือยูนิตด้วยภาษาจาวา (Java) ซึ่งสอดคล้องกับแนวคิดที่ใช้ในการสร้างคอมโพเนนต์ โดยสนับสนุนให้แต่ละคอมโพเนนต์เป็นอิสระในการคอมไพล์ สามารถเชื่อมต่อยูนิตที่ภายนอก ทำให้สามารถแทนที่ยูนิตเดิมด้วยยูนิตใหม่ได้ สามารถประกอบยูนิตเข้าด้วยกันเป็นโครงสร้างที่ใหญ่ขึ้นได้

คอมโพเนนต์หรือยูนิตในภาษา Jiazzi นั้นเปรียบเสมือนที่บรรจุโค้ดที่คอมไพล์แล้วอยู่ภายในและมีการกำหนดอินเตอร์เฟสของการเชื่อมต่อไว้อย่างชัดเจน การสร้างยูนิตด้วยภาษา Jiazzi นั้นจะประกอบไปด้วย 3 ส่วน คือ

1. แพ็กเกจซิกเนเจอร์ (Package Signature) เปรียบเสมือนส่วนที่เป็นอินเตอร์เฟสให้กับยูนิต ภายในแพ็กเกจซิกเนเจอร์จะประกอบด้วย คลาส หรือ อินเตอร์เฟส ของภาษาจาวา ยูนิตที่อิมพอร์ตแพ็กเกจซิกเนเจอร์จะสามารถอ้างถึงคลาสที่ประกาศไว้ในแพ็กเกจซิกเนเจอร์ได้
2. อะตอมยูนิต (Atom Unit) คือ หน่วยที่เล็กที่สุดของยูนิต อะตอมยูนิตจะเชื่อมต่อกับยูนิตอื่นๆ ได้โดยการอิมพอร์ตหรือเอ็กซ์พอร์ตแพ็กเกจซิกเนเจอร์
3. คอมพาวด์ยูนิต (Compound Unit) คือ ยูนิตที่ประกอบกันขึ้นมาจากยูนิตอื่นๆ ไม่ว่าจะเป็นอะตอมหรือคอมพาวด์ยูนิตเอง และเชื่อมต่อกับยูนิตอื่นๆด้วยวิธีการเดียวกับอะตอมยูนิต

2.1.5 การวัดซอฟต์แวร์ (Software Measurement) [8]

การวัดเป็นพื้นฐานสำคัญสำหรับทุกศาสตร์ทางวิศวกรรม การวัดเป็นกระบวนการกำหนดค่าตัวเลขหรือสัญลักษณ์ ให้กับคุณลักษณะ (Attributes) ของเอนทิตี (Entities) หรือสิ่งที่อยู่ในโลกของความเป็นจริงที่เราสนใจ เพื่อบรรยายเอนทิตีนั้นให้อยู่ในรูปของกฎที่นิยามไว้ชัดเจน

การวัดในทางวิศวกรรมซอฟต์แวร์ (Software Engineering) สามารถแบ่งได้เป็น 2 ประเภท ได้แก่

- **การวัดทางตรง** (Direct measurement) เป็นการวัดเฉพาะคุณลักษณะของสิ่งที่เราสนใจโดยไม่นำคุณลักษณะ หรือ เอนทิตีอื่นมาเกี่ยวข้อง ผลที่ได้จากการวัดทางตรงทำให้ทราบลักษณะในด้านโครงสร้างของซอฟต์แวร์ เช่น การวัดความยาวของซอร์สโค้ด สามารถวัดได้จากการนับจำนวนบรรทัดทั้งหมดของโปรแกรม เป็นต้น
- **การวัดทางอ้อม** (Indirect measurement) เป็นการวัดโดยนำเอาการวัดทางตรงมาประกอบการวัดคุณลักษณะบางอย่าง ซึ่งไม่สามารถวัดได้โดยตรง ตัวอย่างคุณลักษณะของวัดทางอ้อมคือ คุณภาพของซอฟต์แวร์ในด้านต่างๆ เช่น ความสามารถในการบำรุงรักษา (Maintainability) ความสามารถในการทำความเข้าใจระบบ (Understandability) เป็นต้น การที่จะวัดเพื่อทราบถึงคุณภาพของซอฟต์แวร์ในด้านใดด้านหนึ่งนั้น จะต้องกำหนดเกณฑ์ (Criteria) หรือปัจจัย (Factor) ในการวัด แต่ละเกณฑ์จะกำหนดมาตรวัดที่เกี่ยวข้องซึ่งเป็นได้ทั้งทางตรงและทางอ้อม

