แนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุ

Object-Oriented Software Development Concept

ผู้ช่วยศาสตราจารย์ ดร. รัฐสิทธิ์ สุขะหุต

คำนำ

แนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุเป็นการมองปัญหาต่างๆในโลกของความเป็นจริง เพื่อวิเคราะห์หาแนวทางการแก้ไขปัญหา โดยมองปัญหาออกเป็นแบบจำลองควบคู่ไปกับการวิเคราะห์ เพื่อมุ่งเน้นไปสู่การพัฒนาซอฟต์แวร์ที่มีประสิทธิภาพ โดยการพัฒนาซอฟต์แวร์เชิงวัตถุนี้ปัจจุบันมีภาษาต่างๆ หลายภาษาที่รองรับและสนับสนุน ทั้งในเรื่องของเทคโนโลยี เครื่องมือสำหรับพัฒนา รวมถึงกลุ่มคอมโพเนนท์และแพจเกจ แนวคิดเชิงวัตถุเป็นแนวคิดที่มุ่งเน้นการแก้ไขปัญหาโดยอาศัยการทำโมเดลเพื่อให้ภาพมุมมองของปัญหาสอดคล้องกับโดเมนของธุรกิจ โมเดลเชิงวัตถุช่วยทำให้การวิเคราะห์ ออกแบบและการพัฒนาซอฟต์แวร์เป็นไปอย่างมีระบบและมีประสิทธิภาพ ซึ่งการพัฒนาระบบจะสามารถใช้ประโยชน์จากแนวคิดของการนำโปรแกรมโค๊ดกลับมาใช้ใหม่ โดยนำเอาคอมโพเนนท์หรือแพจเกจมาใช้พัฒนาในการพัฒนาซอฟต์แวร์ใหม่ หรือในบางครั้งเป็นลักษณะของการพัฒนาต่อยอด เพื่อให้ระบบที่มีอยู่เดิมมีความสมบูรณ์มากยิ่งขึ้น ซึ่งแนวคิดนี้เป็นแนวคิดที่สนับสนุนการใช้ประโยชน์จากกลุ่มออบเจ็กต์ที่มีอยู่แล้วกลับมาใช้ใหม่ทำให้การพัฒนาซอฟต์แวร์สามารถทำได้อย่างรวดเร็ว มีความเสถียรภาพสูง และการบำรุงดูแลรักษาระบบหลังการพัฒนาเสร็จสิ้นก็สามารถทำได้ง่าย นอกจากนี้จากแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุยังช่วยให้ซอฟต์แวร์ถูกออกแบบเป็นโครงสร้าง ทั้งในระดับของออบเจ็กต์ ไลบรารีหรือคอมโพเนนท์ ทำให้ทีมพัฒนาซอฟต์แวร์สามารถทำงานได้อย่างเป็นระบบ

ในการจัดทำตำราเรื่องแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุ ผู้จัดทำมีวัตถุประสงค์เพื่อที่จะอธิบายถึงหลักการและแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุ โดยไม่ได้มุ่งเน้นที่ภาษาใดภาษาหนึ่งโดยเฉพาะ แต่ต้องการให้ผู้อ่านเห็นถึงแนวคิดในการพัฒนาโปรแกรมในเชิงวัตถุในรูปแบบภาษาต่างๆ ซึ่งปัจจุบันการพัฒนาซอฟต์แวร์นั้นจะต้องคำนึงถึงประสิทธิภาพและความสามารถในการทำงานและโดยผู้พัฒนาจะต้องทำขั้นตอนการวิเคราะห์ออกแบบและพัฒนาระบบ โดยสามารถควบคุมคุณภาพของซอฟต์แวร์เป็นไปตามความต้องการของผู้ใช้งาน โดยอาศัยแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุเพื่อให้เกิดงานซอฟต์แวร์ที่มีประสิทธิภาพ ผู้จัดทำหวังเป็นอย่างยิ่งว่า ตำราเล่มนี้จะเป็นประโยชน์ต่อผู้ที่สนใจในการศึกษาแนวคิดในการพัฒนาซอฟต์แวร์ตามหลักการเชิงวัตถุ เพื่อเป็นพื้นฐานในการศึกษาหาความรู้เพิ่มเติม และขอขอบคุณผู้อ่านทุกท่านที่ให้ความสนใจตำราเล่มนี้

ผู้ช่วยศาสตราจารย์ ดร. รัฐสิทธิ์ สุขะหุต
ภาควิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ มหาวิทยาลัยเชียงใหม่

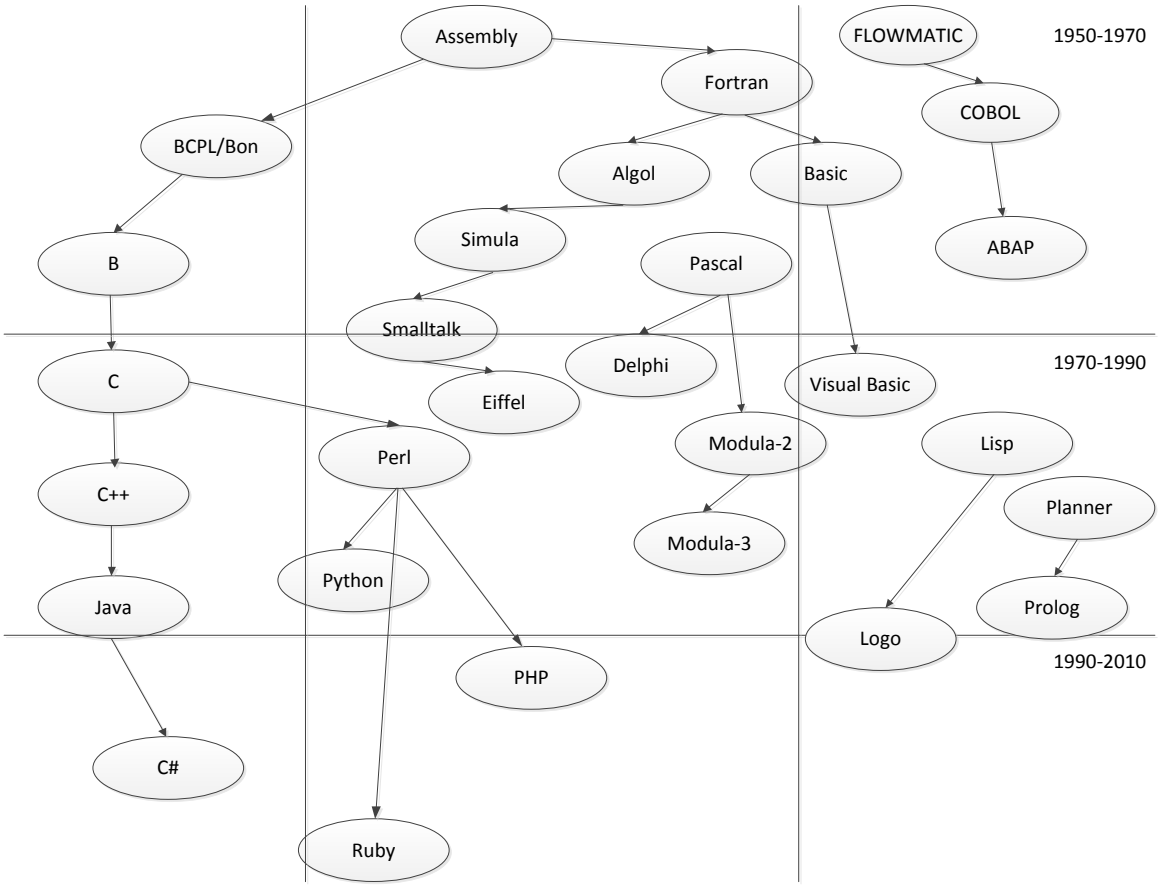
สารบัญ

สารบัญรูป

บทที่ 1 - วิวัฒนาการของโปรแกรมภาษา

ก่อนที่ซอฟต์แวร์จะถูกพัฒนาออกมาอยู่ในรูปของผลิตภัณฑ์ที่สมบูรณ์ โปรแกรมจะต้องผ่านขั้นตอนการวิเคราะห์ออกแบบและพัฒนาหลายขั้นตอนการควบคุมคุณภาพของการพัฒนาในแต่ละขั้นตอนจึงเป็นสิ่งสำคัญ ในระยะ 30 กว่าปีที่ผ่านมาขนาดของโปรแกรมที่ถูกพัฒนามีขนาดใหญ่ขึ้นและมีความซับซ้อนมากยิ่งขึ้น การพัฒนาโปรแกรมจึงต้องมีหลักโครงสร้างการออกแบบและการพัฒนาโปรแกรมที่ชัดเจน และนอกเหนือไปจากการควบคุมคุณภาพของโปรแกรมแล้ว การควบคุมเวลาและต้นทุนของการผลิตก็เป็นสิ่งสำคัญ แนวคิดของการพัฒนาโปรแกรมแบบเชิงวัตถุจัดได้ว่าเป็นแนวทางใหม่ ซึ่งแนวคิดนี้ต่างจากการพัฒนาซอฟต์แวร์แบบโครงสร้าง เช่นภาษา C, Pascal, Fortran, Cobol หรือภาษา Basic อย่างไรก็ตามแนวคิดของออบเจ็กต์ก็ไม่ได้ถือว่าเป็นเรื่องใหม่ทั้งหมด เนื่องจากว่าหลักการบางอย่างของออบเจ็กต์ได้ถูกนำมาใช้กับการพัฒนาซอฟต์แวร์แบบโครงสร้างแล้วบ้าง ตัวอย่างที่เห็นได้ชัดคือการแบ่งโปรแกรมออกเป็นโปรแกรมย่อยหรือเรียกว่าโพซีเยอร์และฟังก์ชัน และสำหรับโปรแกรมที่มีขนาดใหญ่ขึ้นก็อาจมีการนำเอาโพซีเยอร์ต่างๆมาจัดรวมกันเป็นลักษณะของโมดูลหรือไลบรารี ซึ่งทำให้โปรแกรมเมอร์สามารถพัฒนาซอฟต์แวร์ได้รวดเร็ว เป็นระเบียบ และมีคุณภาพมากยิ่งขึ้น อย่างไรก็ตามแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุไม่ได้ถือว่าเป็นเรื่องใหม่ทั้งหมดเลยทีเดียว เนื่องจากได้มีการนำเสนอแนวคิดที่ใช้คุณสมบัติของออบเจ็กต์มาตั้งแต่ในอดีตเช่น การนำโปรแกรมโค๊ดเก่ากลับมาใช้ การจัดแบ่งโปรแกรมโค๊ดออกเป็นส่วนต่างๆเรียกว่า ไลบรารีหรือโมดูล หรือแบ่งโปรแกรมออกเป็นไฟล์ย่อยต่างๆ เพื่อให้การพัฒนาซอฟต์แวร์เป็นไปอย่างมีประสิทธิภาพสามารถที่จะแก้ไขปัญหาหรือหาข้อผิดพลาดได้ง่าย โดยเฉพาะโปรแกรมที่มีขนาดใหญ่และมีการพัฒนาร่วมกันหลายคน

จากรูป 1.1 จะเห็นว่าเรามีการพัฒนาโปรแกรมภาษาคอมพิวเตอร์มาตั้งแต่ปี 1950 ที่มีความพยายามที่จะควบคุมการทำงานของเครื่องคอมพิวเตอร์โดยตรง หรือพยายามที่จะสั่งให้เครื่องทำงานโดยภาษาเครื่อง (Machine Language) แต่ก็เกิดความยุ่งยากและซับซ้อน จนมาถึงในช่วงของภาษา Assembly ซึ่งเป็นภาษาที่มีการทำงานในลักษณะของชุดคำสั่งที่มนุษย์ไม่จำเป็นต้องเรียนรู้ภาษาเครื่อง แต่ก็ยังเป็นลักษณะของชุดคำสั่งที่ไม่มีโครงสร้างของภาษาชัดเจน แต่ก็นับได้ว่าเป็นจุดเริ่มต้นของการควบคุมการทำงานของเครื่องคอมพิวเตอร์โดยอาศัยโปรแกรมภาษา ซึ่งเราจะเห็นได้ว่านับจากนั้นเป็นต้นมาก็ได้มีการพัฒนาโปรแกรมภาษาต่างๆออกมาอีกมากมาย ซึ่งมีความแตกต่างกันไปทั้งในเรื่องของโครงสร้างภาษา และความสามารถที่ภาษานั้นๆสามารถทำได้



รูป 1.1 วิวัฒนาการของโปรแกรมภาษาคอมพิวเตอร์

1.1 โปรแกรมภาษาคอมพิวเตอร์ (Programming Languages)

ในทางคอมพิวเตอร์ เนื่องจากคอมพิวเตอร์เป็นอุปกรณ์อิเล็กทรอนิกส์ มีเฉพาะวงจรการเปิดและปิด ทำให้เครื่องคอมพิวเตอร์สื่อสารโดยใช้เลขฐานสองเท่านั้น เรียกภาษาที่ใช้เฉพาะเลขฐานสองนี้ว่า ภาษาเครื่อง (Machine Language) ทำให้การใช้งานยุ่งยาก จึงมีการพัฒนาภาษาคอมพิวเตอร์ขึ้นมาเพื่อใช้สื่อสารระหว่างเครื่องคอมพิวเตอร์กับมนุษย์ได้ โดยชนิดของภาษาคอมพิวเตอร์ได้ถูกจำแนกตามระดับภาษาดังนี้

- ภาษาเครื่อง (Machine Language)
- ภาษาแอสเซมบลี (Assembly Language)
- ภาษาระดับสูง (High-Level Language)
- ภาษาระดับสูงมาก (Very High-Level Language)
- ภาษาธรรมชาติ (Natural Language)

ภาษาเครื่อง (Machine Language)

เป็นภาษาระดับต่ำที่สุด เพราะใช้เลขฐานสองแทนข้อมูล และคำสั่งต่างๆ จะเป็นภาษาที่ขึ้นอยู่กับชนิดของเครื่องคอมพิวเตอร์ หรือหน่วยประมวลผลที่ใช้ ดังนั้นนักพัฒนาซอฟต์แวร์จะต้องรู้จักวิธีที่จะรวมตัวเลขเพื่อแทนคำสั่งต่างๆ ทำให้การพัฒนาซอฟต์แวร์มีความซับซ้อนมาก คำสั่งในภาษาเครื่อง จะประกอบด้วย 2 ส่วน คือ

1) Operation Code เป็นคำสั่งที่สั่งให้คอมพิวเตอร์ปฏิบัติการ เช่น การบวก การลบ
2) Operands เป็นตัวที่ระบุตำแหน่งที่เก็บของข้อมูลที่จะนำเข้าสู่คอมพิวเตอร์เพื่อนำไปปฏิบัติตามคำสั่งใน Operation Code

ภาษาแอสเซมบลี (Assembly Language)

เมื่อในปี ค.ศ. 1952 ได้มีการพัฒนาภาษาระดับต่ำตัวใหม่ โดยภาษาแอสเซมบลีใช้รหัสแทนคำสั่งภาษาเครื่อง ทำให้นักพัฒนาซอฟต์แวร์สามารถพัฒนาซอฟต์แวร์ได้ง่ายขึ้น ซึ่งสัญลักษณ์ที่ใช้จะเป็นคำสั่งสั้นๆ ที่จำได้ง่าย เรียกว่า mnemonic code ตัวอย่างเช่น

A => ADD
C => Compare
MP => Multiply
STO => Store

นอกจากนี้ภาษาแอสเซมบลี ยังอนุญาตให้ผู้เขียนใช้ตัวแปรที่ตั้งขึ้นมาเองในการเก็บค่าข้อมูลใด ๆ เช่น X, Y, TOTAL แทนการอ้างถึงตำแหน่งที่เก็บข้อมูลจริง ในการพัฒนาโปรแกรมแอสแซมบลีผู้พัฒนาซอฟต์แวร์ ต้องอาศัยโปรแกรมแปลภาษาเรียกว่า แอสเซมเบลอ (Assembler) เพื่อทำการแปลภาษาแอสเซมบลีให้เป็นภาษาเครื่อง เพื่อให้คอมพิวเตอร์ทำงานตามต้องการ ซึ่งภาษาแอสเซมบลี 1 คำสั่งสามารถแปลเป็นภาษาเครื่องได้ 1 คำสั่งเช่นกัน ข้อจำกัดของภาษาแอสเซมบลี คือ ลักษณะการเขียนจะแตกต่างกันไปในแต่ละเครื่องเช่นเดียวกับภาษาเครื่อง นอกจากนี้ผู้พัฒนาซอฟต์แวร์จะต้องมีความรู้ในเรื่องฮาร์ดแวร์ เนื่องจากจะต้องยุ่งเกี่ยวกับการใช้งานหน่วยความจำที่อยู่ในเครื่องตลอดดังนั้น จึงเหมาะกับงานที่ต้องการความเร็วสูงหรืองานพัฒนาที่ต้องทำงานร่วมกับอุปกรณ์ฮาร์ดแวร์

ภาษาระดับสูง (High-Level Language)

ในปี ค.ศ. 1960 ได้มีการพัฒนาภาษาระดับสูงขึ้น โดยจะใช้โค๊ดภาษาอังกฤษ (English Code) แทนคำสั่งต่างๆ เช่น คำสั่ง Begin .. End หรือ if..then..else รวมทั้งใช้การนิพจน์ทางคณิตศาสตร์ และการจัดกลุ่มโปรแกรมให้เป็นลักษณะของบล็อกตามกลุ่มคำสั่ง เรียกว่าฟังก์ชันและโพรซีเยอร์ ทำให้นักพัฒนาซอฟต์แวร์ไม่จำเป็นต้องเข้าใจการทำงานของฮาร์ดแวร์มากนัก แต่สำหรับการพัฒนาโปรแกรมระดับสูงนี้ยังต้องอาศัยตัวแปลภาษาเรียกว่าคอมไพเลอร์ (Compiler) เพื่อทำการแปลจากภาษาโค๊ดที่เขียน (Source Code) ให้เป็นภาษาที่เครื่องเข้าใจ (Machine Language) ในการแปลภาษานี้ คอมไพเลอร์จะทำการตรวจสอบไวยากรณ์ของภาษาของ Source Program หรือ Source Module แต่ถ้าผ่านการแปลเรียบร้อย

และไม่มีข้อผิดพลาดจะเรียกว่า Object Program หรือ Object Module ซึ่งยังไม่สามารถทำงานได้ ต้องทำการ link หรือรวมเข้ากับ Library ของระบบก่อน จึงจะทำให้โปรแกรมทำงานได้หรือในภาษาเครื่องเรียกว่า Execute Program หรือ Load Module โดยทั่วไปจะมีนามสกุลเป็น .exe หรือ .com

ข้อดีของภาษาระดับสูง	ข้อจำกัดของภาษาระดับสูง
1. อำนวยความสะดวกแก่โปรแกรมเมอร์ โดยไม่ต้องมีความรู้ในเรื่องฮาร์ดแวร์ ก็สามารถเขียนได้	1. ยังต้องมีการแปลภาษาชั้นสูงเป็นภาษาเครื่องอยู่
2. มีลักษณะไม่ขึ้นอยู่กับอุปกรณ์ฮาร์ดแวร์ (Hardware Independent)	2. อาจประมวลผลช้ากว่าเขียนโดยภาษาเครื่องโดยตรงหรือภาษาแอสเซมบลี
3. มีเครื่องมือและเทคโนโลยีที่ช่วยเอื้อต่อการพัฒนาซอฟต์แวร์ได้ง่าย เช่น RAD (Rapid Application Development) เป็นต้น	3. เนื่องจากภาษาชั้นสูงนี้ยังจัดเป็นว่าเป็น Procedural Language จึงทำให้การพัฒนาซอฟต์แวร์ค่อนข้างซับซ้อน และใช้เวลาในการพัฒนาค่อนข้างมาก

การแปลโค๊ดให้เป็นภาษาเครื่องอีกวิธีหนึ่งเรียกว่าอินเตอร์พรีเตอร์ (Interpreter) ซึ่งเป็นการแปลโปรแกรมแต่ละคำสั่งให้เป็นภาษาเครื่องทีเดียว ซึ่งจะต้องทำการการตรวจสอบไวยากรณ์ของภาษา และทำงานคำสั่งนั้นทันที ก่อนที่จะทำการแปลคำสั่งถัดไปได้ ถ้าในระหว่างการแปลเกิดพบข้อผิดพลาดก็จะฟ้องเพื่อให้ทำการแก้ไขทันที ซึ่งโปรแกรมแปลภาษานี้เมื่อแปลเสร็จแล้วจะไม่สามารถเก็บเป็น execute program ได้ ดังนั้นเมื่อจะเรียกใช้งานก็จะต้องทำการแปลโปรแกรมใหม่ทุกครั้ง จึงทำให้โปรกรมทำงานช้ากว่าโปรแกรมที่ผ่านการคอมไพล์ แต่โปรแกรมภาษาที่แปลในรูปแบบนี้จะมีโครงสร้างที่ง่ายในการพัฒนา ตัวอย่างของโปรแกรมที่เป็นแบบอินเตอร์พรีเตอร์ ได้แก่โปรแกรมประเภทที่เป็นสคริปต์เช่น Perl, Java Script, Shell Script, VB Script รูปแบบทั่วไปของคำสั่งที่มีอยู่ในภาษาโปรแกรมประกอบด้วยคุณสมบัติดังต่อไปนี้

- การประกาศชนิดข้อมูลตัวแปร (Definition data type) จะเป็นการตั้งชื่อตัวแปร และบอกว่าเป็นตัวแปรชนิดใด เพื่อใช้อ้างอิงในหน่วยความจำ
- คำสั่งกำหนดค่าข้อมูลให้กับตัวแปร (Definition data value) ซึ่งอาจกำหนดค่าโดยตรงหรือให้รับจากการอ่านข้อมูลจากภายนอก เช่น รับค่าจากคีย์บอร์ดหรือแฟ้มข้อมูล
- คำสั่งควบคุม (Control command) เป็นคำสั่งควบคุมการทำงานเพื่อให้รู้ว่าต้องไปทำงานยังส่วนใดของโปรแกรม โดยอาจมาจากคำสั่งเงื่อนไขหรือคำสั่งวนซ้ำ
- คำสั่งการประมวลผล เป็นคำสั่งการคำนวณทางคณิตศาสตร์
- คำสั่งการแสดงผล เป็นคำสั่งให้แสดงผลลัพธ์ที่ได้จากการประมวลผล
- คำสั่งแสดงการเริ่มต้นและการสิ้นสุดการทำงาน ซึ่งอาจจะเป็นการเขียนบล็อกของโปรแกรม

ตัวอย่างโปรแกรมภาษาระดับสูง

FORTRAN = FORmula TRANslator เป็นภาษาที่พัฒนาโดยบริษัท IBM ตั้งแต่ปี ค.ศ. 1957 ถือเป็นภาษาระดับสูงภาษาแรก นิยมใช้สำหรับงานที่มีการคำนวณมากๆ เช่น งานด้านคณิตศาสตร์ วิทยาศาสตร์ วิศวกรรมศาสตร์ เป็นต้น การแปลภาษาใช้แบบคอมไพเลอร์ ซึ่งข้อดีของภาษานี้คือเหมาะกับงานที่มีการคำนวณทางคณิตศาสตร์ที่ซับซ้อน และคำสั่งส่วนใหญ่จะง่ายและสั้น ทำให้ผู้พัฒนาโปรแกรมไม่จำเป็นต้องมีการประกาศชนิดของตัวแปร อย่างไรก็ตามภาษานี้มีข้อจำกัดคือไม่เหมาะกับงานทางธุรกิจที่เกี่ยวข้องกับการรับข้อมูลเข้าและออก การสร้างรายงานจำนวนมากๆ หรืองานที่ต้องเก็บเป็นแฟ้มข้อมูล

COBOL = Common Business-Oriented Language เป็นภาษาที่พัฒนาเมื่อปี ค.ศ. 1960 นิยมใช้สำหรับงานทางธุรกิจ หรืองานเกี่ยวกับแฟ้มข้อมูล เช่น การจัดเก็บ การเรียกใช้ งานประมวลผลทางบัญชี ตลอดจนงานด้านควบคุมสินค้าคงคลัง และการรับจ่ายเงิน เป็นต้น ภาษาโคบอลเป็นภาษาที่เหมาะกับงานด้านธุรกิจขนาดใหญ่ที่มีข้อมูลมากๆ หรือใช้ในการออกรายงานที่ซับซ้อนและไม่ได้มุ่งเน้นที่ความสวยงาม ซึ่งโปรแกรมไม่ขึ้นอยู่กับชนิดหรือเครื่องคอมพิวเตอร์ อีกทั้งคำสั่งต่าง ๆ ใกล้เคียงกับภาษาอังกฤษ ทำให้ทำความเข้าใจได้ง่าย ใช้เป็นเอกสารอ้างอิงหรือคู่มือประกอบได้ ซึ่งปัจจุบันภาษาโคบอลนี้ก็ยังเป็นที่นิยมใช้ในงานธุรกิจด้านการเงิน การธนาคาร ซึ่งทำงานอยู่บนเครื่องแม่ข่ายขนาดใหญ่ เช่น เครื่องเมนเฟรม เป็นต้น อย่างไรก็ตามภาษานี้ยังมีข้อจำกัดอยู่ค่อนข้างมาก ตัวอย่าง เช่น การกำหนดโครงสร้างโปรแกรมที่แน่นอน และโปรแกรมที่เขียนจึงค่อนข้างยาวมากจึงไม่เหมาะกับงานที่มีการคำนวณที่ซับซ้อน

PASCAL มาจาก ชื่อของ Blasie Pascal ผู้ประดิษฐ์เครื่องคำนวณ Pascaline Calculator ซึ่งเป็นต้นแบบของคอมพิวเตอร์ในปัจจุบัน เป็นภาษาที่เอื้ออำนวยในการพัฒนาซอฟต์แวร์แบบโครงสร้าง นิยมใช้บนเครื่องไมโครคอมพิวเตอร์ มีตัวแปลภาษาทั้งแบบ compiler และ interpreter ซึ่งการจบคำสั่งแต่ละคำสั่งจะต้องมีเครื่องหมาย **“;”** เนื่องจาก Pascal เป็นโปรแกรมที่เขียนได้ง่าย สามารถประมวลได้เร็วและรูปแบบของคำสั่งมีความชัดเจนไม่จำกัดลักษณะงาน จึงเหมาะกับผู้เริ่มต้นพัฒนาซอฟต์แวร์โครงสร้าง ดังจะเห็นว่าจะใช้เป็นภาษาที่ใช้สอนในสถาบันการศึกษา ซึ่งภาษานี้ปัจจุบันยังเป็นที่นิยมอยู่ และนำมาเขียนในลักษณะของการพัฒนาเชิงวัตถุ Object Pascal ภายใต้ชื่อภาษาใหม่ว่า Delphi นอกจากนี้ภาษา Pascal ยังมีข้อจำกัดในคือไม่เหมาะกับงานด้านธุรกิจขนาดใหญ่ที่มีข้อมูลมากๆ หรือใช้ในการออกรายงานที่ซับซ้อนและสวยงาม

BASIC มาจาก Beginner’s All-purpose Symbolic Instruction Code มีจุดประสงค์สำหรับใช้ในการสอนนักศึกษาที่วิทยาลัย จากนั้นขยายวงกว้างมาใช้ในทางธุรกิจ นิยมใช้บนเครื่องไมโครคอมพิวเตอร์ มีลักษณะการทำงานแบบโต้ตอบ จึงเหมาะกับผู้เริ่มต้นพัฒนาซอฟต์แวร์ สำหรับในเวอร์ชันแรกๆ มีตัวแปลภาษาแบบ interpreter แต่เวอร์ชันหลังสามารถใช้ได้ทั้ง 2 แบบ ซึ่งข้อดีของภาษานี้คือ เนื่องจากเป็นโปรแกรมสามารถตรวจสอบและแก้ไขข้อผิดพลาดได้ทันทีจึงทำให้การพัฒนาซอฟต์แวร์สามารถทำได้อย่างรวดเร็ว อีกทั้งง่ายต่อการเรียนรู้ และสามารถใช้งานได้ทั้งเครื่องขนาดใหญ่

จนถึงเครื่องระดับเล็ก (Personal Computer: PC) ซึ่งภาษานี้มีข้อจำกัดคือไม่สนับสนุนโปรแกรมประเภทโครงสร้าง จึงไม่เหมาะกับงานขนาดใหญ่

C ถูกพัฒนาโดย Dennis Ritchie ปี ค.ศ. 1972 ณ ศูนย์วิจัย Bell Lab เป็นภาษามีประสิทธิภาพการทำงานใกล้เคียงภาษาแอสเซมบลี โดยมีจุดประสงค์สำหรับใช้เขียนซอฟต์แวร์ระบบ จากนั้นขยายวงกว้างมาใช้ในงานด้านต่าง ๆ ได้แก่ โปรแกรมระบบปฏิบัติการ ระบบจัดการฐานข้อมูล โปรแกรมประยุกต์ต่างๆโปรแกรมทางด้านกราฟิก เช่น เกมส์ ภาษาซีเป็นภาษาที่ได้รับความนิยมค่อนข้างมาก เนื่องจากเป็นภาษาที่รวมเอาข้อดีของภาษาระดับสูงในเรื่องความยืดหยุ่น และการพัฒนาโปรแกรมที่ง่ายรวม กับข้อดีของภาษาแอสเซมบลีในเรื่องการประมวลที่เร็วมากกว่าภาษาระดับสูงอื่น นอกจากนี้ยังสามารถพัฒนาซอฟต์แวร์โต้ตอบกับฮาร์ดแวร์ได้โดยตรง แต่ทั้งนี้ภาษาซีเองก็ยังมีข้อจำกัดคือไม่มีรูปแบบคำสั่งที่ตายตัว ทำให้เรียนรู้ยากกว่าภาษาระดับสูงอื่น การตรวจสอบโปรแกรมค่อนข้างยาก จึงไม่เหมาะกับงานที่เกี่ยวกับการออกรายงานที่ซับซ้อน

ภาษาระดับสูงมาก (Very High-Level Language)

มีลักษณะของภาษาที่ออกแบบมาเพื่อให้ง่ายต่อการใช้ง่ายและรวดเร็ว ซึ่งมักจะเห็นในลักษณะที่เป็นโปรแกรมแบบวิชวล (Visual Programming) คือมีลักษณะการทำงานที่ให้ผู้พัฒนาสามารถพัฒนาซอฟต์แวร์โดยวิธีการลาก-วาง (Drag and Drop) การเลือกใช้เครื่องมือวิซาร์ด (Wizard) จากการออกแบบหน้าจอในลักษณะโปรแกรมต้นแบบหรือโพรโตไทป์ โดยมีเครื่องมือที่สามารถออกแบบโปรแกรมและมองเห็นผลลัพธ์ได้ทันที โดยจะช่วยในการสร้างแบบฟอร์มบนหน้าจอ เพื่อจัดการกับข้อมูล รวมไปถึงการสร้างรายงาน ส่วนประกอบที่สำคัญได้แก่เครื่องมือที่ช่วยในการพัฒนาโปรแกรมการสอบถามข้อมูลหรือการคิวรี เช่นระบบฐานข้อมูลเชิงสัมพันธ์จะมีเครื่องมือชื่อว่า QBE (Query by Example) เครื่องมือในการสร้างรายงานและเครื่องมืออื่นๆเช่น

- เครื่องมือช่วยสร้างโปรแกรม (Application Generators) ที่มีรูปแบบการพัฒนาเป็นแบบวิชวล คือ การออกแบบและพัฒนาโปรแกรมโดยเริ่มต้นจากการออกแบบจอภาพ (Graphical User Interface) ที่ผู้ใช้งานต้องการ และความสามารถในการสร้างโปรแกรมโค๊ดอัตโนมัติ (Automatic Code Generation) ทำให้ผู้พัฒนาซอฟต์แวร์ลดระยะเวลาในการพัฒนาโปรแกรม
- ภาษาช่วยค้นหาข้อมูล (Query Languages) เป็นภาษาที่ช่วยค้นหาหรือดึงข้อมูลจากฐานข้อมูล ภาษานี้ง่ายต่อการใช้งาน เนื่องจากมีรูปแบบใกล้เคียงภาษาอังกฤษมาก ได้แก่ ภาษา SQL (Structured Query Language) หรือ QBE (Query-By-Example)
- เครื่องมือช่วยสร้างรายงาน (Report Generators or Report Writer) สามารถกำหนดเงื่อนไขและข้อมูลที่นำออกรายงาน รวมไปถึงรูปแบบของการออกรายงาน

สำหรับภาษาระดับสูงมากนี้ทำให้ผู้พัฒนาซอฟต์แวร์สามารถผลิตซอฟต์แวร์ได้อย่างรวดเร็ว อีกทั้งการลดความผิดพลาดของโปรแกรมให้น้อยลง ซึ่งความสามารถต่างๆสามารถสรุปได้ดังนี้

- เป็นภาษาที่ง่ายต่อการเรียนรู้ คำสั่งแต่ละคำสั่งสื่อความหมายได้ชัดเจน
- ประหยัดเวลาในการพัฒนาซอฟต์แวร์ได้มาก

- สนับสนุนระบบจัดการฐานข้อมูล ทำให้สามารถจัดการกับข้อมูลได้อย่างสะดวกและรวดเร็ว
- สามารถสร้างแบบฟอร์มเพื่อจัดการฐานข้อมูล และออกรายงานได้ง่าย
- สามารถทำงานในลักษณะโต้ตอบได้
- ผู้พัฒนาซอฟต์แวร์ไม่ต้องรู้รายละเอียดการทำงานของฮาร์ดแวร์ และโครงสร้างของโปรแกรม

ภาษาธรรมชาติ (Natural Language)

มีลักษณะการเขียนคล้ายภาษาธรรมชาติหรือ ภาษาพูดของมนุษย์ เป็นแบบไม่เป็นลำดับการปฏิบัติงาน (Non-Procedural Language) จึงไม่มีรูปแบบแน่นอนตายตัว คอมพิวเตอร์จะพยายามแปลประโยคสั่งงานเหล่านั้น ถ้าไม่เข้าใจจะถามย้อนกลับมาเพื่อยืนยันความถูกต้อง ถูกพัฒนามาจากเทคโนโลยีทางด้านระบบผู้เชี่ยวชาญ (Expert System) ซึ่งอยู่ในสาขาปัญญาประดิษฐ์ (Artificial Intelligence: AI) ซึ่งต้องเก็บข้อมูลต่างๆ เป็นฐานความรู้ (Knowledge base) เพื่อจะใช้ในการแปลความหมายคำสั่งต่าง ๆ

1.2 วิวัฒนาการของภาษาที่ใช้ในการพัฒนาโปรแกรม

การพัฒนาโปรแกรมคอมพิวเตอร์ได้มีการวิวัฒนาการมาตั้งแต่ในยุคเริ่มต้นของการสร้างเครื่องคอมพิวเตอร์ โดยในยุคนั้นยังไม่มีการพัฒนาโปรแกรมที่มีรูปแบบของภาษา (Programming Code) ที่หลากหลายและมีความสามารถสูงเหมือนในปัจจุบัน การพัฒนาซอฟต์แวร์จึงเป็นลักษณะการอาศัยมนุษย์ในการกำหนดการทำงานของเครื่องคอมพิวเตอร์เป็นส่วนใหญ่ ตัวอย่างเช่นการใช้ปุ่มเพื่อควบคุมการทำงานหรือการสลับสายไฟต่างๆ ต่อมาได้มีการนำเอาขั้นตอนการทำงานหรือชุดคำสั่งมารวบรวมบนกระดาษเจาะรู (Punch Card) โดยมีเครื่องสำหรับอ่าน ในสมัยนั้นอาจจะไม่มีการส่งให้เครื่องคอมพิวเตอร์ทำงานในลักษณะของการพัฒนาซอฟต์แวร์ เนื่องจากชุดคำสั่งไม่ได้อยู่ในรูปของโปรแกรมภาษา แต่ลักษณะของชุดคำสั่งส่วนใหญ่จะเป็นการสั่งงานให้เป็นภาษาเครื่อง (Machine Language) โดยตรงซึ่งเป็นเลขฐานสอง (Binary Code) ชุดคำสั่งเหล่านี้ส่วนใหญ่จะมีขนาดเล็กเนื่องจากคำสั่งมีความซับซ้อนและยากต่อการตรวจสอบหรือแก้ไขข้อผิดพลาดส่วนการแสดงผลก็เช่นกันก็จะแสดงผลในรูปของเลขฐานสอง อาจจะใช้หลอดไฟแสดงผลแทน เช่น ถ้าหลอดไฟปิดก็จะหมายถึงเลข 0 หรือ 1 เมื่อหลอดไฟเปิด การใช้ชุดคำสั่งเพื่อควบคุมการทำงานของเครื่องคอมพิวเตอร์แบบนี้มีความซับซ้อนมาก เนื่องจากว่าผู้พัฒนาซอฟต์แวร์เองจะต้องมีความเข้าใจในตัวฮาร์ดแวร์เป็นอย่างมาก และจะต้องเข้าใจภาษาเครื่องอีกด้วย จากปัญหานี้ทำให้มีความคิดที่จะพัฒนาจากชุดคำสั่งให้เป็นโปรแกรมภาษาที่ มีรูปแบบของไวยากรณ์ (Program Syntax) และโครงสร้าง (Programming Structure) ที่ชัดเจนมากขึ้น นักพัฒนาซอฟต์แวร์เองสามารถพัฒนาเป็นโปรแกรมได้โดยที่ไม่ต้องอาศัยความรู้เกี่ยวกับฮาร์ดแวร์

จากอดีตถึงปัจจุบัน เราจะเห็นว่าโปรแกรมคอมพิวเตอร์มีการเปลี่ยนแปลงเรื่อยมาโดยมีการพัฒนาให้อยู่ในรูปแบบของภาษาต่างๆ ซึ่งแต่ละภาษาก็แตกต่างกันไปไม่ว่าจะเป็นไวยากรณ์ โครงสร้างหรือความสามารถในการทำงานซึ่งทำให้นักพัฒนาสามารถเลือกภาษาที่เหมาะสมที่จะนำไปใช้พัฒนา

งานของตัวเองอย่างไรก็ตามเราอาจจะแบ่งโปรแกรมคอมพิวเตอร์ได้ตามลักษณะโครงสร้างของ โปรแกรมได้เป็น 4 ประเภทหลักดังนี้

1. โปรแกรมแบบไม่มีโครงสร้าง (Unstructured Programming)
2. โปรแกรมแบบมีโครงสร้างเป็นแบบโพซีเยอร์ (Procedural Programming)
3. โปรแกรมแบบมีโครงสร้างเป็นแบบโมดูล (Modular Programming)
4. โปรแกรมเชิงวัตถุ (Object-Oriented Programming)

1.2.1 โปรแกรมแบบไม่มีโครงสร้าง (Unstructured Programming)

ลักษณะของโปรแกรมแบบไม่มีโครงสร้างนั้นนับได้ว่าอยู่ช่วงเริ่มต้นของการพัฒนาโปรแกรมคอมพิวเตอร์ ซึ่งแนวคิดของการพัฒนาโปรแกรมในสมัยนั้นก็ยังเป็นที่นิยมใช้อยู่ในปัจจุบัน นั่นก็คือการพัฒนาโปรแกรมโดยเขียนเป็นโปรแกรม (Source Code) ซึ่งเป็นภาษาที่มนุษย์สามารถเข้าใจได้ หลังจากนั้นก็นำเอาโปรแกรมมาแปลงให้เป็นภาษาเครื่องโดยอาศัยโปรแกรมอีกประเภทหนึ่งที่เรียกว่าคอมไพเลอร์ (Compiler) ซึ่งวิธีนี้ทำให้นักพัฒนาโปรแกรมไม่จำเป็นที่จะต้องมีความรู้เกี่ยวกับภาษาเครื่อง ทำให้การพัฒนาโปรแกรมมีความสะดวกรวดเร็วมากยิ่งขึ้น อย่างไรก็ตามโครงสร้างของโปรแกรมภาษาแบบไม่มีโครงสร้างนี้ โปรแกรมจะถูกเขียนรวมกันทั้งหมดโดยการทำงานของโปรแกรมจะเรียงลำดับจากบนลงล่าง (Top-down) ไม่มีการบ่งบอกว่าส่วนไหนของโปรแกรมทำหน้าที่อะไร การพัฒนาซอฟต์แวร์ลักษณะนี้ ถ้าโปรแกรมมีขนาดใหญ่ขึ้น โปรแกรมก็จะเริ่มซับซ้อนและยากแก่การตรวจสอบข้อผิดพลาด อีกประเด็นหนึ่งคือเนื่องจากโปรแกรมจะถูกเขียนรวมกันเป็นชิ้นเดียวกันทั้งหมด หากต้องการที่จะทำซ้ำขั้นตอนใดขั้นตอนหนึ่ง โปรแกรมในส่วนนั้นๆก็จะต้องเขียนซ้ำอีก ทำให้โปรแกรมมีขนาดใหญ่เกินความจำเป็น ตัวอย่างของโปรแกรมประเภทนี้ได้แก่ ภาษา Assembly และภาษา Basic[1]

1.2.2 โปรแกรมแบบมีโครงสร้างเป็นแบบโพซีเยอร์ (Procedural Programming)