ความสามารถในการบำรุงรักษา เป็นหนึ่งในคุณภาพของซอฟต์แวร์ที่สำคัญ ซึ่งมีคุณลักษณะย่อย คือ ความสามารถในการทำความเข้าใจ โดยสามารถวัดได้จากค่าความซับซ้อนของโครงสร้าง (Structural Complexity) แผนภาพคลาส ซึ่งมาตรวัดที่ใช้สำหรับแผนภาพคลาส [11] มีดังนี้

- Number of Classes (NC) หมายถึง จำนวนคลาสทั้งหมดของแผนภาพคลาส
- Number of Attributes (NA) หมายถึง จำนวนแอตทริบิวต์ทั้งหมดของแผนภาพคลาส
- Number of Methods (NM) หมายถึง จำนวนเมท็อดทั้งหมดของแผนภาพคลาส
- Number of Associations (NAssoc) หมายถึง จำนวนของเส้นแอสโซซิเอชั่นในแผนภาพคลาสทั้งหมด
- Number of Aggregation (NAgg) หมายถึง จำนวนความสัมพันธ์แบบแอ็กกรีเกชันภายในแผนภาพคลาส (จำนวนของคู่ความสัมพันธ์แบบ whole-part)
- Number of Dependencies (NDep) หมายถึง จำนวนความสัมพันธ์ที่เป็นแบบการขึ้นต่อกัน
- Number of Generalizations (NGen) หมายถึง จำนวนรวมทั้งหมดของความสัมพันธ์แบบการสืบทอดของแผนภาพคลาส (จำนวนคู่ความสัมพันธ์ระหว่างคลาสแม่และคลาสลูก)
- Number of Generalisations Hierarchies (NGenH) หมายถึง จำนวนโครงสร้างแบบการสืบทอด ของแผนภาพคลาส
- Maximum DIT หมายถึง จำนวนระดับชั้นความลึกของโครงสร้างแบบการสืบทอดที่ค่ามากที่สุด
- Maximum HAgg (MaxHAgg). หมายถึง จำนวนความยาวของความสัมพันธ์แบบแอ็กกรีเกชันแบบต่อเนื่องที่ยาวที่สุดของคลาส

โดยที่ผลของการวัดที่มีค่ามาก แสดงว่าแผนภาพคลาสมีความซับซ้อนสูงกว่าตัวเลขที่มีค่าน้อย

นอกจากมาตรวัดสำหรับแผนภาพคลาสแล้ว มาตรวัดสำหรับแพ็กเกจก็เป็นอีกมาตรวัดที่ใช้การวัดค่าความซับซ้อนของโครงสร้าง [9] ซึ่งประกอบด้วยมาตรวัดดังนี้

- NumCls หมายถึง จำนวนคลาสที่อยู่ภายในแพ็กเกจ
- NumCls_tc หมายถึง จำนวนของคลาสที่อยู่ในภายในแพ็กเกจและรวมถึงแพ็กเกจที่ซ้อนอยู่ภายในด้วย
- NumOpsCls หมายถึง จำนวนฟังก์ชันของทุกคลาสที่อยู่ภายในแพ็กเกจ
- NumInterf หมายถึง จำนวนอินเตอร์เฟสที่อยู่ภายในแพ็กเกจ
- R หมายถึง จำนวนความสัมพันธ์ระหว่างคลาสและอินเตอร์เฟสที่อยู่ภายในแพ็กเกจ
- Ca หมายถึง จำนวนของคลาสหรืออินเตอร์เฟสที่อยู่นอกแพ็กเกจ ขึ้นต่อกันกับคลาสหรืออินเตอร์เฟสที่อยู่ภายในอีกแพ็กเกจที่สนใจ
- Ce หมายถึง จำนวนของคลาสหรืออินเตอร์เฟสที่อยู่นอกแพ็กเกจ ซึ่งคลาสหรืออินเตอร์เฟสที่อยู่ภายในแพ็กเกจหนึ่งไปขึ้นต่อกัน
- DepPack หมายถึง จำนวนแพ็กเกจซึ่งคลาสหรืออินเตอร์เฟสที่อยู่ในแพ็กเกจไปขึ้นต่อกันกับแพ็กเกจอื่น