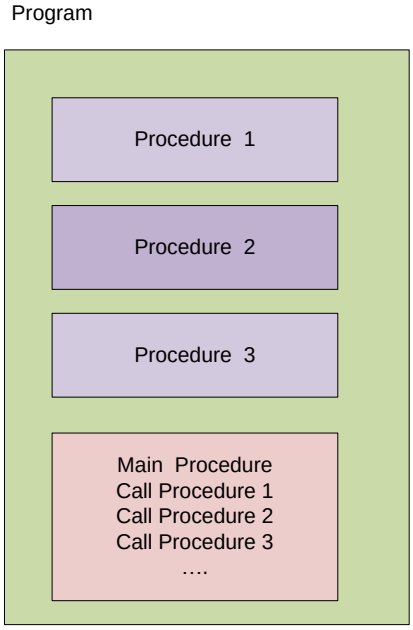
ลักษณะของโปรแกรมประเภทนี้ โปรแกรมจะถูกแบ่งออกเป็นโปรแกรมย่อยเรียกว่าโพซีเยอร์ (Procedure) หรือฟังก์ชัน (Function)[2] เรียกว่า Procedural Programming ดังนั้นโปรแกรมหนึ่งๆก็จะประกอบไปด้วยโปรแกรมย่อยหลายๆโปรแกรมในบรรดาโปรแกรมย่อยเหล่านี้จะต้องมีอย่างน้อยหนึ่งโปรแกรมย่อยที่ทำหน้าที่เป็นโปรแกรมย่อยหลักเรียกว่าโปรแกรมหลัก (Main Program) ซึ่งจะทำ

---

[1] ภาษา Basic เป็นภาษาหนึ่งที่มีการพัฒนาและเปลี่ยนแปลงเรื่อยมาตั้งแต่ในอดีตและปัจจุบันก็ยังจัดเป็นภาษาหนึ่งที่ได้รับความนิยมเช่น Quick Basic, Visual Basic, Active Server Page (ASP) และ ASP .NET

[2] โปรแกรมบางภาษาก็จำแนกข้อแตกต่างระหว่างโพซีเยอร์กับฟังก์ชันตัวอย่างเช่นภาษา Basic โดยกำหนดว่าโพซีเยอร์ไม่สามารถส่งค่ากลับได้ และฟังก์ชันสามารถส่งค่ากลับได้

หน้าที่เป็นจุดเริ่มต้นของการทำงานของโปรแกรม และจากโปรแกรมหลักก็สามารถเรียกใช้งานจากโปรแกรมย่อยโดยอาศัยหลักการของ Function Call ดังตัวอย่างในภาพนี้



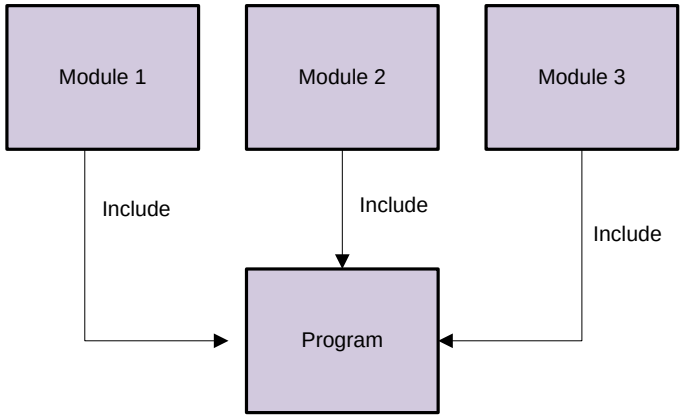
รูป 1.2 โปรแกรมแบบมีโครงสร้างแบบโพซีเยอร์

การแบ่งหน้าที่การทำงานออกเป็นส่วนย่อยๆนี้ แต่ละโพซีเยอร์ก็สามารถมีการส่งค่าไปหรือกลับ (Parameter Passing and Return Values) ทำให้การใช้งานโพซีเยอร์มีความยืดหยุ่นมากยิ่งขึ้น เนื่องจากมีการแบ่งหน้าที่การทำงานออกเป็นโปรแกรมย่อยต่างๆ และสามารถเรียกใช้งานโปรแกรมย่อยขึ้นมาทำงานกี่ครั้งก็ได้ ทำให้เกิดประสิทธิภาพในการนำเอาโปรแกรมกลับมาเรียกใช้โดยไม่ต้องเขียนซ้ำอีก นอกจากนี้การใช้โพซีเยอร์ก็ยังทำให้ซอสโค๊ดมีความเป็นระเบียบมากยิ่งขึ้น การทำงานแบบนี้นอกจากจะทำให้เกิดความคล่องตัวในการจัดการกับโครงสร้างของโปรแกรมแล้ว ก็ยังทำให้การจัดการและการตรวจสอบแก้ไขข้อผิดพลาดสามารถทำได้ง่ายอย่างไรก็ตาม การทำงานของโปรแกรมลักษณะนี้ยังเป็นลักษณะจากบนลงล่างอยู่ เพียงแต่ว่าขอบเขตของการทำงานจะอยู่ภายในแต่โพซีเยอร์นั่นเอง ตัวอย่างของโปรแกรมประเภทนี้ได้แก่โปรแกรมภาษา C, Pascal, Fortran, Cobol, Ada และ Quick Basic

1.2.3 โปรแกรมแบบมีโครงสร้างเป็นแบบโมดูล (Modular Programming)

สำหรับการพัฒนาโปรแกรมโดยแบ่งออกเป็นโพซีเยอร์หรือฟังก์ชัน เมื่อโปรแกรมมีขนาดใหญ่ขึ้นส่งผลให้มีโพซีเยอร์เกิดขึ้นมากมาย และการจัดการมีความยากขึ้นตามไปด้วย ดังนั้นจึงมีแนวคิดที่จะนำเอาโพซีเยอร์ย่อยที่ทำหน้าที่คล้ายกันมาจัดให้อยู่ในหมวดหมู่เดียวกันเรียกว่า โมดูล (Module) ซึ่งในแต่ละภาษาก็อาจจะมีชื่อเรียกแตกต่างกันไปตัวอย่างเช่น ภาษาซีเรียกว่าไลบรารี (Library) และใน

ภาษาปาสคาลเรียกว่ายูนิต (Unit) ซึ่งเราเรียกการจัดการโปรแกรมในลักษณะนี้โดยรวมว่าเป็น Modular Programming ซึ่งแต่ละโมดูลจะสามารถนำไปประกอบกันเป็นโปรแกรมได้ หรือสามารถนำกลับมาใช้ใหม่ ทำให้ลดขั้นตอนและลดเวลาในการพัฒนาโปรแกรมลง ดังแสดงในรูปภาพต่อไปนี้



รูป 1.3 การกำหนดโครงสร้างของโปรแกรมแบบ Modular Programming

สำหรับการพัฒนาโปรแกรมโดยแบ่งออกเป็นโมดูลแท้จริงแล้วเป็นหลักการเดียวกันกับการแบ่งโปรแกรมออกเป็นโพซีเยอร์ต่างๆ เพียงแต่โปรแกรมที่มีขนาดใหญ่ขึ้นอาจจะประกอบไปด้วยหลายโมดูลเข้าด้วยกัน และแต่ละโมดูลก็จะประกอบ ไปด้วยหลายๆโพซีเยอร์ที่ทำหน้าที่คล้ายกันมาอยู่รวมกันนั่นเอง การแบ่งโปรแกรมที่มีขนาดใหญ่ออก เป็นโมดูลย่อยต่างๆทำให้การจัดการกับโปรแกรมง่ายขึ้น ตัวอย่างเช่นสำหรับโครงการขนาดใหญ่ที่มีการพัฒนาซอฟต์แวร์แบบทีมงาน อาจจะมีการแบ่งงานออกเป็นโมดูลเพื่อให้แต่ละทีมงานรับผิดชอบซึ่งทำให้ ทีมงานสามารถพัฒนาโปรแกรมหลายๆโมดูลพร้อมกันได้ ข้อดีอีกอย่างหนึ่งสำหรับการแบ่งงานเป็นโมดูลคือความสามารถในการนำเอาโมดูลกลับมาใช้งานใหม่ได้อีก

การพัฒนาโปรแกรมแบบมีโครงสร้างนี้ปัจจุบันก็ยังเป็นที่นิยมสำหรับการพัฒนาซอฟต์แวร์ทั่วไปจนบางครั้งเราเรียกการพัฒนาซอฟต์แวร์แบบนี้ว่าเป็น Traditional Programming Language อย่างไรก็ตามการพัฒนาซอฟต์แวร์แบบนี้ ตัวแปรและฟังก์ชันย่อยจะถูกเขียนอยู่รวมกันซึ่งทำให้เกิดปัญหาในการควบคุมขอบเขตของการเข้าถึงข้อมูล (Scope of Variables) และการเรียกใช้งานฟังก์ชัน ปัญหานี้เองทำให้หน้าที่การทำงานของฟังก์ชันหรือขอบเขตของข้อมูลในแต่ละงาน ไม่สามารถแบ่งออกได้อย่างชัดเจนเมื่อโปรแกรมมีขนาดใหญ่ขึ้น

1.2.4 โปรแกรมเชิงวัตถุ (Object-Oriented Programming)

โปรแกรมเชิงวัตถุ คือ โปรแกรมที่มีโครงสร้างแบ่งออกเป็นออบเจ็กต์ต่างๆ หรือเราเรียกการพัฒนาโปรแกรมแบบนี้ว่าเป็นการพัฒนาโปรแกรมเชิงวัตถุ (Object-Oriented Programming) เรียกสั้นๆว่า OOP การพัฒนาโปรแกรมเชิงวัตถุมีแนวคิดของการพัฒนาโปรแกรมต่างจากการพัฒนาแบบ

โครงสร้าง โดยแนวคิดของการแก้ไขปัญหานั้นได้มองปัญหาออกเป็นวัตถุ (Object) และเป็นการนำเอาวัตถุต่างๆมาประกอบกันเป็นโปรแกรม แต่ละวัตถุก็จะมีความเป็นเอกเทศหรือมีความสมบูรณ์ในตัวเอง คือมีทั้งข้อมูล (Data members) และเมทธอดที่ใช้ในการเข้าถึงและจัดการกับตัวข้อมูล ดังนั้นในแต่ละโปรแกรมก็อาจจะมีวัตถุหลายๆวัตถุมาประกอบกันเป็นโปรแกรมขนาดใหญ่ โดยที่แต่ละวัตถุก็จะแยกหน้าที่ทำงานกันไป ทั้งนี้วัตถุแต่ละตัวสามารถติดต่อสื่อสารระหว่างกันได้ด้วยการเรียกใช้ออบเจ็กต์อื่นๆ เรียกว่า Message Passing จะเห็นว่าซอฟต์แวร์ประกอบไปด้วย Object A ซึ่งทำหน้าที่ในการแสดงส่วนติดต่อกับผู้ใช้ (Graphical User Interface) เพื่อรอรับคำสั่งหรือข้อมูลจากผู้ใช้ หลังจากนั้น Object A ก็จะเรียก Object B ซึ่งทำหน้าที่ในการเชื่อมต่อกับฐานข้อมูล เมื่อได้ข้อมูลแล้ว Object B ก็สามารถส่งข้อมูลกลับไปให้ Object A เพื่อทำการแสดงผลต่อไป

ในยุคของการพัฒนาโปรแกรมเชิงวัตถุนี้ หรือเรียกกันว่าเป็นยุคสมัยที่สี่ Fourth Generation Programming Language (4^{th} GL) เป็นยุคของการพัฒนาโปรแกรมแบบวิชวล (Visual Programming) ซึ่งตามหลักการของวิศวกรรมซอฟต์แวร์ (Software Engineering) ในปัจจุบันนี้ได้ให้ความสำคัญกับผลลัพธ์ของโปรแกรม (Program Output) เป็นหลัก เช่น การจัดทำต้นแบบของซอฟต์แวร์ (Program Prototyping) โดยผลลัพธ์นี้มักจะแสดงออกมาเป็นแบบกราฟิก เรียกว่า Graphical User Interface (GUI) ดังนั้นจะเห็นได้ว่าการพัฒนาโปรแกรมแบบวิชวลนี้สามารถให้นักพัฒนาซอฟต์แวร์สร้าง Output ขึ้นมาในรูปแบบของฟอร์มและรายงานในลักษณะของการลากและวาง (Drag and Drop) คอนโทรลต่างๆมาไว้บนฟอร์ม ยกตัวอย่างคอนโทรล เช่น InputBox, Static Text หรือ List Box ซึ่งที่จริงแล้วในมุมมองของการพัฒนาเชิงวัตถุ คอนโทรลแต่ละตัวก็คือแต่ละออบเจ็กต์นั่นเอง จะเห็นได้ว่าการพัฒนาซอฟต์แวร์ในลักษณะวิชวลนี้ทำให้นักพัฒนาซอฟต์แวร์สามารถพัฒนาโปรแกรมได้เร็วขึ้น และได้ผลลัพธ์ (รูปแบบหน้าจอ) ตรงตามความต้องการ นอกจากนี้การลากและวางคอนโทรลก็คือการเรียกใช้ออบเจ็กต์ที่ถูกสร้างขึ้นมาโดยคอมไพเลอร์แต่ละตัว ทำให้ช่วยลดระยะเวลาในการพัฒนาซอฟต์แวร์ หรือในการที่จะพัฒนาออบเจ็กต์นั้นขึ้นมาใหม่ ตัวอย่างของโปรแกรมที่สนับสนุนการทำงานแบบวิชวลนี้ได้แก่ Visual C++, Visual Java, Visual Foxpro และ Visual Basic

1.3 แนวคิดและหลักการพัฒนาโปรแกรม

ก่อนที่จะเริ่มมีการพัฒนาเป็นโปรแกรมนั้น นักพัฒนาโปรแกรมจะต้องทำความเข้าใจถึงวัตถุประสงค์และขอบเขตของปัญหา ซึ่งเรียกว่าการทำ User Requirement และจะต้องศึกษาความเป็นไปได้ของการแก้ไขปัญหา (Feasibility Study) ขั้นตอนเหล่านี้จัดว่าเป็นสิ่งสำคัญอย่างยิ่งเนื่องจากโปรแกรมที่พัฒนาจะสำเร็จหรือไม่ขึ้นอยู่กับความสามารถของนักพัฒนาโปรแกรมเองในการที่จะวิเคราะห์โจทย์หรือปัญหา และทำการวางแผนการพัฒนาล่วงหน้าก่อนที่จะเริ่มพัฒนาซอฟต์แวร์ ในการพัฒนาโปรแกรมแบบมีโครงสร้าง ขั้นตอนเหล่านี้จัดเป็นส่วนสำคัญอย่างหนึ่งหรือเรียกได้ว่าเป็นวงจรชีวิตของการพัฒนาโปรแกรม (Software Development Life Cycle: SDLC) ซึ่งผู้ใช้ก็จะต้องให้ความร่วมมือกับนักพัฒนาโปรแกรมเพื่อให้สำเร็จตามที่ได้คาดหวังเอาไว้ สำหรับการแก้ไขปัญหาโดยการ

พัฒนาโปรแกรมเชิงวัตถุนั้นเรียกว่าโดเมนปัญหา (Domain Problem) โดยการวิเคราะห์ปัญหานั้นจะอาศัยการมองในรูปแบบของออบเจ็กต์ ซึ่งมีความใกล้เคียงกับปัญหาที่เกิดขึ้นในโลกความเป็นจริง การวิเคราะห์ปัญหาตามแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุนี้จะ ทำให้นักพัฒนาซอฟต์แวร์ไม่จำเป็นต้องคำนึงถึงรายละเอียดของการพัฒนา ตัวอย่างเช่น โปรแกรมเครื่องคิดเลข ก็ยังไม่ต้องคิดว่าข้อมูลที่ป้อนเข้ามาจะเป็นตัวเลขจำนวนเต็มหรือเลขทศนิยม แต่จะมองว่าเครื่องคิดเลขนี้เป็นออบเจ็กต์เครื่องคิดเลขที่มีข้อมูลเป็นตัวเลขและฟังก์ชันที่ใช้จัดการกับข้อมูลนี้คือการบวก ลบ คูณและหาร และในการพัฒนาออบเจ็กต์ให้อยู่ในรูปของโปรแกรมออบเจ็กต์ที่ออกแบบจะถูกสร้างให้อยู่ในรูปของคลาส ซึ่งเราอาจจะเรียกได้ว่าออบเจ็กต์ก็คือคลาสนั่นเอง สำหรับการพัฒนาโปรแกรมที่มีขนาดใหญ่ขึ้น เราอาจจะมองว่าโปรแกรมหนึ่งๆก็อาจประกอบได้ด้วยหลายๆออบเจ็กต์นั่นเอง ซึ่งในหลักการแล้วออบเจ็กต์แต่ละตัวก็จะมีบทบาทและหน้าที่ที่ต่างกันไป

ในการเขียนชุดคำสั่งงานด้วยภาษาคอมพิวเตอร์จะมีแนววิธีการอยู่สองรูปแบบด้วยกันคือแบบที่เรียกว่า ภาษาเชิงกระบวนการคำสั่ง (Procedural Programming) และแนวเชิงวัตถุ (Objected Oriented Programming) การพัฒนาซอฟต์แวร์ภาษาเชิงกระบวนการคำสั่งจะเริ่มต้นจากการตั้งตัวแปร (Variable) เพื่อทำงานจองเนื้อที่ในหน่วยความจำและจัดเตรียมเนื้อที่ในการจัดเก็บข้อมูล จากนั้นก็เขียนขั้นตอนหรือลำดับในการที่จะสั่งให้เครื่องคอมพิวเตอร์ทำงานตามที่กำหนด ซึ่งลำดับในการทำงานเหล่านี้จะมีการจัดหมวดหมู่ในการทำงาน กล่าวคือลำดับการทำงานที่เป็นเรื่องเดียวกันหรือการทำงานเพื่อให้ได้ผลลัพธ์ที่ต้องการจะนำมารวมในหมู่เดียวกัน ตัวอย่างเช่น ระบบบัญชีเงินเดือน จะมีการตั้งตัวแปรชื่อ อัตราค่าจ้าง (RateOfPay) และ จำนวนชั่วโมงที่มีการว่าจ้าง (WorkingHours) และจะต้องมีการคำนวณว่าในแต่ละช่วงเวลา จะต้องคำนวณค่าจ้างที่ต้องจ่ายให้กับผู้รับจ้าง ดังนั้นก็จะมีพัฒนาซอฟต์แวร์หรือชุดคำสั่งงานเพื่อให้คำนวณจำนวนค่าจ้างที่ต้องจ่าย

ซึ่งในการเรียกโปรแกรมมาทำงานอาจมีการส่งค่าตัวแปรไปให้ เช่น อัตราค่าจ้าง และ จำนวนชั่วโมงการทำงาน และเมื่อคำนวณเรียบร้อยก็จะมีการส่งข้อมูลคืนมาให้ เป็นต้น

ส่วนการพัฒนาซอฟต์แวร์เชิงวัตถุจะแตกต่างจากภาษาเชิงกระบวนคำสั่ง ผู้พัฒนาโปรแกรมเชิงวัตถุจะต้องมีมุมมองที่อยู่ในรูปแบบของเชิงวัตถุ จากตัวอย่างข้างต้น ในการคำนวณหาค่าแรงของพนักงานแต่ละคนจะมีตัวแปรที่กำหนดความแตกต่างของพนักงาน คือ อัตราค่าจ้างต่อชั่วโมงอาจแตกต่างกัน และจำนวนชั่วโมงในการทำงานของพนักงานแต่ละคนก็แตกต่างเช่นกัน ดังนั้นตัวแปรเหล่านี้เราเรียกว่าเป็น คุณสมบัติของออบเจ็กต์ (Attribute) แต่กระบวนการหลักในการทำงานข้างต้นจะประกอบด้วยการคำนวณการคำนวณค่าจ้างที่ต้องจ่าย นอกจากนี้อาจมีรายการอื่นๆ อีก เช่น การคำนวณหาภาษีหัก ณ ที่จ่าย การพิมพ์สลิปเงินเดือน ซึ่งในเชิงออบเจ็กต์เรียกว่าพฤติกรรมของออบเจ็กต์ (Behaviour) ดังนั้นการมองแบบเชิงวัตถุจะต้องพยายามวิเคราะห์ปัญหาเพื่อมองภาพเป็นออบเจ็กต์และวิเคราะห์องค์ประกอบที่สำคัญของออบเจ็กต์ ความสัมพันธ์ระหว่างออบเจ็กต์ และการทำงานร่วมกันระหว่างออบเจ็กต์ ซึ่งจะนำไปสู่การออกแบบเป็นคลาสและการพัฒนาเป็นโปรแกรมต่อไป

บางครั้งการพัฒนาโปรแกรมเชิงวัตถุอาจมองว่าเป็นสร้างออบเจ็กต์โมเดลจากปัญหาจริง (Real World Object Modeling) เนื่องจากการแก้ไขปัญหาในเชิงวัตถุนั้น โมเดลของออบเจ็กต์จะถูกสร้างจากการวิเคราะห์ปัญหาจริงซึ่งมองว่าปัญหามีข้อมูลอะไร (Data) และสามารถมีเหตุการณ์อะไรเกิดขึ้นได้ (Behavior) การแบ่งโปรแกรมออกเป็นออบเจ็กต์ต่างๆ โดยที่แต่ละออบเจ็กต์มีความสมบูรณ์ในตัวเองทำให้เพิ่มความสามารถของซอฟต์แวร์คือการนำเอาโค๊ดกลับมาใช้อีก หรือสามารถเรียกใช้งานออบเจ็กต์ใดๆก็ได้ซึ่งขึ้นอยู่กับเหตุการณ์เรียกว่า Event-Driven Programming ในที่นี้จะขอยกตัวอย่างโปรแกรมที่มีหน้าจอเป็นลักษณะของฟอร์มและมีปุ่มต่างๆ เมื่อมีการเลือกกดปุ่มใดๆ เราจะเรียกว่าเป็นเหตุการณ์ของการกดปุ่ม (On_Click Event) ออบเจ็กต์ที่สัมพันธ์กับปุ่มนั้นๆก็จะถูกเรียกขึ้นมาใช้งาน

สิ่งสำคัญอย่างหนึ่งสำหรับการพัฒนาโปรแกรมเชิงวัตถุ คือ หลักการนำโปรแกรมกลับมาใช้ได้อีก (Code Reusability) จากในอดีตจะเห็นได้ว่าการพัฒนาซอฟต์แวร์ตัวหนึ่งจะใช้ระยะเวลานานบางครั้งอาจเป็นหลายๆเดือนหรืออาจจะเป็นปีก็ได้ การพัฒนาซอฟต์แวร์แบบออบเจ็กต์โอเรียนเท็ดโดยการจัดโครงสร้างของโปรแกรมออกเป็นออบเจ็กต์ต่างๆนั้น สามารถช่วยลดระยะเวลาในการพัฒนาโปรแกรมโดยการนำเอาออบเจ็กต์ที่มีอยู่แล้ว หรือออบเจ็กต์ที่ถูกสร้างโดยโปรแกรมเมอร์อื่น มาประกอบกันเป็นซอฟต์แวร์ใหม่ๆได้อีก ดังนั้นจะเห็นได้ว่าปัจจุบันมีการพัฒนาออบเจ็กต์ใหม่ๆขึ้นมามากมายโดยอาจจะมาจากนักพัฒนาโปรแกรมทั่วไปหรือจากผู้ผลิตซอฟต์แวร์อื่นๆเรียกว่าเป็น Third-party Packages

อีกตัวอย่างที่นิยมยกมาให้เห็นทางการพัฒนาซอฟต์แวร์เชิงวัตถุ คือ คลาสของรถ ตัวแปร หรือ คุณลักษณะที่ใช้ในการอธิบายความแตกต่างของรถแต่ละคัน คือ ยี่ห้อ สี และจำนวนผู้โดยสารที่สามารถนั่งได้ก็เป็นอีกตัวแปรที่ใช้อธิบายความแตกต่างในเชิงคุณสมบัติของตัวรถ ส่วนพฤติกรรมของรถ เช่น สามารถขับเคลื่อนไปข้างหน้า ถอยหลัง และการหยุดเป็นต้น ซึ่งข้อมูลเหล่านี้จะถูกรวบรวมมาเป็นเสมือนพิมพ์เขียวที่ใช้ในการผลิตรถ และรถแต่ละคันที่ผลิตออกมาเราจะเรียกว่า เป็นอินสแตนซ์ (Instance) ซึ่งแต่ละอินสแตนซ์จะมีคุณสมบัติ (Attribute) และเมทธอด (Method) เป็นของตัวเอง

แนวการพัฒนาเชิงวัตถุอีกแนวหนึ่งก็คือ ความพยายามในการทำตามแบบสถาปัตยกรรมทางฮาร์ดแวร์ จะเห็นว่าในทางฮาร์ดแวร์ ผู้ใช้สามารถเลือกซื้ออุปกรณ์ต่างๆ ตามที่ต้องการ สิ่งที่ต้องทราบคือว่าส่วนเชื่อมประสานเป็นแบบที่ต้องการหรือไม่ แนวคิดนี้จะคล้ายกับการติดตั้งอุปกรณ์ฮาร์ดแวร์ของคอมพิวเตอร์ที่สามารถมีอุปกรณ์เมนบอร์ด ฮาร์ดดิสต์ หน่วยความจำ และซีดีรอม ซึ่งหากมีอุปกรณ์ใดเสียหรือไม่สามารถใช้งานได้ ก็สามารถที่จะเปลี่ยนเฉพาะอุปกรณ์นั้นๆโดยที่จะมองว่าอุปกรณ์ต่างๆนั้นเสมือนกับว่าเป็นออบเจ็กต์ต่างๆที่สามารถนำมาประกอบกัน และทำงานร่วมกันเป็นเครื่องคอมพิวเตอร์หนึ่งเครื่องนั่นเอง ซึ่งสิ่งที่เราต้องทราบคือมาตรฐานที่สามารถเข้ากันได้ ซึ่งเทียบได้กับอินเตอร์เฟสในการพัฒนาซอฟต์แวร์เชิงวัตถุ และสิ่งที่สำคัญของออบเจ็กต์ที่สามารถประกอบการทำงานคือการที่ออบเจ็กต์จะส่งข้อมูลหรือประสานกันจะทำได้อย่างไร

ในการออกแบบเชิงวัตถุที่สำคัญคือการรวบรวมคุณสมบัติและพฤติกรรม ซึ่งคุณสมบัติจะแทนด้วยตัวแปรคือ สิ่งที่ใช้ในการอธิบายความแตกต่างระหว่างวัตถุต่างๆ เช่น ตัวอย่างเรื่อง รถ โครงของตัวถังของรถอาจเหมือนกัน แต่การตกแต่งจะทำให้รถแต่ละคันแตกต่างกัน เช่น เครื่องยนต์ที่ใส่ในรถแต่ละ

คัน ก็ทำให้ราคาขายแตกต่างกัน หรือ อย่างกรณีของรูปทรงเรขาคณิต เส้นตรง สามเหลี่ยม วงกลม ทั้งหมดถือเป็นรูปทรงเรขาคณิต แต่จำนวนจุดที่ใช้ในการสร้างรูปทรงแตกต่างกันทำให้รูปทรงที่เห็นแตกต่างกัน ส่วนพฤติกรรม คือ บริการ (Service) หรืองานที่วัตถุต้องทำ เช่น กรณีของรูปทรงเรขาคณิต เราต้องการบริการคือการวาดรูปทรงนั้นๆ หรือ ของรถอาจเป็นการขับเคลื่อนของรถ เป็นต้น ดังนั้นจะเห็นว่าการพัฒนาเชิงวัตถุสอดคล้องกับความเป็นจริงในโลกมนุษย์มากขึ้น ถ้าต้องการฟังก์ชันงานใดเพิ่มก็เพียงแต่ซื้อฟังก์ชันมาใส่เพิ่มโดยที่เราไม่จำเป็นต้องทราบว่าฟังก์ชันนั้นมีกรรมวิธีในการทำงานอย่างไร หรือไม่ต้องทราบว่าภายในฟังก์ชันมีรายละเอียดอะไรบ้าง แต่สิ่งที่ต้องทราบคือส่วนเชื่อมประสาน หรือ ส่วนที่แต่ละออบเจ็กต์จะติดต่อกัน ดังนั้นในการออกแบบหรือพัฒนาซอฟต์แวร์เชิงวัตถุ ภาษาคอมพิวเตอร์ ต้องมีคุณสมบัติที่สนับสนุนการพัฒนาซอฟต์แวร์เชิงวัตถุ ซึ่งต้องประกอบด้วยหลักการ ดังนี้

1. ความสามารถในการสืบทอด (Inheritance) เป็นเทคนิคที่สำคัญในการสนับสนุนแนวคิดของการนำเอาโปรแกรมโค๊ดกลับมาใช้งานใหม่ โดยวิธีการใช้ประโยชน์จากโปรแกรมโค๊ดที่เขียนขึ้นอยู่เดิม และโปรแกรมใหม่นี้จะใช้ประโยชน์จากสิ่งที่มีอยู่เพื่อการต่อยอดให้มีการทำงานที่เฉพาะเจาะจงมากยิ่งขึ้น โดยอาศัยเทคนิคเรียกว่าการสืบทอดและสืบสกุล โดยออบเจ็กต์ที่สร้างขึ้นนั้นไม่จำเป็นต้องเป็นออบเจ็กต์ใหม่ทั้งหมด แต่อาจจะเป็นออบเจ็กต์ที่พัฒนาขึ้นมาเป็นแบบออบเจ็กต์ลูก (Child Object) และมีการสืบทอดจากอีกออบเจ็กหนึ่ง (Parent Object) โดยที่ออบเจ็กต์แม่จะมีการทำงานที่เป็นกลาง (Generalization) และ ออบเจ็กต์ลูกที่สืบทอดจะมีการทำงานที่เฉพาะเจาะจง (Specialization) ลงไปในงานนั้นๆ ดังตัวอย่างออบเจ็กต์ Window ซึ่งการแสดงผลหน้าจอวินโดว์ที่มีค่าตำแหน่ง Coordinate X1, Y1, X2, Y2 มีขนาด สี ซึ่งเป็นหน้าต่างปกติ ที่สามารถถูกนำไปสืบทอดเป็นออบเจ็กต์หน้าต่างในรูปแบบต่างๆเช่น Dialog Window หรือ Pop-up Window โดยที่ออบเจ็กต์หน้าต่างที่สืบทอดนั้นสามารถสืบทอดเอาคุณสมบัติต่างๆมาจากออบเจ็กต์แม่ ทั้งในส่วนของคุณสมบัติและการพฤติกรรม โดยที่จะลดความซ้ำซ้อนในการพัฒนา ลดข้อผิดพลาด และเพิ่มประสิทธิภาพในการพัฒนาโปรแกรมมากยิ่งขึ้น
2. ความสามารถในการเก็บซ่อนข้อมูล (Data Encapsulation) โดยที่แนวคิดนี้มองว่าแบ่งสถาปัตยกรรมของซอฟต์แวร์เป็นออบเจ็กต์นั้น ควรจะต้องมีวิธีการป้องกันคุณสมบัติของออบเจ็กต์จากการมองเห็นและการเข้าถึงโดยออบเจ็กต์อื่นๆ ซึ่งทั้งนี้รวมถึงคุณสมบัติและพฤติกรรมด้วย ซึ่งการเก็บซ่อนนั้นสามารถกำหนดได้ตามความเหมาะสม ตามที่แต่ออบเจ็กต์จะอนุญาติให้สามารถมองเห็นได้ซึ่งทั้งนี้เพื่อให้เหมาะสมต่อการเรียกใช้งานออบเจ็กต์ ตัวอย่างเช่น ออบเจ็กต์รถยนต์ หากต้องการให้รถยนต์ติดเครื่อง ผู้ใช้งานก็เพียงแค่เรียกใช้พฤติกรรม StartEngine() ของออบเจ็กต์ ซึ่งผู้ที่ใช้งานออบเจ็กต์รถยนต์ก็คงไม่ต้องรู้ว่า การจ่ายไฟ จ่ายน้ำมัน หรือหัวเทียน ทำงานอย่างไร

3. ความสามารถในการใช้งานได้หลายรูปแบบ (Polymorphism) ในการสืบทอดคุณสมบัติ ไม่จำเป็นเสมอไปที่จะต้องสืบทอดคุณสมบัติทุกอย่าง ในบางครั้งอาจต้องมีการแก้ไขลำดับในการทำงานของฟังก์ชันเดิมเพื่อให้การพัฒนาไม่ต้องเสียเวลาไปแก้ไขชุดคำสั่งเดิม ซึ่งถ้ามีการแก้ไขชุดคำสั่งเดิม จะทำให้ต้องเสียเวลาในการตามแก้ไขโปรแกรมต่างๆ ที่ใช้ฟังก์ชันที่ถูกแก้ไข วิธีการที่ดีคือ อนุญาตให้มีการเขียนชุดคำสั่งใหม่ภายใต้ชื่อฟังก์ชันเดิมที่มีอยู่ เช่น ในกรณีของฟังก์ชัน draw() ในคลาส shape จะเห็นว่าการวาดรูปทรงต่างๆ จะมีกรรมวิธีในการวาดที่แตกต่างกัน ดังนั้นคลาสของ Line, Polygon, Circle และ Rectangle สามารถที่จะเขียนชุดคำสั่ง draw() คือมีชื่อฟังก์ชันซ้ำกันแต่การทำงานภายในมีความแตกต่างกัน

ความสามารถในการจัดการโครงสร้างข้อมูลแบบเชิงนาม หรือ แอ็บแสต็ก (Abstract) ความหมายของข้อมูลแบบเชิงนาม คือ ข้อมูลหรือกระบวนการที่ยังไม่เกิดขึ้นจริง ซึ่งในการพัฒนาระบบงานข้อมูลอาจจะยังไม่เกิด หรือ ยังไม่สามารถที่จะอธิบายได้ว่าขั้นตอนในการทำงานเป็นอย่างไร แต่เพื่อไม่ให้งานต้องหยุดชะงัก การสร้างคลาสเป็นเชิงนามก็จะเป็นอีกวิธีหนึ่ง ซึ่งในการที่จะพิจารณาว่าภาษาใดเป็นภาษาเชิงวัตถุ ภาษาดังกล่าวควรมีคุณสมบัติตามที่ได้กล่าวมา

1.4 การนำเอาโปรแกรมโค๊ดกลับมาใช้ใหม่ (Code Reusability)

หัวใจสำคัญของการออกแบบและพัฒนาซอฟต์แวร์เชิงวัตถุ คือ การพัฒนาออบเจ็กต์ใหม่ ๆ ขึ้นมาใช้งาน โดยอาศัยการพัฒนาต่อยอดจากออบเจ็กต์เดิมที่มีอยู่ ซึ่งในหลักการนี้เราเรียกว่า Code Reusability ซึ่งแนวคิดนี้เป็นแนวคิดหลักของออบเจ็กต์เพื่อให้การพัฒนาซอฟต์แวร์สามารถกระทำได้อย่างรวดเร็ว ลดข้อผิดพลาดการทำงานของซอฟต์แวร์ เนื่องจากออบเจ็กต์หรือ คอมโพเนนท์ที่จะทำการต่อยอดนั้นอาจจะได้รับการแก้ไขปัญหาหรือได้รับการทดสอบมาเป็นระยะเวลาหนึ่งแล้ว ซึ่งแนวคิดนี้ได้นักวิจัย Szyperski (1997) ได้เสนอรูปแบบของคอมโพเนนท์ ที่เหมาะสมที่จะเป็น unit deployment และสนับสนุนแนวคิดแบบ component-oriented เพื่อการส่งต่อเรียกได้ว่า “off-the-shelf” คอมโพเนนท์ ทำให้การนำกลับมาใช้ได้ใหม่ ซึ่งเหมาะสำหรับการพัฒนาในลักษณะที่เป็นแอพพลิเคชั่นขนาดใหญ่

สำหรับแนวคิดของการนำเอาออบเจ็กต์กลับมาใช้งานใหม่ (Object Reusability) หมายถึงการนำเอาส่วนประกอบของซอฟต์แวร์กลับมาใช้ใหม่ ซึ่งคำว่าส่วนประกอบในที่นี้ ไม่จำเป็นต้องเป็นโมดูล หากเราจะนิยามเรื่องการนำกลับมาใช้ใหม่แบบง่ายๆ ก็คือ ความสามารถที่เรานำเอาโปรแกรมหรือส่วนใดส่วนหนึ่งของโปรแกรมโค๊ด นำไปใช้ทำงานร่วมกับโปรแกรมใหม่ได้ แม้ผู้ใช้โปรแกรมจะไม่ได้รู้ถึงรายละเอียดภายในของโค๊ดนั้นๆว่าทำได้อย่างไร แต่รู้ว่าสามารถทำอะไรได้ เสมือนกับเป็นกล่องดำที่เพียงสนใจแต่ input ที่รับเข้าและ output ที่ส่งออกมาเท่านั้น แต่การที่เราสามารถนำโค๊ดมาใช้ใหม่ได้นั้น ทำให้เราไม่ต้องทำงานซ้ำซ้อน ผลก็คือทำให้โปรแกรมพัฒนาได้รวดเร็วขึ้น และมีข้อผิดพลาดที่น้อยลง ส่งผลให้เรื่องของการผลิตซอฟต์แวร์มีประสิทธิภาพนั่นเอง ดังนั้นเราพอจะสรุปถึงความสำคัญของการนำเอาโค๊ดกลับมาใช้ ได้ดังนี้

1) ลดเวลาและความพยายามในการสร้างซอฟต์แวร์ใหม่
2) ลดค่าใช้จ่ายในการสร้างซอฟต์แวร์ใหม่
3) เพิ่มคุณภาพของระบบซอฟต์แวร์
4) ลดเวลาและความพยายามในการดูแลระบบ

การนำเอาโปรแกรมโค๊ดกลับมาใช้ใหม่ หากนักพัฒนาพบว่าส่วนประกอบของผลิตภัณฑ์เก่าที่ได้พัฒนาไว้สามารถนำมาใช้กับผลิตภัณฑ์ใหม่ที่กำลังพัฒนาอยู่ได้ ในระยะหลัง การพัฒนาระบบมักเกิดการเปลี่ยนแปลงความต้องการต่างๆ ดังต่อไปนี้

1) ฮาร์ดแวร์หรือซอฟต์แวร์ระบบ เช่น การเปลี่ยนระบบปฏิบัติการที่ต้องการใช้
2) ผู้ใช้งานที่ได้รับมอบหมายหรือการติดตั้ง เช่น การเปลี่ยนผู้ใช้หรือเปลี่ยนแปลงความต้องการในการใช้งานระบบ
3) ฟังก์ชันหรือคุณสมบัติในการใช้งาน เช่น การเปลี่ยนแปลงขั้นตอนหรือลักษณะการทำงานในรายละเอียด

การนำเอาโปรแกรมโค๊ดกลับมาใช้ (Code Reuse) นั้นนอกจากจำแนกออกเป็นประเภทแล้วยังสามารถจำแนกได้ตามระดับการ Reuse ซอฟต์แวร์ที่นำเอาโค๊ดกลับมาใช้ ดังนี้

Reuse ระดับที่ 1 – ระดับฟังก์ชัน (Function)

ระดับที่ 1 หมายถึง ฟังก์ชันในอดีตที่เราพัฒนาซอฟต์แวร์นั้น เราเขียนภาษา Assembly หรือภาษา BASIC รุ่นแรกๆ จะไม่มีฟังก์ชันให้ใช้ การพัฒนาซอฟต์แวร์ก็เลยใช้คำสั่ง GOTO ซึ่งส่งผลให้โปรแกรมมีลักษณะที่ไม่เป็นระเบียบหรือไม่เป็นสัดเป็นส่วน ซึ่งการใช้ฟังก์ชันในยุคแรก ๆ มีลักษณะของการยุบส่วนโปรแกรมที่เขียนซ้ำซ้อนไปเขียนอยู่ในที่เดียวกันเพื่อที่จะได้ประหยัดทรัพยากรต่าง ๆ ของระบบโดยเฉพาะอย่างยิ่งก็คือหน่วยความจำ ซึ่งเครื่องคอมพิวเตอร์สมัยก่อนหน่วยความจำจะมีจำกัด

หลังจากนั้นกว่า 20 ปี ความคิดเกี่ยวกับการ Reuse จึงเกิดขึ้น มีการสังเกตว่า เมื่อเราใช้ฟังก์ชันในการสร้างเนื้องานใดๆผู้ที่มาเรียกใช้นั้นไม่จำเป็นต้องรู้รายละเอียดภายในเลย (Encapsulation) ผลลัพธ์ก็คือ เกิดการเรียกใช้ซ้ำๆ กันได้จากหลายโปรแกรม ทำให้ลดงานลงไปได้ ข้อผิดพลาดของโปรแกรมก็ลดน้อยลง เพราะว่าเนื้อหาในฟังก์ชันนั้นถูกทดสอบมาแล้วหลายโปรแกรม และฟังก์ชันที่เกิดขึ้นในยุคแรกๆ นั้นมาจากภาษา FORTRAN ซึ่งนำเอาฟังก์ชันทางคณิตศาสตร์เช่น sin, cos, tan มาสร้างเป็นฟังก์ชัน นั่นจึงเป็นที่มาของคำว่า "ฟังก์ชัน" ถึงยุคปัจจุบัน ฟังก์ชันในยุคต่อมามีการพัฒนาขึ้นเพื่อเพิ่มพารามิเตอร์ต่างๆ เพื่อเพิ่มประสิทธิภาพอีกทั้งความยืดหยุ่นของการทำงานของฟังก์ชัน ส่งผลให้เราสามารถพัฒนาซอฟต์แวร์ได้ใหญ่ขึ้น จากนั้นแนวคิดนี้จึงเป็นที่มาของ Software Engineering เพื่อสร้างคุณภาพให้กับซอฟต์แวร์โดยในยุคนั้น Top-Down Programming เป็นที่นิยมมาก