2.2 งานวิจัยที่เกี่ยวข้อง

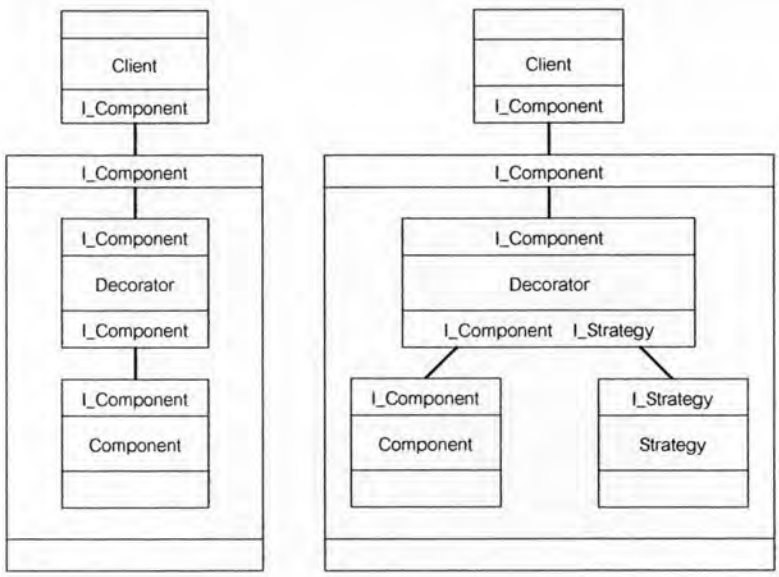
2.2.1 งานวิจัย "Static and Dynamic Structure in Design Patterns" [10]

งานวิจัยของ Eric Eide และคณะได้ศึกษาเกี่ยวกับดีไซน์แพทเทิร์นพบว่า ดีไซน์แพทเทิร์นนั้น มีข้อเสียหลายประการที่ทำให้การพัฒนาซอฟต์แวร์นั้นไม่เป็นไปตามที่ผู้ออกแบบได้กำหนดไว้ คือ ในขั้นตอนของการออกแบบ (Design Time) โมเดลจะถูกนำเสนอในรูปแบบของคลาสและความสัมพันธ์ระหว่างคลาส แต่ในขณะรันไทม์ (Run Time) การทำงานของโปรแกรมจะขึ้นกับโค้ดที่อิมพลีเมนท์อยู่ภายใน วิธีการนี้ยืดหยุ่นในการทำงานแต่บางครั้งนั้นทำให้คุณสมบัติที่ระบบควรจะต้องมีไม่ได้ถูกตรวจสอบ

ตัวอย่างเช่น เดคคอร์เรเตอร์แพทเทิร์น (Decorator Pattern) ถูกออกแบบเพื่อให้ผู้ออกแบบสามารถเพิ่มหน้าที่ความรับผิดชอบ (Responsibility) ให้กับอ็อบเจกต์ใดๆ เช่น คลาส A มีฟังก์ชันการทำงานที่ไม่ได้ถูกออกแบบให้รองรับการเข้าถึงข้อมูลพร้อมกันหลายเทรดได้อย่างปลอดภัย (Non thread safe) การแก้ปัญหาสามารถทำได้โดยการสร้างคลาสเดคคอร์เรเตอร์ เพื่อห่อหุ้มคลาสเดิมไว้ โดยคลาส เดคคอร์เรเตอร์จะทำหน้าที่ในการควบคุมให้การเข้าถึงข้อมูลที่ใช้งานร่วมกันหลายเทรดได้อย่างปลอดภัย (Thread safe) ซึ่งพบว่าดีไซน์แพทเทิร์นนี้ ได้ซ่อนข้อมูลที่สำคัญบางอย่างไว้คือ อินสแตนท์ของแต่ละคลาสจะต้องมีเพียงหนึ่งอินสแตนท์เท่านั้น นอกจากนั้นยังไม่ได้กำหนดเงื่อนไขใน

การเรียกใช้ไว้อย่างชัดเจนว่าคลาสแม่จะต้องถูกเรียกใช้โดยผ่านทางคลาสที่เป็นเดคคอร์เรเตอร์เท่านั้น กล่าวในอีกทางหนึ่งก็คือ ผู้ที่เรียกใช้จะต้องไม่ไปเรียกใช้บริการที่คลาสแม่ จะต้องเรียกผ่านทางคลาสลูกเท่านั้นจึงจะทำผลการทำงานของโปรแกรมถูกต้อง

แนวทางในการปรับปรุงของ Eric Eide คือ การออกแบบโดยแยกเอาข้อมูลความรู้มีอยู่ในดีไซน์แพทเทิร์น (Static Knowledge) ออกมาจากซอร์สโค้ด เพื่อนำมาเป็นข้อกำหนดไว้ที่ระดับของยูนิตและการเชื่อมต่อกันระหว่างยูนิต เนื่องจากยูนิตโมเดลนั้นอนุญาตให้ผู้ออกแบบระบบสามารถกำหนดคุณสมบัติหรือเงื่อนไข (Design Constraints) ที่ระบบจะต้องมีได้ตั้งแต่ในช่วงของการออกแบบสถาปัตยกรรมของระบบ