Reuse ระดับที่ 2 – ระดับโมดูล (Module)

โมดูลถือว่าเป็นการทำ Reuse ระดับที่ 2 ซึ่งภาษาที่ผลักดันในการใช้ โมดูลให้เป็นที่นิยมก็คือภาษา C โดยที่เรียกว่าเป็นไลบรารี (Library)โมดูลเป็นการจัดขอบเขตของโปรแกรมใหม่ที่สูงกว่าฟังก์ชัน หรืออาจกล่าวได้ว่าโมดูลเป็นที่รวมของฟังก์ชันที่มีลักษณะการทำงานที่เหมือนกันหรือคล้ายกันมาอยู่ด้วยกัน มีตัวแปรและฟังก์ชันที่เรียกใช้อยู่ในเฉพาะส่วนของโมดูล ตัวอย่างที่นิยมของโมดูลคือ เรื่องการจัดการกับ Input และ Output เช่น การเรียกใช้ฟังก์ชัน Print โดยมีการส่งค่าเป็นพารามิเตอร์ของข้อมูลที่ต้องการพิมพ์ โดยผู้ที่เรียกใช้งานฟังก์ชันนี้ไม่จำเป็นต้องรู้ว่าฟังก์ชันมีการติดต่อกับจอภาพอย่างไร ซึ่งขั้นตอนการจัดแบ่งโปรแกรมเป็นแบบโมดูลนี้ เป็นการพัฒนาแบบก้าวกระโดด เพราะการที่รวบเอาเรื่องที่เกี่ยวกันเข้าด้วยกัน (Abstraction) ส่งผลให้โมดูลมีความสัมพันธ์กับโปรแกรมที่เรียกใช้ค่อนข้างหลวม (Loosely Couple) ทำให้โมดูลนั้นสามารถนำไปใช้กับโปรแกรมอื่นได้

Reuse ระดับที่ 2.5 – ระดับคลาส (Class)

แนวคิดของคลาสนั้นเป็นแนวคิดที่คล้ายกับโมดูลซึ่งเป็นแนวคิดของออบเจ็กต์ จริงแล้วแนวคิดของคลาสนั้นมีมานานแล้วตั้งแต่ยุคของภาษา Simula ซึ่งน่าจะก่อนที่มีการคิดค้นโมดูล แนวคิดของคลาสมีระดับการนำกลับมาใช้สูงกว่าโมดูลเนื่องจากทุกอย่างที่มีในโมดูลก็มีในคลาสด้วย แต่สิ่งที่คลาสต่างจากโมดูลก็คือ โมดูลจะมีเพียง Copy เดียวในหน่วยความจำ แต่คลาสนั้นมีจำนวน Copy ได้ไม่จำกัด (ซึ่งในที่นี้เรามองว่าคลาสสามารถนำไปสร้างเป็นออบเจ็กต์ในหน่วยความจำ) ประโยชน์คือทำให้การพัฒนาซอฟต์แวร์ได้ง่ายขึ้น สมมุติว่าเราสร้างโมดูลในการจัดการกับระบบฐานข้อมูลแต่เป็นไปได้ที่เราจะติดต่อกับฐานข้อมูลหลายตัวพร้อมกันเช่นการพัฒนาซอฟต์แวร์เพื่อโอนข้อมูลจากโปรแกรม Microsoft Access ไปยัง Microsoft SQL Server ดังนั้นในโมดูลเราจำเป็นต้องมีระบบการจัดการฐานข้อมูลหลายตัว แต่ในทางกลับกันถ้าเราใช้คลาสเราก็จะไม่ต้องสนใจเรื่องนั้น เพียงแค่เราสร้างคลาสที่จัดการฐานข้อมูลเพียงตัวเดียว โดยอาศัยกลไกของการพัฒนาเชิงวัตถุทำให้เราสามารถสร้างการติดต่อกับฐานข้อมูลได้หลายตัว โดยที่แต่ละตัวมีความอิสระต่อกัน

Reuse ระดับที่ 3 – ระดับไลบรารี (Library)

สำหรับการพัฒนาโปรแกรมโค๊ดให้อยู่ในระดับไลบรารีนี้เป็นการวิเคราะห์ว่าโปรแกรมในส่วนไหนที่สามารถแชร์ร่วมกันกับโปรแกรมโค๊ดในส่วนอื่นๆได้ ซึ่งจะถูกนำมาเก็บไว้ในไลบรารี ซึ่งไลบรารีเป็นการรวมโมดูล (ซึ่งแต่ละโมดูลอาจประกอบด้วยหลายฟังก์ชัน) ที่มีลักษณะการทำงานที่เหมือนหรือคล้ายกัน และในทางการจัดเก็บเชิง Physical Storage แล้วถือว่าเป็นการรวมเอาแฟ้มต่างๆ มาเก็บไว้เป็นแฟ้มเดียวกัน ตัวอย่างเช่นโมดูล Stream ก็จะรวมถึงฟังก์ชันที่ทำงานเกี่ยวข้องกับ Stream Input และ Output ภาษาที่เป็นนิยมใช้ไลบรารีคือภาษา C, Pascal, Modular ฯลฯ

ประเด็นสำคัญที่ควรกล่าวถึงคือ ไลบรารีนี้เป็นอิสระกับโปรแกรม ทำให้เวลาคอมไพล์นั้นต้องมีขั้นตอนเพิ่มขึ้นหนึ่งขั้น โดยขั้นแรกเป็นขั้นตอนของการคอมไพล์โปรแกรมโค๊ดให้เป็นภาษาเครื่อง machine code แต่โปรแกรมที่ได้ยังเอาไปใช้งานไม่ได้ เพราะยังไม่มีโค๊ดในส่วนของไลบรารีที่เรา

เรียกใช้ ซึ่งจะต้องทำขั้นตอนที่ 2 ต่อนั่นก็คือการ Link ซึ่งเราเอาไว้ Link กับไลบรารีที่คอมไพล์มาก่อนหน้าจากที่อื่น แล้วจึงกลายเป็นโปรแกรมที่ทำงานได้

การที่สามารถคอมไพล์แยกส่วน (Separate Compile) นั้น ส่งผลทำให้วงการอุตสาหกรรมซอฟต์แวร์มีความตื่นตัวมากขึ้น เนื่องจากมีหลายบริษัทที่เริ่มพัฒนาโมดูลหรือไลบรารีในลักษณะที่เป็น Third-Party เพื่อที่สนับสนุนโปรแกรมอื่นๆทั้งในเชิงของการวิจัยเพื่อเผยแพร่หรือในเชิงการค้า ซึ่งเราเรียกซอฟต์แวร์เหล่านี้ว่าเป็น Component Based Programming อีกทั้งโมดูลเหล่านี้มีการเผยแพร่ในลักษณะของภาษาเครื่อง Machine Code จึงไม่จำเป็นต้องกังวลเกี่ยวกับการรั่วไหลของโค๊ดหรือเทคนิค Algorithm ที่ใช้ซึ่งส่งผลให้ปัจจุบันมีไลบรารีเกิดขึ้นมากมาย ทำให้นักพัฒนาซอฟต์แวร์ส่วนใหญ่ไม่เป็นเพียงแค่นักสร้างโปรแกรมแต่ยังกลายเป็นนักประกอบไลบรารีไปด้วย

Reuse ระดับที่ 3.5 – ระดับคลาสไลบรารี (Class Library)

คลาสไลบรารีจัดได้ว่าเป็นไลบรารีอีกประเภทหนึ่ง ซึ่งเป็นการนำเอาหลักการของออบเจ็กต์มาใช้อย่างเต็มรูปแบบ โดยที่คลาสไลบรารีจะเก็บรวบรวมคลาสต่างๆ ที่มีลักษณะการทำงานคล้ายกันอยู่ในไลบรารีเดียวกัน ซึ่งเราจะเรียกไลบรารีเหล่านี้ว่าเป็น Package เช่นในภาษาจาวา หรือ Microsoft Foundation Class หรือ .NET Framework สำหรับซอฟต์แวร์ของไมโครซอฟท์ โดยการจัดเก็บคลาสไลบรารีส่วนใหญ่มีวัตถุประสงค์เพื่อที่จะนำเอาคลาสเหล่านั้นไปใช้งาน หรือในลักษณะของการสืบทอด/สืบสกุลจากคลาสที่มีอยู่ในไลบรารี เพื่อลดความซ้ำซ้อนจากสิ่งที่เคยพัฒนาหรือมีอยู่เดิมแล้ว ซึ่งหลักการนี้เรียกว่า การทำ Class Inheritance ซึ่งเป็นการถ่ายทอดคุณสมบัติจากคลาสหนึ่งไปสู่อีกคลาสหนึ่ง ทำให้เกิดการ Reuse โปรแกรมโค๊ดอย่างมีประสิทธิภาพ อีกทั้งในการพัฒนาซอฟต์แวร์ที่มีขนาดใหญ่มีทีมงานพัฒนาจำนวนมาก การแชร์คลาสไลบรารีเดียวกัน ทำให้ภาพรวมของซอฟต์แวร์ออกมาในรูปแบบและทิศทางการพัฒนาเดียวกัน และเมื่อคลาสใหม่ๆที่มีการพัฒนาและผ่านกระบวนการทดสอบเพื่อให้มั่นใจในความถูกต้องแล้ว คลาสนั้นๆ ยังสามารถนำมาเพิ่มลงในคลาสไลบรารีได้อีกด้วย ทำให้นักพัฒนาซอฟต์แวร์ที่รู้จักการเรียกใช้คลาสไลบรารีสามารถพัฒนาซอฟต์แวร์ใหม่ๆ ได้อย่างรวดเร็ว

Reuse ระดับที่ 4 - ระดับคอมโพเนนท์ (Component)

คอมโพเนนท์จัดได้ว่าเป็นคลาสไลบรารีประเภทหนึ่งที่มีลักษณะพิเศษก็คือ การมุ่งเน้นที่วัตถุประสงค์อย่างใดอย่างหนึ่ง ซึ่งการทำเช่นนี้ส่งให้ระดับการ Reuse สูงขึ้นไปอีก ซึ่งในแนวคิดของคอมโพเนนท์นี้มองว่าระบบประกอบไปด้วยส่วนประกอบต่างๆ เช่นเดียวกันกับวัตถุในโลกความเป็นจริง ตัวอย่างเช่นคอมพิวเตอร์ประกอบด้วยเมาส์ คีย์บอร์ด หรือซีพียู ซึ่งต่างมองว่าเป็นคอมโพเนนท์ต่างๆ ที่มีการทำงานเป็นของตนเอง แต่สามารถนำมาเชื่อมต่อหรือทำงานร่วมกันได้ หากมีส่วนใดส่วนหนึ่งเสียไปก็สามารถที่จะหาคอมโพเนนท์ใหม่มาแทนที่ได้ โดยไม่มีผลกระทบต่อการทำงานกับคอมโพเนนท์อื่นๆ

ในโลกคอมพิวเตอร์คงไม่สามารถเปรียบเทียบตรงๆ กับรถยนต์ได้ในลักษณะนั้น แต่หากว่าลักษณะของคอมโพเนนท์ตัวนี้ไม่ดี ก็สามารถเปลี่ยนเป็นตัวใหม่ได้ ภาษาที่ทำให้คอมโพเนนท์เป็นที่นิยมก็คือ

ภาษา Visual Basic ที่คิดค้น .vbx (ปัจจุบันเป็น .ocx) ถ้าเราไม่ชอบ Combo ที่ให้มากับ VB เราก็ไปสามารถเอาของ Third Party มาใช้แทนได้

บริษัทไมโครซอฟท์ เป็นบริษัทหนึ่งที่มีชื่อเสียงในเรื่องของ การพัฒนาความสามารถของคอมโพเนนท์ ทำให้โปรแกรมของไมโครซอฟท์ทุกตัวใช้แนวคิดของคอมโพเนนท์ ซึ่งเราจะเห็นตัวอย่างได้ชัดเจนจากโปรแกรมชุด Microsoft Office ที่ประกอบไปด้วยโปรแกรมหลายตัว แต่ละตัวสามารถทำงานแยกออกจากกันได้อย่างมีอิสระ แต่เมื่อมีความจำเป็นที่จะต้องทำงานร่วมกันหรือแชร์คอมโพเนนท์ระหว่างกันก็จะสามารถทำได้ด้วยดี ซึ่งไม่เพียงแค่ในส่วนของผู้ใช้งานเท่านั้น แต่รวมถึงในระดับของการพัฒนาด้วย เช่น การพัฒนาแมโครหรือซอฟต์แวร์ Visual Basic for Application ที่สามารถทำงานกับชุด Microsoft Office ได้

Reuse ระดับที่ 5 – ระดับเฟรมเวิร์ก (Framework)

ระดับเฟรมเวิร์กถือได้ว่าเป็นระดับของคลาสไลบรารีอย่างหนึ่งโดยมีลักษณะพิเศษอยู่อย่างหนึ่งคือ สามารถสื่อสารกับโปรแกรมของเราแบบ 2 ทาง โดยต่างจากคลาสไลบรารีหรือคอมโพเนนท์ทั่วไปคือ สามารถสื่อสารกับโปรแกรมได้เพียงทางเดียว เช่นในโปรแกรมที่มีการเรียกใช้เมทธอดของคลาสที่อยู่ในคลาสไลบรารีหรือคอมโพเนนท์ แต่สำหรับเฟรมเวิร์กนั้น ตัวเฟรมเวิร์กจะสามารถเรียกใช้คลาสและเมทธอดของเราเองได้ด้วย ซึ่งคำถามในที่นี้คือเฟรมเวิร์กทำไมจึงต้องมาเรียกใช้โค๊ดของเรา ซึ่งคำตอบในที่นี้ก็คือปัจจุบันการพัฒนาซอฟต์แวร์ได้ก้าวหน้าไปไกลมาก เรามีคลาสไลบรารีที่ได้มีการพัฒนาโดยหลายองค์กร ทำให้คลาสไลบรารีเหล่านี้มีความสมบูรณ์ การพัฒนาซอฟต์แวร์ในปัจจุบันจึงมีเครื่องมือต่าง ๆ ที่ทำให้การพัฒนาซอฟต์แวร์เป็นไปอย่างรวดเร็ว เรียกว่า RAD – Rapid Application Development เรามีเครื่องมือที่มีการทำงานแบบวิชวล (Visual) มากขึ้น คือสามารถพัฒนาโดยอาศัยคอมโพเนนท์ที่มีอยู่แล้ว (โดยอาศัยวิธีการลากและวาง) เฟรมเวิร์กเป็นการจัดการซอฟต์แวร์ในระดับโครงสร้าง โดยการกำหนดส่วนการทำงานของซอฟต์แวร์ตามหน้าที่ที่ชัดเจน ซึ่งการพัฒนาซอฟต์แวร์โดยเฟรมเวิร์ก นักพัฒนาซอฟต์แวร์ไม่จำเป็นต้องกังวลในเรื่องของโครงสร้างหลักของซอฟต์แวร์ การควบคุมการทำงานโดยรวม เช่น การควบคุมการแสดงผลหน้าจอ การเชื่อมต่อกับอุปกรณ์ฮาร์ดแวร์ การดักจับ Event ต่างๆ ซึ่งจะสังเกตได้ว่าเราไม่ได้พัฒนาซอฟต์แวร์เพื่อควบคุมการทำงานของโปรแกรมแต่จะเป็นเฟรมเวิร์กที่ทำหน้าที่นี้แทน

Reuse ระดับที่ 6 – (ระดับแอปพลิเคชันเฟรมเวิร์ก) Application Framework

ในวิศวกรรมซอฟต์แวร์ การพัฒนาซอฟต์แวร์อย่างใดอย่างหนึ่งขึ้นมานั้นเราจะต้องเสียเวลาในการพัฒนาส่วนที่เชื่อมต่อกับผู้ใช้งาน (User Interface) มากกว่าครึ่ง อีกทั้งเราจะต้องพัฒนาลอจิกของโปรแกรมซ้ำๆอยู่ตลอดเวลา ทั้งที่จริงแล้วโปรแกรมที่พัฒนาขึ้นใหม่นั้นอาจจะมีเพียงแค่ 10-20% เท่านั้น ธรรมชาติของโค๊ดโปรแกรมที่เกี่ยวข้องกับลอจิกทางธุรกิจนี้เรียกว่า CRUD (Create Retrieve Update Delete) คืองานที่เกี่ยวกับการจัดการข้อมูล ซึ่งนิยมจัดเก็บในรูปของระบบฐานข้อมูลเชิงสัมพันธ์ ซึ่งโดยปกติแล้วธรรมชาติงานจะมีรูปแบบการจัดการที่ตายตัว เช่นการเพิ่ม การลบ การ

ปรับปรุง หรือการค้นหาข้อมูล ในฐานข้อมูล แต่ทั้งนี้ขึ้นอยู่กับความซับซ้อนของโครงสร้างของฐานข้อมูล ซึ่งจะมีผลทำให้การบำรุงรักษาซอฟต์แวร์สามารถทำได้ง่าย อีกทั้งสามารถรองรับการเปลี่ยนแปลงที่อาจจะเกิดขึ้นได้ในอนาคต ซึ่งเราต้องหาวิธีลดการเกิดผลกระทบต่อโปรแกรมในส่วนอื่นๆ ซึ่งเราเรียกปัญหานี้ว่าปฏิกิริยาลูกโซ่ (Chain Reaction) ซึ่งมักจะเกิดขึ้นเมื่อมีการปรับเปลี่ยนแปลงในโปรแกรมโค๊ดและมีผลกระทบต่อการทำงานในส่วนอื่นๆของโปรแกรม ทำให้เกิดผลลัพธ์การทำงานของโปรแกรมที่ไม่อาจคาดคิดมาก่อน หรือบางครั้งอาจจะก่อให้เกิดข้อผิดพลาดในส่วนการทำงานของโปรแกรมที่เคยทำงานปกติ ซึ่งปัญหาเหล่านี้ก่อปัญหาต่อการพัฒนาโปรแกรม สำหรับกรณีที่การพัฒนาเป็นระบบซอฟต์แวร์ขนาดใหญ่ที่มีนักพัฒนาหลายคน ซึ่งอาจจะแบ่งออกเป็นชุดหรือทีมงาน ซึ่งนักพัฒนาแต่ละคนต่างก็จะมีสไตล์การทำงานที่ไม่เหมือนกัน การออกแบบและการพัฒนาของซอฟต์แวร์อาจจะมีหลากหลายวิธีการ เช่นการกำหนดชื่อของโปรแกรม ชื่อของตัวแปร หรือวิธีการเขียนโค๊ด เป็นต้น ซึ่งจะทำให้ซอฟต์แวร์ขาดความเป็นเอกภาพในที่สุด ดังนั้นแอพลิเคชันเฟรมเวิร์ก (Application Framework) จึงเป็นการกำหนดกรอบการทำงานของทีมพัฒนา เพื่อให้งานเป็นไปตามกรอบหรือรูปแบบที่กำหนดไว้ ซึ่งจะทำให้การควบคุมโครงงาน การติดตามความก้าวหน้า ความเป็นเอกภาพของทีมงานสามารถทำได้ อีกทั้งการแก้ไขปัญหากรณีที่เกิดข้อผิดพลาดในโปรแกรมก็สามารถทำได้ง่าย และหลีกเลี่ยงปัญหาของ Chain Reaction ที่อาจจะเกิดขึ้นอีกด้วย

จากแนวคิดของการนำโค๊ดกลับมาใช้ (Code Reusability) ที่ได้กล่าวมานั้น อันที่จริงแนวคิดนี้ไม่ใช่เป็นแนวคิดใหม่แต่มีมาตั้งแต่ในอดีตเนื่องจากการพัฒนาซอฟต์แวร์แต่เดิมนั้นเริ่มต้นจากการเขียนโค๊ดในลักษณะของ Top-Down ซึ่งพบปัญหาของการตรวจสอบแก้ไขเมื่อพบข้อผิดพลาด หรือการเขียนโค๊ดซ้ำซ้อนกับลักษณะงานที่มีความใกล้เคียงกัน ซึ่งแนวคิดนี้ถูกนำมาใช้กับการจัดกลุ่มโปรแกรมโค๊ดให้อยู่ในรูปของฟังก์ชัน หรือโมดูล และพัฒนามาเป็นไลบรารีต่างๆ ซึ่งเมื่อมีการนำเอาแนวคิดของพัฒนาแบบออบเจ็กต์มาใช้ ก็จะเห็นแนวคิดของการ Reuse ในลักษณะของคอมโพเนนท์ หรือแพคเกจ การนำเอาโปรแกรมโค๊ดกลับมาใช้ใหม่นั้นเป็นมีความชัดเจนมากยิ่งขึ้น ส่งผลต่อการเพิ่มประสิทธิภาพต่อกระบวนการพัฒนาซอฟต์แวร์ในเชิงวิศวกรรมซอฟต์แวร์ (Software Engineering) ได้อีกด้วย เนื่องจากเราสามารถมั่นใจได้ว่าซอฟต์แวร์ที่พัฒนาขึ้นนั้นมีเป็นซอฟต์แวร์แวร์ที่มีประสิทธิภาพและสามารถที่จะพัฒนาได้ทันตามเวลาที่กำหนด ใช้จำนวนบุคลากรที่เหมาะ รวมถึงงบประมาณที่เพียงพอด้วย

1.5 การพัฒนาซอฟต์แวร์เชิงคอมโพเนนท์

การพัฒนาซอฟต์แวร์ที่เป็นลักษณะของคอมโพเนนท์นั้นถือว่าเป็นแนวคิดที่มีความสำคัญมาก เนื่องจากเราสามารถใช้คอมโพเนนท์ในการติดต่อและตอบโต้กับผู้ใช้ และเป็นเครื่องมือที่ช่วยให้ควบคุมการทำงานภายในระบบซอฟต์แวร์สามารถทำได้อย่างง่ายดายและสะดวกรวดเร็ว คอมโพเนนท์คือออบเจ็กต์ต่างๆ ที่นำมาใช้ประกอบในการสร้างแอพพลิเคชัน ซึ่งคอมโพเนนท์ส่วนใหญ่จะถูกจัดเก็บไว้ใน Component Palette โดยแยกเก็บเป็นหมวดหมู่เอาไว้พร้อมนำไปประกอบแอพพลิเคชันที่เราสร้างขึ้น โดยคอมโพเนนท์ จะถูกแบ่งออกเป็น 2 ประเภทดังนี้

- Visual Component เป็นคอมโพเนนท์ที่มีส่วนติดต่อกับผู้ใช้ โดยจะแสดงให้เห็นในขณะที่รันแอพพลิเคชัน เช่น ข้อความหรือปุ่มต่างๆ
- Non-Visual Component เป็นคอมโพเนนท์ที่ไม่แสดงให้เห็นในขณะที่แอพพลิเคชันทำงาน โดยจะเป็นไอคอนหรือคอนโทรลอยู่บนฟอร์มขณะออกแบบ เช่น MainMenu หรือ Popup Menu

แอพพลิเคชันที่ทำงานอยู่ในโลกธุรกิจหลายๆตัวเช่น ในข่ายงานของมัลติมีเดีย การสื่อสาร ซึ่งจำเป็นต้องมีระบบการคำนวณและระบบปฏิบัติการสนับสนุนที่มีประสิทธิภาพ จึงต้องการการสนับสุนนสำหรับการทำงานแบบกระจายและสามารถทำงานได้พร้อมๆ กัน ซึ่งในแง่ของการพัฒนาซอฟต์แวร์เชิงวัตถุนั้นแอพพลิเคชันเหล่านี้มีการทำงานที่ซับซ้อน และอาจจะมีการแยกส่วนการทำงาน หรือทำงานแบบกระจายอยู่ตามเครื่องแม่ข่ายต่างๆ ซึ่งจำเป็นต้องอาศัยความสามารถในการทำงานร่วมกันระหว่างคอมโพเนนท์ของซอฟต์แวร์ สำหรับองค์ประกอบของคอมโพเนนท์ที่สำคัญประกอบด้วย 3 ส่วนหลักดังต่อไปนี้

1. **คุณสมบัติของคอมโพเนนท์ (Component Property)**

คอมโพเนนท์ประกอบด้วยคุณสมบัติต่างๆ ที่ใช้ในการกำหนดลักษณะที่แสดงถึงตัวตนของคอมโพเนนท์นั้นๆ และการทำงานของคอมโพเนนท์ จะมีลักษณะเฉพาะตัวเช่น ชื่อ ความกว้าง ความสูงเป็นต้น ในคอมโพเนนท์ชนิดเดียวกันนั้นจะมีคุณสมบัติที่แสดงถึงความเหมือนหรือความเป็นชนิดเดียวกันแต่ในแต่ละคอมโพเนนท์อาจจะมีค่าของแต่ละคุณสมบัติที่แตกต่างกัน โดยเราสามารถกำหนดค่าของคุณสมบัตินั้น ได้

2. **เหตุการณ์ที่สามารถเกิดขึ้นได้กับคอมโพเนนท์ (Component Event)**

เหตุการณ์หรือการกระทำที่เกิดขึ้นกับแต่ละคอมโพเนนท์ซึ่งอาจเกิดจากผู้ใช้หรือการทำงานภายในแอพพลิเคชันเองก็ได้ เช่น เมื่อคลิกเมาส์ที่ปุ่มจะเกิดอีเวนต์ OnClick กับปุ่มนั้น เป็นต้น

3. **สิ่งที่คอมโพเนนท์สามารถกระทำได้ (Component Method)**

โพรซีเยอร์หรือฟังก์ชันการทำงานของคอมโพเนนท์หรืออาจจะกล่าวได้ว่า เป็นความสามารถในการทำงานอย่างใดอย่างหนึ่งของแต่ละคอมโพเนนท์ เช่น แบบฟอร์มประกอบด้วยเมทธอดสำหรับการแสดง Show() เพื่อใช้สำหรับการเปิด หรือแสดงฟอร์ม และเมทธอดการซ่อน Hide() ที่ใช้สำหรับการซ่อนฟอร์ม เป็นต้น คอมโพเนนท์เหล่านี้ในแต่ละระบบหนึ่งๆ อาจจะมีจำนวนมาก ที่ถูกสร้างขึ้นมาจากโมดูลต่างๆ ซึ่งเราสามารถจัดกลุ่มหรือจำแนกคอมโพเนนท์ออกเป็น Palette ต่างตามประเภทของการใช้งาน เพื่อให้การเรียกใช้หรือการพัฒนาต่อยอดนั้นสามารถทำได้ง่ายขึ้น

บทที่ 2 - ชนิดข้อมูลแบบนามธรรม

โปรแกรมคอมพิวเตอร์เป็นชุดคำสั่งที่มีการทำงานอย่างเป็นลำดับขั้นตอน ซึ่งโปรแกรมที่พัฒนาขึ้นมานั้นมีวัตถุประสงค์ที่ชัดเจน เมื่อเราพัฒนาซอฟต์แวร์ด้วยภาษาใดภาษาหนึ่ง คอมพิวเตอร์จะไม่สามารถเข้าใจภาษาหรือโค๊ดที่เราเขียนโดยอัตโนมัติ โปรแกรมคอมพิวเตอร์จำเป็นต้องมีการแปลงโปรแกรมคอมพิวเตอร์ให้อยู่ในรูปของภาษาเครื่องโดยอาศัยโปรแกรมแปลภาษาได้แก่คอมไพเลอร์ (Compiler) โปรแกรมที่เขียนขึ้นและผ่านการคอมไพล์ และมีการลิงค์ (Link) กับไลบรารีอื่นๆที่จำเป็น ทำให้เป็นโปรแกรมสำเร็จรูปที่มีความสมบูรณ์ เพื่อสามารถนำไปติดตั้งและใช้งานได้

การพัฒนาซอฟต์แวร์ที่ดี ควรมีการวางแผนการวิเคราะห์ ออกแบบ และพัฒนา ตามขั้นตอนของวงจรการพัฒนาซอฟต์แวร์ SDLC (Software Development Life Cycle) และตามหลักวิศวกรรมซอฟต์แวร์ (Software Engineering) เพื่อให้ซอฟต์แวร์ที่พัฒนานั้นมีคุณภาพ ซึ่งทั้งนี้ไม่ว่าจะเป็นการพัฒนาซอฟต์แวร์ด้วยภาษาใดๆก็ตาม ควรมีการกำหนดความต้องการของผู้ใช้ และกำหนดแนวทางการแก้ปัญหาที่ที่ชัดเจน ซึ่งแน่นอนว่าโปรแกรมภาษาแต่ละภาษาจะมีจุดดีจุดเด่นไม่เหมือนกัน ตัวอย่างเช่น ภาษา C เป็นภาษาที่มีความเหมาะสมเชิงโครงสร้าง และสามารถนำไปเขียนในระดับระบบปฏิบัติการ หรือภาษา Perl เป็นภาษาที่เหมาะกับการทำ Text Processing เป็นต้น โครงสร้างไวยากรณ์ภาษา (Syntax) ของภาษาเหล่านี้ก็จะมีความแตกต่างกัน ขึ้นอยู่กับโปรแกรมภาษา (Programming Language ที่เลือกใช้และเครื่องมือ (Intergrated Development Environment: IDE) สำหรับการพัฒนา

การพัฒนาซอฟต์แวร์คือการพยายามที่จะแก้ไขปัญหาโดยอาศัยโปรแกรมคอมพิวเตอร์ ปัญหาดังกล่าวอาจจะเป็นปัญหาในเชิงธุรกิจ เช่น การจัดเก็บสต๊อกสินค้า การบัญชี หรือการเงิน เชิงการศึกษา เชิงการบันเทิง ซึ่งซอฟต์แวร์ได้ถูกนำไปประยุกต์ใช้เพื่อเป็นการแก้ไขปัญหาเหล่านี้ การวิเคราะห์ปัญหาที่เกิดขึ้นในโลกธุรกิจหรือในด้านต่างๆที่กล่าวมา ล้วนแล้วเรียกว่าเป็น Real-World Problem หรือ Real-Life Problem ซึ่งอาจจะเป็นปัญหาเฉพาะในโดเมนนั้นๆ เรียกว่า Domain Problem สิ่งแรกที่นักพัฒนาโปรแกรมจะต้องทำคือพยายามเข้าใจถึงปัญหา รายละเอียด และขอบเขตของปัญหาที่ต้องการแก้ไข เช่น หากต้องพัฒนาโปรแกรมที่เกี่ยวข้องกับระบบคลังสินค้า ก็จำเป็นต้องศึกษาระบบงานที่เกี่ยวข้องกับระบบสินค้า การสั่งซื้อ การขาย หรือแม้กระทั่งขั้นตอนของการจัดซื้อจนกระทั่งถึงการจัดทำสต๊อกสินค้า เพื่อนำมาสร้างเป็นแบบจำลองของโมเดลที่เป็นนามธรรม (Abstraction)

การพัฒนาซอฟต์แวร์ที่มีโครงสร้างแบบ Procedure หรือ Modular Programming นักพัฒนาจะศึกษาความต้องการของผู้ใช้ โดยทำการวิเคราะห์ความต้องการซึ่งมักจะอยู่ในรูปแบบของคำบรรยาย คำสัมภาษณ์ หรือเอกสาร และทำการวิเคราะห์เพื่อมาจัดทำเป็นเอกสารการวิเคราะห์และแผนภาพที่อยู่ในรูปของไดอาแกรมต่างๆ เช่น Data Flow Diagram และ Context Diagram โดยนักพัฒนาซอฟต์แวร์ส่วนใหญ่จะเริ่มจากการทำโมเดลข้อมูล (Data Model) ในลักษณะโครงสร้างของข้อมูล

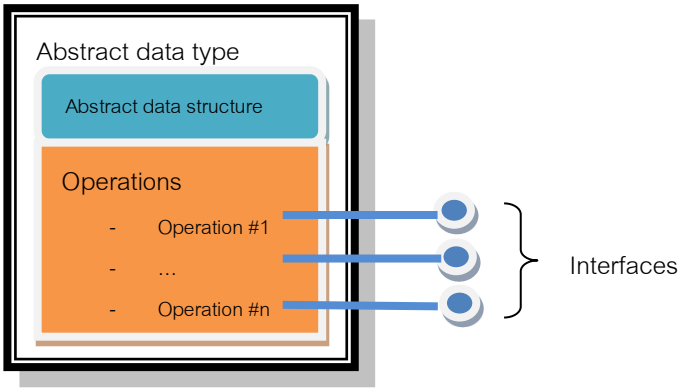
(Data Structure) ตัวแปรข้อมูลและชนิดของข้อมูล (Variable and Data Type) เพื่อใช้สำหรับการจัดเก็บข้อมูล จากนั้นก็จะทำการออกแบบฟังก์ชันที่เกี่ยวข้องเพื่อใช้สำหรับจัดการกับข้อมูลที่จัดเก็บในรูปของฟังก์ชันและโพรซีเยอร์ ซึ่งจะเห็นได้ว่าการออกแบบโมเดลของระบบ ในลักษณะนี้ข้อมูล (Data) และฟังก์ชันที่ใช้จัดการ (Function) จะถูกวิเคราะห์แยกออกจากกัน ซึ่งจะเห็นได้ว่าการแบบจำลองของระบบมักจะอยู่ในรูปของโมเดลข้อมูลเป็นหลัก โดยที่ไม่ได้มองในภาพรวมของระบบ

การวิเคราะห์แบบจำลองข้อมูลนั้นเป็นการสร้างมุมมองของวัตถุในลักษณะที่เป็น Abstraction คือมุมมองต่อวัตถุหนึ่งๆ ว่ามีคุณลักษณะเป็นอย่างไรและมีลักษณะการทำงานเป็นอย่างไร เช่น เมื่อเราพูดถึงคอมพิวเตอร์บางคนอาจจะคิดถึงเครื่องคอมพิวเตอร์ส่วนบุคคลแบบตั้งโต๊ะ และบางคนอาจจะคิดถึงเครื่องโน๊ตบุ๊ค เป็นต้น แต่ในทางนามธรรม จะมองว่าเครื่องคอมพิวเตอร์ประกอบ ด้วยเมาส์ คีย์บอร์ด จอภาพ และซีพียู โดยสามารถทำงานรับข้อมูล ประมวลผล และแสดงผลลัพธ์ออกทางจอภาพ ซึ่งในเบื้องต้นเราอาจจะไม่ได้สนใจว่าภายในเครื่องคอมพิวเตอร์จะมีการทำงานอย่างไร ทั้งนี้จะเห็นได้ว่าเรากำลังสร้างมุมมองที่เป็นนามธรรม หรือมองภาพของออบเจ็กต์ในลักษณะที่เป็นกล่องดำ (Black Box) ซึ่งการมองภาพของวัตถุในลักษณะของนามธรรม นี้มีบทบาทอย่างยิ่งในการวิเคราะห์และออกแบบเชิงวัตถุ ซึ่งจากตัวอย่างจากปัญหานี้มักจะพบในสมาชิกในทีมพัฒนาซอฟต์แวร์ ซึ่งแต่ละคนอาจมีมุมมองต่อระบบที่แตกต่างกัน ซึ่งเราต้องพยายามให้มุมมองของทีมพัฒนาระบบเป็นไปในทางเดียวกัน เพื่อให้ได้ระบบตรงตามที่ต้องการ

สำหรับการพัฒนาโปรแกรมเชิงวัตถุนั้น โจทย์หรือปัญหาที่นำมาสู่การแก้ไขปัญหาด้วยการพัฒนาโปรแกรมขึ้นนั้นจำเป็นจะต้องมีการวิเคราะห์และออกแบบที่ดี โดยมุ่งเน้นการวิเคราะห์และการจัดทำโมเดลของระบบ ซึ่งโมเดลนั้นจะต้องมีความใกล้เคียงกับสภาพปัญหานั้นจริง เป็นโมเดลของออบเจ็กต์ ในแต่ละโปรแกรมจึงประกอบไปด้วยหลายๆ ออบเจ็กต์นำมาประกอบกัน แต่ละออบเจ็กต์ที่ออกแบบก็จะต้องมีลักษณะหน้าที่ที่ต่างกันไป โดยหลักการออกแบบ จะอาศัยหลักการที่ว่า เมื่อมีการเปลี่ยนแปลงใดๆ ในออบเจ็กต์หนึ่งแล้ว การเปลี่ยนแปลงแก้ไขออบเจ็กต์นั้นไม่ควรมีผลกระทบต่อออบเจ็กต์อื่นๆ ซึ่งบางครั้งทำให้เรามองการพัฒนาแบบออบเจ็กต์นี้เป็นลักษณะของโมดูลหรือ คอมโพเนนท์ที่สามารถถอดเข้า - ออกได้ หรือนำไปประกอบกับคอมโพเนนท์อื่นๆ เป็นโปรแกรมใหม่ได้ นั่นก็คือ หลักการของการนำโปรแกรมโค๊ดกลับมาใช้ใหม่ (Code reusability) นั่นเอง

2.1 คุณสมบัติของแบบนามธรรม (Property of Abstract Data Type)

การวิเคราะห์โดเมนปัญหาที่มีอยู่โลกความเป็นจริง เราสามารถแก้ปัญหาโดยนำเอาแนวความคิดเชิงวัตถุมาช่วยได้ โดยการจัดแบ่งกลุ่มปัญหาเป็นกลุ่มย่อยที่เป็นอิสระ เพื่อให้เราสามารถแก้ปัญหาได้อย่างสมบูรณ์ เมื่อเราได้จัดแบ่งปัญหาเสร็จแล้วหลักจากนั้นเราก็สามารถส่งชนิดข้อมูลแบบนามธรรมขึ้นมาได้ โดยกำหนดให้มีคุณสมบัติเป็นต้นแบบดังรูป 2.1 องค์ประกอบประเภทข้อมูลแบบนามธรรม



รูป **2.1** องค์ประกอบประเภทข้อมูลแบบนามธรรม

การวิเคราะห์ชนิดข้อมูลในเชิงวัตถุนั้นต่างจากการวิเคราะห์ข้อมูลในเชิงโครงสร้าง เนื่องจากเป็นการมองภาพของข้อมูลโดยรวมทั้งในส่วนข้อมูลและกิจกรรมที่เกี่ยวข้อง ซึ่งองค์ประกอบที่สำคัญของออบเจ็กต์นั้นประกอบด้วยส่วนสำคัญ 2 ส่วนคือ

1. คุณสมบัติของออบเจ็กต์ (Object Property) – โดยมองว่าออบเจ็กต์ต่างๆที่เกิดขึ้นมีคุณสมบัติเฉพาะของออบเจ็กต์ ซึ่งเราอาจจะเรียกได้ว่าเป็นข้อมูล (Data Property) หรือคุณสมบัติ (Attribute) ที่ใช้สำหรับการอธิบายคุณลักษณะและการเป็นตัวตนของออบเจ็กต์นั้นๆ ดังตัวอย่างเช่น ออบเจ็กต์รถยนต์ จะมีคุณสมบัติคือยี่ห้อ สี ขนาดกำลังเครื่องยนต์ และจำนวนที่นั่ง เป็นต้น
2. พฤติกรรมของออบเจ็กต์ (Object Behavior) – โดยมองว่าออบเจ็กต์จะมีพฤติกรรม หรือออบเจ็กต์สามารถกระทำได้ เช่นกรณีของออบเจ็กต์รถยนต์ ก็จะสามารถเดินหน้า ถอยหลัง หรือหยุด

ซึ่งออบเจ็กต์ที่ถูกสร้างขึ้นมาในลักษณะที่เป็นโมเดลนั้นสามารถที่จะนำมาใช้ในการอ้างอิงในลักษณะที่เป็นประเภทข้อมูล (Data Type) เช่นออบเจ็กต์รถยนต์ Car

```
Object Car
   Attribute: Brand_Name, Color, Engine, NoOfSeat
Behavior: Run(), Stop(), Reverse()
```

โดยประเภทของข้อมูล Car ในภายหลังสามารถนำมาใช้ในการสร้างเป็นชนิดข้อมูลสำหรับรถยนต์ขึ้นมาใหม่ได้ ดังเช่น

```
Car myCar;

myCar.BrandName = "Toyota";
   myCar.Color = "red";
myCar.Engine = 2000;
myCar.NoOfSeat = 4;
```

```
myCar.Start();
myCar.Run();
myCar.Stop();
myCar.Reverse();
```

ซึ่งจะเห็นว่าการสร้างโมเดลข้อมูลที่เป็นออบเจ็กต์นั้นสามารถทำให้วิเคราะห์โครงสร้างของโมเดลได้ใกล้เคียงกับโมเดลในโลกของความเป็นจริง และการวิเคราะห์ในระดับโมเดลนี้ เรายังไม่จำเป็นต้องคำนึงถึงการพัฒนา (Implementation) ว่าจะทำได้อย่างไร เช่นกรณีนี้รถจะวิ่งหรือหยุดได้อย่างไร เพียงแต่เราจะวิเคราะห์ในภาพรวมในเชิงของคุณสมบัติ (Property) และพฤติกรรม (Behaviour) ของออบเจ็กต์

ซึ่งจากแนวคิดนี้จะเห็นว่าเราสามารถแบ่งระดับของความเป็นนามธรรมของโมเดล (Level of Model Abstraction) ได้ดีกว่าการวิเคราะห์ออกแบบโปรแกรมเชิงโครงสร้าง ซึ่งจะทำให้ระดับของการวิเคราะห์มีความเป็นนามธรรมในระดับหนึ่ง แต่ก็สามารถที่มองเห็นภาพที่จะนำไปสู่การพัฒนาเป็นโปรแกรมต่อไป