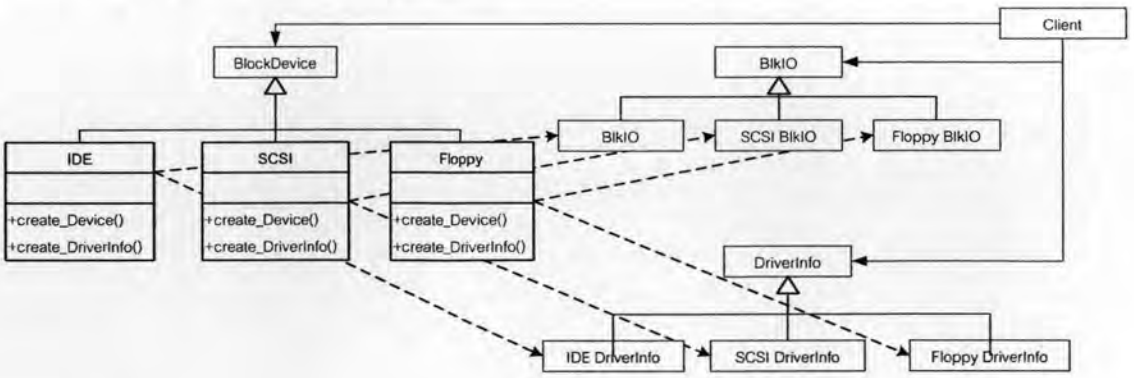
รูปที่ 2.10 ดีไซน์แพทเทิร์นในรูปของยูนิตโมเดล [10]

จากรูปที่ 2.10 ทางด้านซ้ายมือแสดงเดคคอร์เรเตอร์แพทเทิร์นด้วยยูนิตโมเดล ซึ่งยูนิตโมเดลจะซ่อนคุณสมบัติไว้ภายในโมเดล สามารถกำหนดได้ว่าให้มีเพียงแค่อินสแตนท์เดียวเท่านั้นในการทำงานของโปรแกรม การเรียกใช้บริการจะต้องเรียกใช้ผ่านทางอินเตอร์เฟสที่เอ็กซ์พอร์ตเท่านั้น จึงทำให้ผู้ที่เรียกใช้ไม่สามารถเรียกใช้บริการไปยังคลาสแม่ได้ และสามารถกำหนดให้ยูนิตอิมพอร์ตยูนิตที่ทำงานเป็นแบบ non-thread safe ไว้ภายในและเปิดเผยเฉพาะอินเตอร์เฟสให้ยูนิตอื่นเรียกใช้ผ่านทางอินเตอร์เฟสที่เป็นแบบ thread safe ได้เท่านั้น นอกจากนั้นยังคงคุณสมบัติของการนำกลับมาใช้ใหม่ได้โดยการนำยูนิตมาเชื่อมต่อกันโดยที่ไม่ส่งผลกระทบกับการเรียกใช้จากภายนอก ดังรูปที่ 2.10 ทางด้านขวามือ

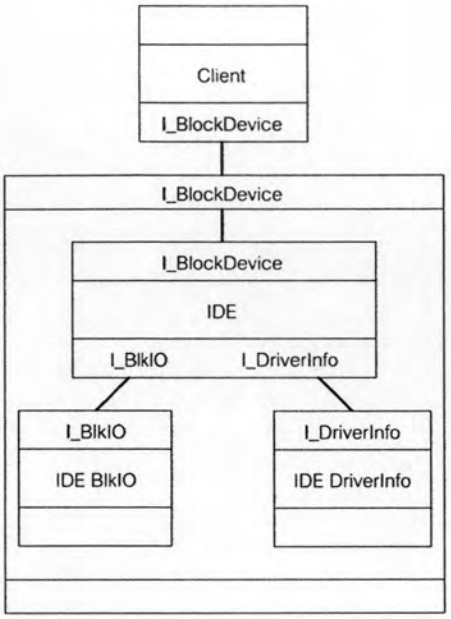
ในขั้นตอนการนำเสนอดีไซน์แพทเทิร์นด้วยยูนิตโมเดลนั้น Eric Eide ได้แนะนำขั้นตอนการแปลงให้อยู่ในรูปของยูนิตโมเดล ดังนี้คือ

1. ระบุคลาสที่เป็น abstract และ interface เพื่อทำการกำหนดอินเตอร์เฟสร่วมกันระหว่างคลาส เนื่องจากในดีไซน์แพทเทิร์นนั้นมักจะมีคลาสที่เป็น abstract หรือ interface ทำหน้าที่เป็นตัวกำหนดเมท็อดที่ให้บริการแก่คลาสอื่นๆที่เรียกใช้
2. หาส่วนประกอบที่เป็นแบบ static participants และ dynamic participants ของดีไซน์แพทเทิร์น เพื่อแยกส่วนประกอบที่ให้ข้อมูลที่สำคัญออกมา เช่น เดคคอร์เรเตอร์แพทเทิร์นภายในจะประกอบไปด้วยอินสแตนท์ของแต่ละคลาสเพียงอินสแตนท์เดียว หรือแพทเทิร์น Abstract Factory ก็มักจะมีอินสแตนท์ของ Concrete Factory เพียงหนึ่งอินสแตนท์ของแต่ละคลาส ซึ่งคลาสเหล่านี้ถือเป็นส่วนประกอบที่เป็น static participants คือทราบจำนวนที่แน่นอนตั้งแต่การออกแบบ ส่วนประกอบที่ไม่ทราบช่วงการออกแบบก็เป็น dynamic participants
3. กำหนดอินเตอร์เฟสให้กับส่วนประกอบที่เป็น static participants ซึ่งส่วนประกอบนี้สามารถกำหนดให้เป็นฟังก์ชันที่ไม่จำเป็นต้องอยู่ในรูปของอ็อบเจกต์ (จากเหตุผลในข้อ 2) เช่น อยู่ในรูปสแตติกเมท็อด ในบางแพทเทิร์น static participants ผู้ออกแบบอาจเลือกที่จะรวมให้เป็นอินเตอร์เฟสของคลาสต่างๆเป็นเพียงอินเตอร์เฟสเดียว หรือ แยกกันเป็นหลายๆอินเตอร์เฟสของแต่ละคลาสก็ได้
4. กำหนดอินเตอร์เฟสให้กับส่วนประกอบที่เป็น dynamic participants ซึ่งส่วนนี้จะต้องกำหนดให้อยู่ในรูปแบบของอ็อบเจกต์เนื่องจากไม่ทราบจำนวนอินสแตนท์ที่แน่นอนในช่วงของการออกแบบ ดังนั้นอินเตอร์เฟสของยูนิตจะต้องกำหนดในรูปแบบของคลาส
5. เขียนข้อกำหนดของยูนิตให้กับ static และ dynamic participants
6. ภายในคอมพาวด์ยูนิตให้สร้างอินสแตนท์ของยูนิตที่นำมาประกอบกันและเชื่อมต่อกันระหว่างยูนิต

รูปที่ 2.11 แสดงตัวอย่างของแผนภาพคลาสที่ออกแบบโดยใช้ดีไซน์แพทเทิร์นและผลที่ได้จากการแปลงให้อยู่ในรูปของยูนิตโมเดลในรูปที่ 2.12



รูปที่ 2.11 แผนภาพคลาสของ Managing Block Device ด้วยแฟคทอรีแพทเทิร์น [10]



รูปที่ 2.12 ผลที่ได้จากการแปลงแผนภาพคลาสเป็นยูนิตโมเดล [10]

ผลสรุปที่ได้จากการวิจัยของ Eric Eide นั้นพบว่า

- เครื่องมือตรวจสอบความสอดคล้องกับเงื่อนไขนั้นสามารถตรวจพบข้อผิดพลาดที่ High-level ได้
- ข้อบังคับสามารถถูกกำหนดให้เป็นเงื่อนไขการออกแบบสำหรับโดเมนใดๆได้
- สามารถแยกข้อผิดพลาดที่เกิดจากการออกแบบ ออกจากข้อผิดพลาดของการอิมพลีเมนท์ได้
- เครื่องมือที่ใช้ในการตรวจสอบเงื่อนไขไม่ขึ้นกับภาษาที่ใช้ในการอิมพลีเมนท์
- ช่วยปรับปรุงประสิทธิภาพในการทำงานของโปรแกรม
- ช่วยให้ง่ายต่อการทำความเข้าใจ และสามารถนำกลับมาใช้ใหม่ได้

ผู้จัดทำวิทยานิพนธ์ได้นำแนวคิดในการแก้ปัญหาของดีไซน์แพทเทิร์นด้วยยูนิตโมเดลมาประยุกต์ใช้