2.2 การแก้ไขปัญหาในแนวคิดเชิงวัตถุ

ในการพัฒนาโปรแกรมเชิงวัตถุ เรามองแนวทางการแก้ปัญหาในลักษณะของการสร้างแบบจำลองที่มีความสอดคล้องกับปัญหาในโลกของความเป็นจริง (Real-World Problem) และการสร้างโมเดลของออบเจ็กต์จะถูกแสดงในรูปของคลาส (Class) ซึ่งจะกล่าวในบทต่อไป โดยการวิเคราะห์ปัญหาต่างๆนั้นจะมองให้อยู่ในรูปของออบเจ็กต์ โดยที่ระบบหนึ่งๆนั้นสามารถมีได้หลายออบเจ็กต์ และออบเจ็กต์ต่างๆจะมีหน้าที่การทำงานที่แตกต่างกันไป และออบเจ็กต์สามารถสื่อสารระหว่างกันได้ (Object Communication) การมองปัญหาในลักษณะที่เป็นออบเจ็กต์นั้นทำให้การจำลองแบบโมเดลของออบเจ็กต์มีความใกล้เคียงกับปัญหาจริง และสามารถมองปัญหาในเชิงธุรกิจในระดับที่เป็นนามธรรมได้ดีมากขึ้น โดยในระดับนี้ เราไม่จำเป็นต้องคำนึงถึงวิธีการพัฒนาหรือการเขียนโค๊ด หรือแม้กระทั่งการพิจารณาถึงชนิดของข้อมูลที่จะต้องจัดเก็บ

ในปัจจุบันเราจะเห็นว่าแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุนั้น ส่งผลต่อการพัฒนาเครื่องมือและรูปแบบของการพัฒนาซอฟต์แวร์ในปัจจุบันให้มีความก้าวหน้าและมีความทันสมัยมากยิ่งขึ้น โดยเฉพาะอย่างยิ่งในปัจจุบันเราจะเห็นการพัฒนาซอฟต์แวร์ที่อาศัย IDE เป็นลักษณะโปรแกรมแบบวิชวล (Visual Programming) มากยิ่งขึ้น อย่างเช่น Visual Basic, Visual C++, Visual Delphi เป็นต้น ซึ่งเป็นตัวอย่างของการพัฒนาโปรแกรมในลักษณะที่มีการเลือกใช้เครื่องมือ (Tool) เพื่อลากและวาง (Drag and Drop) ลงบนวินโดว์ฟอร์ม และสามารถในผู้พัฒนาเขียนโปรแกรมในลักษณะที่เป็นโปรแกรมโค๊ดที่ทำงานอยู่เบื้องหลัง (Code Behind) ได้ ซึ่งประโยชน์ในแง่มุมหนึ่งคือการที่ผู้พัฒนาสามารถที่จะออกแบบซอฟต์แวร์จากการออกแบบส่วนที่ติดต่อกับผู้ใช้ (User Interface) ได้อย่างรวดเร็ว ซึ่งเป็นการส่งเสริมให้เกิดการพัฒนาซอฟต์แวร์ในลักษณะที่เป็นโพรโตไทป์ (Prototype) ซึ่งเป็นต้นแบบของโปรแกรม เพื่อให้ผู้ใช้งานของระบบสามารถเห็นผลลัพธ์ของโปรแกรมได้อย่างรวดเร็ว

สำหรับเครื่องมือที่ให้ผู้พัฒนาเลือกใช้ในโปรแกรมนั้นมีอยู่มากมาย สำหรับเครื่องมือพื้นฐานอาทิเช่น TextBox, Button, ListBox, RadioButton หรือในระดับที่มีความซับซ้อนสูงเช่น ProgressBar, Calendar, Picture เป็นต้น ซึ่งเครื่องมือต่างๆเหล่านี้แท้จริงแล้วก็คือออบเจ็กต์นั่นเอง นั่นก็คือการพัฒนาซอฟต์แวร์เชิงวัตถุโดยรวมแล้วจะสามารถใช้ประโยชน์จากออบเจ็กต์ที่มีอยู่แล้ว (Existing Object) ซึ่งเป็นหัวใจสำคัญของการพัฒนาโปรแกรมเชิงวัตถุนั่นคือการนำเอาโปรแกรมโค๊ดที่มีอยู่เดิมกลับมาใช้ใหม่ และการพัฒนาออบเจ็กต์ใหม่ที่เราจะต้องวิเคราะห์และออกแบบเองนั้นก็จะเป็นส่วนที่เกี่ยวข้องกับเนื้องานของเราเอง ทั้งนี้จะเห็นได้ว่าการพัฒนาซอฟต์แวร์เชิงวัตถุนั้น เราจะต้องมีการวิเคราะห์ออกแบบให้ดี โดยมีการใช้ประโยชน์จากออบเจ็กต์ที่มีอยู่ ซึ่งอาจจะอยู่ในรูปของเครื่องมือที่มากับภาษาเอง หรือเครื่องมือที่อยู่ใน IDE หรืออาจจะเป็นออบเจ็กต์ที่เป็น Third-Party ออบเจ็กต์ที่เป็นขององค์กรหรือหน่วยงานเอกชนที่เป็นลักษณะของซอฟต์แวร์เปิด (Open Source) หรือในบางครั้งเป็นลักษณะของการขาย (Commercial) ซึ่งต่างๆเหล่านี้ก็จะต้องถูกนำมาใช้ในการออกแบบและพัฒนาร่วมกับออบเจ็กต์ที่มีการพัฒนาขึ้นมาเอง

โดยหลักการวิเคราะห์ออกแบบเชิงวัตถุนี้เอง ทำให้กระบวนการพัฒนาทางวิศวกรรมซอฟต์แวร์เปลี่ยนไปจากเดิม โดยมุ่งเน้นในเรื่องของการวิเคราะห์และออกแบบมากยิ่งขึ้น ซึ่งกระบวนการพัฒนาซอฟต์แวร์ตามวิศวกรรมของออบเจ็กต์นี้เรียกว่า Object-Oriented Software Engineering (OOSE)

2.3 คุณสมบัติทั่วไปของออบเจ็กต์

การแบ่งโปรแกรมออกเป็นออบเจ็กต์ต่างๆ โดยที่แต่ละออบเจ็กต์มีความสมบูรณ์ในตัวเองทำให้เพิ่มความสามารถของซอฟต์แวร์คือ การนำเอาโค๊ดกลับมาใช้อีก หรือสามารถเรียกใช้งานออบเจ็กต์ได้หลายๆ ครั้ง ซึ่งโดยหลักการแล้วออบเจ็กต์จะถูกเรียกขึ้นมาใช้งานในช่วงไหนก็ได้ระหว่างที่โปรแกรมทำงานอยู่ ดังนั้นเราอาจจะมองการทำงานของออบเจ็กต์ว่าเป็นการถูกเรียกใช้งานจากเหตุการณ์ (Event-Driven Programming) ที่เกิดขึ้นในระหว่างการทำงานของโปรแกรม ความหมายในที่นี้ก็คือโปรแกรมหนึ่งอาจจะประกอบไปด้วยหลายออบเจ็กต์ แต่ละออบเจ็กต์ก็อาจจะถูกเรียกขึ้นมาใช้งานได้ขึ้นอยู่กับเหตุการณ์ต่างๆ สิ่งสำคัญอย่างหนึ่งสำหรับการพัฒนาโปรแกรมเชิงวัตถุคือ หลักการนำโปรแกรมกลับมาใช้ได้อีก (Code Reusability) จากในอดีตจะเห็นได้ว่าการพัฒนาซอฟต์แวร์ตัวหนึ่งจะใช้ระยะเวลานาน บางครั้งอาจเป็นหลายๆ เดือนหรืออาจจะเป็นปีก็ได้ การพัฒนาโปรแกรมแบบออบเจ็กต์โอเรียนเท็ดโดยการจัดโครงสร้างของโปรแกรมออกเป็นออบเจ็กต์ต่างๆ นั้น สามารถช่วยลดระยะเวลาในการพัฒนาโปรแกรมโดยการนำเอาออบเจ็กต์ที่มีอยู่แล้ว หรือออบเจ็กต์ที่ถูกสร้างโดยโปรแกรมเมอร์อื่น มาประกอบกันเป็นซอฟต์แวร์ใหม่ๆ ได้อีก ดังนั้นเราจะเห็นได้ว่าปัจจุบันนี้มีการพัฒนาออบเจ็กต์ใหม่ๆ ขึ้นมามากมายโดยอาจจะมาจากนักพัฒนาโปรแกรมทั่วไปหรือจากผู้ผลิตซอฟต์แวร์อื่นๆ การจัดรูปแบบโครงสร้างของโปรแกรมเป็นแบบออบเจ็กต์ (Object-Oriented Structure) เป็นผลทำให้ข้อมูลและฟังก์ชันที่จัดการกับข้อมูลอยู่รวมกันภายในออบเจ็กต์เดียวกัน ทำให้การออกแบบ การพัฒนา การแก้ไขข้อผิดพลาด และการนำออบเจ็กต์ไปใช้งานใหม่สามารถทำได้ง่ายขึ้น

2.3.1 องค์ประกอบที่สำคัญของออบเจ็กต์

ในการพัฒนาซอฟต์แวร์เชิงวัตถุ แนวคิดของการสร้างโมเดลของข้อมูลในรูปของออบเจ็กต์ถือได้ว่าเป็นหัวใจสำคัญ เนื่องจากเป็นแนวคิดที่ต่างจากการวิเคราะห์ในเชิงโครงสร้างและจะต้องมองโจทย์ปัญหาที่กำลังวิเคราะห์ในเชิงของวัตถุ นอกจากนี้องค์ประกอบอื่นๆที่สำคัญที่จะต้องทำความเข้าใจเพื่อให้การวิเคราะห์ออกแบบและการสร้างเป็นโมเดลข้อมูลเป็นไปอย่างมีประสิทธิภาพ ซึ่งองค์ประกอบเหล่านี้ประกอบด้วย

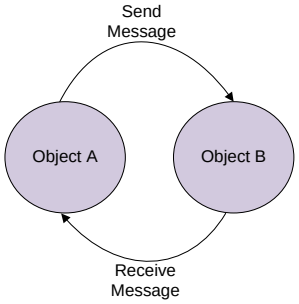
1. การสืบทอด/สืบสกุล (Inheritance) การที่ออบเจ็กต์ต่างๆสามารถถูกสร้างขึ้นมาในลักษณะที่สืบทอดจากอีกออบเจ็กต์หนึ่ง ในลักษณะที่มีความสัมพันธ์แบบออบเจ็กต์แม่ (Parent Object) และออบเจ็กต์ลูก (Child Object) โดยที่ออบเจ็กต์ที่สืบทอดนั้นสามารถสืบทอดเอาคุณสมบัติต่างๆจากออบเจ็กต์แม่มาสู่อ็อบเจ็กต์ลูก ซึ่งแนวคิดของการสืบทอดนี้ก็เป็นแนวคิดหนึ่งของการพัฒนาต่อยอดจากออบเจ็กต์เดิมที่มีคุณสมบัติคล้ายกัน
2. การควบคุมการเข้าถึง (Encapsulation) คำว่า Encapsulate แปลว่าการห่อหุ้ม ซึ่งเป็นลักษณะหนึ่งของการควบคุมการเข้าถึง เพื่อเป็นการป้องกันการเปลี่ยนแปลงคุณสมบัติต่างๆที่อยู่ภายในออบเจ็กต์นั้น โดยจะต้องกระทำผ่านเมทธอดที่กำหนดไว้ให้เท่านั้น ซึ่งการซ่อนข้อมูลภายในออบเจ็กต์ (Data Hiding) ในออบเจ็กต์แต่ละตัวนั้นสามารถกำหนดการเข้าถึงข้อมูล และการเรียกใช้งานของฟังก์ชันได้ โดยหลักการทำงานนั้นจะคล้ายกับการกำหนดขอบเขตของการเข้าถึงตัวแปรในการพัฒนาซอฟต์แวร์แบบโครงสร้าง ที่เป็นแบบ local และ global variable แต่สำหรับในออบเจ็กต์โอเรียนเท็ดนั้นสามารถกำหนดการเข้าถึงได้ในระดับต่างๆ ได้ โดยทั่วไปแล้วจะสามารถควบคุมการเข้าถึงได้ 3 ระดับดังนี้ การกำหนดแบบส่วนตัว (Private) แบบป้องกัน (Protected) และแบบที่ยอมให้มีการเข้าถึงจากภายนอกได้ (Public)
3. การกำหนดผลลัพธ์ของพฤติกรรมที่แตกต่างกันในสภาวะที่ต่างกัน (Polymorphism) ซึ่งเป็นการควบคุมพฤติกรรมของออบเจ็กต์ เพื่อให้ได้ผลลัพธ์กลับต่อผู้ใช้งานอย่างมีประสิทธิภาพ โดยที่ลดความซับซ้อนของการใช้งานออบเจ็กต์ให้แก่ผู้ใช้ ตัวอย่างเช่นการสั่งคำสั่ง CalculateSalary() ซึ่งเป็นคำสั่งที่ใช้สำหรับการคำนวณเงินเดือนของพนักงาน ซึ่งโดยรวมแล้วจะมีความหมายเดียวกันคือการคำนวณเงินเดือนของพนักงาน แต่โดยหลักการทำงานของ Polymorphism ที่มีการทำงานกับแนวคิดเชิงวัตถุนี้ คือโปรแกรมจะคำนวณเงินเดือนของ พนักงานที่เป็นระดับผู้จัดการ หรือเสมียน หรือคนขับรถ ได้โดยจะทำการเรียกใช้ CalculateSalary() ที่ตรงกับออบเจ็กต์ที่ทำงานอยู่ในปัจจุบัน ปกติแล้วจะเกิดขึ้นในขณะที่ออบเจ็กต์ทำงาน (Run-Time)

แนวคิดของการนำเอาโปรแกรมโค๊ดที่มีอยู่กลับมาใช้งานใหม่ (Code Reusability) เป็นแนวคิดที่ทำให้การพัฒนาซอฟต์แวร์สามารถทำได้อย่างรวดเร็ว อีกทั้งลดข้อผิดพลาดของโปรแกรมที่เกิดขึ้นอีก

ด้วย ซึ่งแนวคิดนี้เรียกว่าการสืบทอด (Inheritance) ออบเจ็กต์สามารถถูกสร้างขึ้นมาใหม่จากออบเจ็กต์เดิมได้ โดยอาศัยหลักของการสืบสกุลโดยที่ออบเจ็กต์ใหม่นี้จะ มีคุณสมบัติเหมือนกับออบเจ็กต์เดิม ทั้งส่วนที่เป็นข้อมูลและส่วนที่เป็นฟังก์ชัน แต่ทั้งนี้ออบเจ็กต์ใหม่อาจจะมีข้อมูล ที่เพิ่มขึ้นมาจากออบเจ็กต์เดิมก็ได้ สำหรับแนวคิดของการสืบสกุลนี้ทำให้เราสามารถออกแบบออบเจ็กต์ที่มีลักษณะ การทำงานแบบทั่วไป (General object) และออบเจ็กต์ที่ทำงานเฉพาะ (Specific object) ที่ถูกสืบทอดมาจากออบเจ็กต์ ทั่วไปนั่นเอง จะเห็นได้ว่าการพัฒนาโปรแกรมเชิงวัตถุ ทำให้การพัฒนาโปรแกรมเป็นไปอย่างมีระบบมากยิ่งขึ้น เพราะการแบ่งโปรแกรมออกเป็นออบเจ็กต์นั้นทำให้การวิเคราะห์ ออกแบบ และพัฒนาโปรแกรมสามารถทำได้ง่าย โดยเฉพาะเมื่อเป็นโปรแกรมที่มีขนาดใหญ่ หรือมีการพัฒนากันเป็นแบบทีม นอกเหนือจากการพัฒนาที่เป็นระบบแล้ว การตรวจสอบแก้ไข รวมทั้งการดูแลรักษาโปรแกรมก็สามารถทำได้ ปัจจุบันมีโปรแกรมที่สนับสนุนการพัฒนาแบบเชิงวัตถุนี้มากมาย เช่นภาษา C++, Object Pascal, Delphi, Java, Small talk, และ Object Perl เป็นต้น ซึ่งโปรแกรมเมอร์สามารถเลือกใช้ภาษาตามที่ถนัด หรือให้เหมาะกับประเภทซอฟต์แวร์ที่จะพัฒนา

2.3.3 การสื่อสารระหว่างออบเจ็กต์

การวิเคราะห์และออกแบบโมเดลข้อมูลที่อยู่ในรูปของออบเจ็กต์นั้น ทำให้เรามองเห็นว่าองค์ประกอบของระบบซอฟต์แวร์นั้นจะประกอบด้วยออบเจ็กต์ต่างๆที่มีการทำงานร่วมกัน โดยที่ออบเจ็กต์จะต้องมีการสื่อสารระหว่างกัน ในเชิงของการพัฒนานี้เราจะเรียกว่าออบเจ็กต์มีการส่งค่าและรับค่าระหว่างกัน (Message Passing) โดยที่ออบเจ็กต์หนึ่งจะทำหน้าที่เป็นผู้ส่ง (Sender) และอีกออบเจ็กต์หนึ่งจะทำหน้าที่เป็นผู้รับ (Receiver) ดังรูป 2.2 ออบเจ็กต์ A ทำหน้าที่เป็น sender และออบเจ็กต์ B เป็น receiver ดังนั้นเมื่อ sender ส่ง message ไปเพื่อร้องขอการทำงานบางอย่าง receiver ก็จะส่งผลเป็นค่า return value กลับไปยัง sender



รูป 2.2 การส่งข้อมูลระหว่างออบเจ็กต์

จากตัวอย่างดังกล่าวจะสะท้อนให้เห็นถึงแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุนี้ จะเป็นลักษณะของการบริการ (Services) ของออบเจ็กต์ โดยผู้ใช้งานจะมองว่าออบเจ็กต์นั้นเป็นกล่องดำ (Black Box) ที่มีบริการต่างๆ โดยที่สามารถเรียกใช้งานบริการของออบเจ็กต์โดยผ่านทางอินเตอร์เฟส (Interface) ที่ออบเจ็กต์นั้นมีมาให้

2.4 กระบวนการทำงานเชิงวัตถุ

การแก้ไขปัญหาในเชิงวัตถุจำเป็นต้องมีกระบวนการทำงานอย่างเป็นขั้นเป็นตอน หากวิเคราะห์ในเบื้องต้นนั้นจะไม่ต่างจากการวิเคราะห์และออกแบบซอฟต์แวร์ ดังเช่น กระบวนการ SDLC (Software Development Life Cycle) ที่ประกอบด้วยขั้นตอนการวิเคราะห์ปัญหา การศึกษาความต้องการของผู้ใช้งาน การออกแบบเบื้องต้น การออกแบบโดยละเอียด การพัฒนาซอฟต์แวร์ การทดสอบระบบ และการบำรุงรักษาระบบ ซึ่งขั้นตอนในการวิเคราะห์และออกแบบระบบเป็นเรื่องที่สำคัญ โดยการพัฒนาซอฟต์แวร์ของระบบที่มีขนาดใหญ่ทั่วๆ ไปที่ไม่ใช้หลักการเชิงวัตถุแม้ว่าจะได้รับการออกแบบมาอย่างดี แต่ตัวโปรแกรมก็จะมีความซับซ้อนและยุ่งยาก และส่งผลให้เกิดความผิดพลาดได้ง่าย การออกแบบเชิงวัตถุจะทำให้สามารถเข้าใจระบบได้ดีกว่า การออกแบบระบบได้ง่ายกว่า และแม้ว่าจะออกแบบเชิงวัตถุได้ไม่ดี แต่ผลที่ได้จากการออกแบบนั้นก็ยังได้เปรียบมากกว่าการออกแบบที่ไม่ใช้เชิงวัตถุโดยปรกติแล้วการออกแบบโปรแกรมเชิงวัตถุจะแบ่งออกเป็น 5 ขั้นตอนหลักดังนี้

1. **การรวมรวมและศึกษาความต้องการของระบบ (Requirement Gathering)**

อันดับแรกของการออกแบบ คือ อ่านโจทย์ของระบบที่ต้องการออกแบบและให้ทำความเข้าใจตามหลักความจริงในการทำงานของระบบ ยกตัวอย่างเช่น โทรศัพท์มือถือ ต้องมองได้ว่า โทรศัพท์มีองค์ประกอบภายนอกอะไรบ้าง เช่น โทรออกได้ สามารถรับสายได้ มีปุ่มกด 12 ปุ่มหรือมากกว่า มีสัญญาณไฟติดเมื่อยกหูโทรศัพท์ สามารถตั้งระบบเสียงต่างๆ ได้ สามารถรับสัญญาณเสียงได้ ใช้ได้โดยคน เป็นต้น

2. **การกำหนดคลาสเอนทิตี (Class Entity Identification)**

จากข้อมูลที่วิเคราะห์ความต้องการของผู้ใช้งาน เราอาจจะต้องทำการวิเคราะห์เพื่อหาคลาสออกมาจากข้อมูลนั้นให้ได้ โดยคลาสนั้นเปรียบได้กับคำนาม (Noun) ซึ่งคำนามเป็นได้ทั้ง คน สิ่งของ สถานที่ ฯลฯ เช่น คลาสโทรศัพท์ มนุษย์ หรือคน แต่โดยหลักความจริงแล้วระบบที่มีขนาดใหญ่มักจะมีคลาสมากมายซับซ้อนที่ได้จากข้อมูลดังนั้นควรพิจารณาเฉพาะคลาสที่จำเป็นเท่านั้น คลาสใดที่ซ้ำซ้อนกันให้นำมารวมกันเป็นคลาสเดียว

3. **เก็บข้อมูลพฤติกรรมของคลาส (Class Behavior)**

ในขั้นนี้จะเป็นการแจกแจงรายละเอียดของคลาสที่เลือกมาว่าสามารถทำอะไรได้บ้างในระบบเพื่อให้ทราบถึงความสำคัญของคลาสนั้นๆที่จะส่งผลต่อการทำงานของระบบ และเพื่อให้ทราบถึงหน้าที่การทำงานของคลาสนั้น ซึ่งบางทีอาจบอกได้ว่ามีความสัมพันธ์กับคลาสอื่นอย่างไร เช่น คลาสโทรศัพท์มือถือถูกกดเมื่อมีผู้ใช้ ร้องเรียกเมื่อมีสายเข้า และตอบรับได้เองอัตโนมัติ เป็นต้น

4. **สร้างความสัมพันธ์ระหว่างคลาส (Class Relationship)**

จากการศึกษาพฤติกรรมของคลาส เราจะต้องทำการวิเคราะห์ถึงความสัมพันธ์ที่จะเกิดขึ้นระหว่างคลาสด้วยกัน ซึ่งในบางครั้งอาจจะเรียกได้ว่าเป็นบทบาท (Role) ระหว่างคลาส ซึ่งทั้งนี้ในการสร้างความสัมพันธ์ระหว่าง คลาส จะต้องประกอบด้วยการสื่อสารระหว่างคลาส หรือเรียกว่า Message

Passing ซึ่งทำให้สามารถวิเคราะห์ข้อมูลในส่วนของคุณสมบัติและพฤติกรรมของคลาสในรายละเอียดได้เพิ่มมากขึ้น โดยความสัมพันธ์เหล่านี้สามารถนำไปออกแบบแผนภาพของโมเดล และนำไปพัฒนาเป็นโปรแกรมต่อไป ตัวอย่างของความสัมพันธ์ เช่น ความสัมพันธ์ระหว่างคนกับโทรศัพท์มือถือ โทรศัพท์ส่งเสียงเรียกมายังคน คนรับโทรศัพท์ และคนส่งสัญญาณเสียงไปยังโทรศัพท์ เป็นต้น

5. สร้างโมเดลของคลาส (Class Modeling)

การนำคลาสต่างๆมาสร้างภาพความสัมพันธ์ที่สมบูรณ์ ตามหลักการออกแบบเชิงวัตถุด้วยโมเดลแผนภาพ ซึ่งโมเดลที่ได้รับความนิยมอีกโมเดลหนึ่งคือ Unified Modeling Language หรือเรียกโดยย่อว่า UML ซึ่งใช้เป็นเครื่องมือสื่อสารระหว่างทีมงานพัฒนาหรือผู้ใช้งานได้อย่างชัดเจน ซึ่งทำให้สามารถเข้าใจได้ง่ายทั้งนักวิเคราะห์ นักออกแบบและนักพัฒนาซอฟต์แวร์ ซึ่ง UML ประกอบด้วยแผนภาพไดอาแกรมหลัก 13 ไดอาแกรม โดยแบ่งออกเป็นกลุ่มตามวัตถุประสงค์ของไดอาแกรมดังต่อไปนี้

- กลุ่มที่ 1 แผนภาพไดอาแกรมที่แสดงโครงสร้าง
- กลุ่มที่ 2 แผนภาพไดอาแกรมที่แสดงถึงพฤติกรรม
- กลุ่มที่ 3 แผนภาพไดอาแกรมที่แสดงถึงการโต้ตอบ

UML Diagram	คำอธิบาย
กลุ่มที่ 1 แผนภาพไดอาแกรมที่แสดงโครงสร้าง	
Class Diagram	แสดงคลาสต่างๆที่มีอยู่ในระบบ รวมถึงความสัมพันธ์และดีกรีความสัมพันธ์ เพื่อให้เห็นลักษณะของการทำงานร่วมกันระหว่างคลาส
Component Diagram	แสดงให้เห็นถึงความเชื่อมโยงการทำงานของคอมโพเนนท์ต่างๆในระบบ ซึ่งอาจจะแสดงเห็นถึงองค์ประกอบของคอมโพเนนท์หลัก และคอมโพเนนท์ย่อย โดยการเรียกใช้งานคอมโพเนนท์สามารถเรียกผ่านทางอินเตอร์เฟส
Composite Structure Diagram	แสดงถึงความเชื่อมโยงระหว่างองค์กอบภายในของคลาส กับคลาสอื่นๆ หรือคอมโพเนนท์อื่นๆ โดยมีประโยชน์เพื่อแสดงเห็นถึงความสัมพันธ์ของการเรียกใช้งานในขณะที่โปรแกรมทำงาน (Run-Time)
Deployment Diagram	แสดงถึงการนำเอาซอฟต์แวร์ไปติดตั้งและใช้งาน ในเชิงของโครงสร้างสถาปัตยกรรม เช่น สถาปัตยกรรมของเครื่องแม่ข่าย การทำงานร่วมกันระหว่างซอฟต์แวร์ – เครื่องแม่ข่ายของ Application Server – หรือเครื่องแม่ข่ายสำหรับฐานข้อมูล Database Server หรือแม้กระทั่งลักษณะของการเชื่อมต่อโดยผ่านเครือข่ายเน็ตเวิร์กโดยอาศัยโพรโตคอล TCP/IP
Object Diagram	แสดงถึงความสัมพันธ์คล้ายกับโครงสร้างของคลาสไดอะแกรม แต่จะแสดงในรายละเอียดในระดับของออบเจ็กต์ ที่มีการสร้างในระดับของออบเจ็กต์อินสแตนซ์ (Object Instance)
Package Diagram	แสดงถึงการจัดแบ่งคลาสในลักษณะที่เป็นความสัมพันธ์แบบลำดับชั้น

	(Class Hierarchy) และการจัดกลุ่มของคลาสที่มีบทบาทหน้าที่คล้ายกันในลักษณะที่เป็นแพคเกจ
กลุ่มที่ 2 แผนภาพไดอาแกรมที่แสดงถึงพฤติกรรม	
Use Case Diagram	แสดงความสัมพันธ์ระหว่างผู้ใช้ (Actor) กับระบบ (System) ในมุมมองของผู้ใช้งานว่าระบบสามารถที่จะบริการ (Service) อะไรให้แก่ผู้ใช้ ซึ่ง Class Diagram จะเป็นไดอาแกรมที่แสดงให้เห็นถึงขอบเขตการทำงานของระบบโดยรวม หรือการติดต่อสื่อสารระหว่างระบบด้วยกัน
Activity Diagram	แสดงให้เห็นถึงขั้นตอนการทำงาน และตรรกะของกิจกรรมที่มีความซับซ้อนและต้องอาศัยการอธิบายเพื่อให้เห็นลำดับเหตุการณ์การทำงาน เช่น ขั้นตอนการสมัคร หรือขั้นตอนการพิจารณาสินเชื่อ
State Diagram	แสดงให้เห็นถึงสถานะของออบเจ็กต์ เมื่อมีการดำเนินงานกิจกรรมใดๆ เช่น การเรียกเมดธอด Break() ส่งผลให้สถานะของออบเจ็กต์รถยนต์มีสถานะเป็นหยุดนิ่ง
กลุ่มที่ 3 แผนภาพไดอาแกรมที่แสดงถึงการโต้ตอบ	
Sequence Diagram	แสดงความสัมพันธ์ของการเรียกใช้งานเมทธอดที่อยู่ภายในคลาสต่างๆ โดยมีลำดับขั้นตอนและเวลาดำเนินการ และการตอบสนอง สำหรับแต่ละกิจกรรมที่มีการดำเนินการ
Communication Diagram	แสดงความสัมพันธ์ระหว่างออบเจ็กต์ ที่แสดงถึงลำดับเหตุการณ์และการทำงานของออบเจ็กต์เป็นขั้นเป็นตอน โดยที่ออบเจ็กต์ที่เลือกมาจัดทำเป็นไดอาแกรมนี้อาจจะมาจากคลาส ซีเควนซ์ และยูสเคสไดอาแกรม
Interaction Overview Diagram	แสดงให้เห็นถึงขั้นตอนการทำงานในระดับกิจกรรม ซึ่งคล้ายกับ Activity Diagram แต่ในแต่ละกิจกรรมที่แสดงนั้นจะแสดงรายละเอียดที่เป็นการขยายความ โดยแสดงเป็น Sequence หรือ Communication
UML Timing Diagram	แสดงถึงพฤติกรรมการทำงานของออบเจ็กต์ตามช่วงเวลาต่างๆ ซึ่งไดอาแกรมนี้มีความคล้ายกับ Sequence Diagram

2.5 การศึกษาความต้องการของผู้ใช้ (User Requirement)

คุณลักษณะของโปรแกรมที่ดีคือโปรแกรมที่พัฒนาขึ้นมาตรงตามความต้องการของผู้ใช้งาน มีรูปแบบการทำงานที่ชัดเจนตรงตามวัตถุประสงค์ที่ตั้งไว้ มีส่วนเชื่อมต่อกับผู้ใช้งานที่ง่าย สะดวกต่อการเข้าถึงและเรียกใช้ (User Friendly Environment) หรือซอฟต์แวร์ที่ดีจะต้องมีการสื่อสาร (User Interaction) ที่ดีเพื่อให้ผู้ใช้งานรู้ว่าขณะนี้ตัวเองกำลังทำอะไร ซอฟต์แวร์กำลังประมวลผลอะไร และได้ผลลัพธ์อะไรกลับคืนมา ซึ่งนั่นเป็นมุมมองในฝั่งผู้ใช้งานเองที่อาจจะมองซอฟต์แวร์เป็นเหมือนกับกล่อง

ดำ (Blackbox) ที่ผู้ใช้งานไม่จำเป็นต้องสนใจในรายละเอียดของการทำงาน แต่สนใจผลผลิตที่ได้จากการทำงานของซอฟต์แวร์ แต่หากในมุมมองของผู้พัฒนาแล้วซอฟต์แวร์จะต้องมีขอบเขตความต้องการของผู้ใช้งานที่ชัดเจน มีความเข้าใจโดเมนของปัญหา มีแนวทางและเข้าใจขั้นตอนของการแก้ปัญหา ซึ่งการที่จะทำให้นักพัฒนาโปรแกรมรู้และเข้าใจสิ่งที่ผู้ใช้ต้องการ ในการบวนการพัฒนาซอฟต์แวร์จึงมีขั้นตอนของการศึกษาความต้องการของผู้ใช้งาน (User Requirement Gathering) เกิดขึ้น ซึ่งเป็นขั้นตอนหนึ่งที่สำคัญมาก เนื่องจากหากนักพัฒนาโปรแกรมไม่เข้าใจถึงปัญหาความต้องการของผู้ใช้อย่างแท้จริงแล้ว อาจจะทำให้การสื่อความหมายของซอฟต์แวร์ที่ต้องการพัฒนานั้นผิดไป ทำให้ซอฟต์แวร์ที่พัฒนาไม่ได้ตรงกับความต้องการ ทำให้เสียทั้งเวลา งบประมาณ และทรัพยากรบุคลากรในที่สุด

การศึกษาความต้องการของผู้ใช้งานมีวิธีการอยู่หลายวิธีด้วยกันขึ้นอยู่กับวิธี การที่นักพัฒนาซอฟต์แวร์จะเลือกใช้ แต่ขั้นตอนที่สำคัญมีอยู่ 4 วิธีคือ

1. การสัมภาษณ์ (Interview)

การสัมภาษณ์ถือได้ว่าเป็นวิธีการที่ได้รับความนิยมวิธีหนึ่ง เนื่องจากได้สื่อสารกับผู้ใช้โดยตรงในลักษณะ Face-to-Face และเป็นการได้รับฟังปัญหาจากผู้ใช้โดยตรง ซึ่งในระหว่างการสัมภาษณ์อาจจะมีการจดบันทึกเป็นเอกสาร หรือการอัดเสียงการสัมภาษณ์เพื่อนำมาฟังหรือวิเคราะห์ในภายหลัง การขั้นตอนนี้นักพัฒนาอาจจะเลือกสัมภาษณ์ผู้ใช้งานที่เกี่ยวข้องหรือเจ้าหน้าที่ที่เกี่ยวข้องกับงานนั้นๆ หรือเฉพาะระดับผู้จัดการ อย่างไรก็ตามการสัมภาษณ์อาจจะมีข้อเสียตรงที่การสัมภาษณ์แต่ละครั้งนั้นใช้เวลาค่อนข้างมาก ซึ่งจำเป็นต้องมี Outline ของเนื้อหาและกำหนดเวลาในการสัมภาษณ์ให้ดี การเลือกผู้ที่จะสัมภาษณ์ก็เป็นสิ่งจำเป็นเนื่องจาก ผู้ให้สัมภาษณ์แต่ละคนอาจจะมีความรู้ความเข้าใจของปัญหาที่แตกต่างกัน หรืออาจจะมีมุมมองของปัญหาที่ขัดแย้งกัน ซึ่งจะต้องทำการประมวลสิ่งที่ได้รับจากการสัมภาษณ์และนำไปสรุปให้กับผู้ใช้งานอีกครั้งหนึ่ง

2. การทำแบบสอบถาม (Questionnaire)

ในการพัฒนาซอฟต์แวร์จากความต้องการของผู้ใช้งาน ซึ่งบางครั้งผู้ใช้งานเองอาจจะยังไม่เข้าใจถึงปัญหาที่แท้จริงว่าตนเองต้องการอะไร หรือองค์กรต้องการอะไร การแก้ไขปัญหาขององค์กรโดยวิธีการพัฒนาซอฟต์แวร์จะเป็นคำตอบสุดท้ายที่จะนำมาซึ่งการแก้ไขปัญหาหรือไม่ ซึ่งในกรณีแบบนี้การเลือกทำแบบสอบถามจากผู้ใช้งาน ที่อาจจะมีหลายระดับ หลายกลุ่มคนเข้ามาเกี่ยวข้อง อาจจะเป็นทางเลือกหนึ่งในการศึกษาความต้องการของผู้ใช้งาน เพราะแบบสอบถามอาจจะเป็นคำถามปลายปิดเพื่อให้ผู้ใช้งานยืนยันในสิ่งที่ต้องการหรือการยอมรับในสิ่งที่มีอยู่ หรือในลักษณะของปัญหาแบบปลายเปิดเพื่อให้ผู้ใช้สามารถแสดงความคิดเห็นของตนเอง เช่นอธิบายคุณลักษณะของซอฟต์แวร์ที่ต้องการอยากจะเห็นหรืออยากได้ อย่างไรก็ตามการทำแบบสอบถามนั้นก็ยังมีข้อจำกัดในเรื่องของการสื่อสารที่อาจจะต้องมีการตีความในสิ่งที่ยังคลุมเครือและสรุปเป็นประเด็นเพื่อใช้กำหนดเป็นความต้องการที่ชัดเจน

3. การศึกษาจากเอกสารที่มีอยู่ (Existing Documents)

การศึกษาความต้องการของผู้ใช้งานจากการสัมภาษณ์หรือแม้การทำแบบสอบถามเองอาจจะทำให้ภาพความต้องการโดยรวม แต่อาจจะยังไม่เพียงพอสำหรับการนำไปออกแบบเชิง

รายละเอียด ซึ่งวิธีการศึกษาจากเอกสารที่มีเป็นวิธีหนึ่งที่จะทำให้นักพัฒนาเห็นภาพของตัวอย่างของการใช้งานจริงในองค์กร หรือรูปแบบการจัดวางของเอกสาร รายงาน เช่นตัวอย่างของเอกสาร รายงาน ตัวอย่างใบเสร็จ หรือเอกสารอาจจะเป็นรายงานประจำปีขององค์กรเป็นต้น ซึ่งกรณีที่หน่วยงานมีเอกสารที่เคยจัดทำไว้แล้วทั้งที่เป็นอิเล็คทรอนิกส์หรือเป็นกระดาษ และที่สามารถเปิดเผยให้กับนักพัฒนาได้ ถือว่าเป็นเรื่องที่ดีเพราะการแก้ไขปัญหานั้นจะอยู่บนพื้นฐานของปัญหาที่แท้จริง แต่กรณีที่หน่วยงานไม่เคยมีเอกสารหรือระบบงานนั้นๆมาก่อนเลย ก็จะต้องมีการหารือกับนักพัฒนาเพื่อกำหนดรูปแบบและเนื้อหาของเอกสารหรือรายงานนั้นขึ้นมาใหม่

4. การศึกษาโดยวิธีการวิจัย (Research)

ในบางครั้งความต้องการของผู้ใช้งานอาจจะเป็นสิ่งใหม่หรือเป็นสิ่งที่ไม่เคยเกิดขึ้นมาก่อน หรือเป็นความต้องการที่ผู้ใช้งานวาดฝันอยากจะเห็น ซึ่งลักษณะของการศึกษาความต้องการของผู้ใช้งานโดยวิธีการทำวิจัยนั้นอาจเป็นการทดลองโดยจัดทำเป็นต้นแบบ (Prototype) หรือมีการสร้างสถานการณ์จำลอง (Simulation) ในสภาพแวดล้อมที่ใกล้เคียงกับความเป็นจริง ซึ่งลักษณะของการทำวิจัยนั้นอาจจะเป็นการทดลองทำต้นแบบของซอฟต์แวร์ โดยการแสดงเป็นหน้าจอตัวอย่าง หรือต้นแบบในลักษณะของระบบที่ต้องการวัดประสิทธิภาพการทำงานของระบบเป็นต้น การทำวิจัยนี้เป็นวิธีที่เหมาะสมกับสิ่งที่ต้องการทดลองหรือพิสูจน์ถึงความเป็นได้ของงานโดยที่ไม่จำเป็นต้องลงทุนพัฒนาหรือรอให้ทั้งระบบมีการพัฒนาเสร็จสิ้นก่อน

ดังนั้นเราพอจะสรุปได้ว่าความต้องการของผู้ใช้งานนั้นเป็นสิ่งสำคัญ ซึ่งจะนำไปสู่การพัฒนาซอฟต์แวร์อย่างมีประสิทธิภาพ ซอฟต์แวร์ที่พัฒนานั้นจะต้องมีคุณลักษณะตรงตามความต้องการของผู้ใช้ มีวัตถุประสงค์หรือเป้าหมายของการทำงานที่ชัดเจน และในส่วนของผู้พัฒนาเองจะต้องเข้าใจโดเมนปัญหา ทำให้การวิเคราะห์ เป็นระบบที่สามารถพัฒนาและนำไปใช้งานได้จริง หรือท้ายสุดแล้วผู้พัฒนาจะต้องคำนึงถึงว่าการใช้งานซอฟต์แวร์ อาจจะมีอายุการใช้งานที่ยาวนาน ซึ่งอาจจะเป็นปี หรือหลายปี ซึ่งในระหว่างนั้นจะต้องมีการปรับแก้ไข หรือปรับปรุงเมื่อผู้ใช้งานพบปัญหา ดังนั้นการบำรุงดูแลรักษาซอฟต์แวร์ให้สามารถทำงานได้อย่างต่อเนื่องมีความยืดหยุ่นที่สามารถรองรับการเปลี่ยนแปลงอาจจะเกิดขึ้นในอนาคต จึงเป็นเรื่องที่สำคัญ เพื่อให้ซอฟต์แวร์ที่พัฒนานั้น เป็นซอฟต์ที่มีคุณภาพ

บทที่ 3 - คลาสและออบเจ็กต์

สิ่งสำคัญในการพัฒนาโปรแกรมเชิงวัตถุ คือ การวิเคราะห์และออกแบบโครงสร้างของโปรแกรมให้อยู่ในรูปของออบเจ็กต์ต่างๆ และรวมไปถึงความสามารถในการทำงานร่วมกันระหว่างออบเจ็กต์ด้วย ในบทนี้เราจะทำความรู้จักกับคลาสซึ่งเปรียบเสมือนเป็นโครงสร้างของออบเจ็กต์ในมุมมองของการพัฒนาโปรแกรม และรู้จักวิธีการสร้างออบเจ็กต์ขึ้นมาใหม่ด้วยวิธีการสร้าง Instance ของคลาส การสื่อสารระหว่างออบเจ็กต์ด้วยวิธีการของ Message Passing รวมไปถึงการควบคุมการมองเห็นของคุณสมบัติของออบเจ็กต์ (Attributes) และเมทธอด (Methods) ภายในออบเจ็กต์เรียกว่า Polymorphism หรือ Information Hiding

วัตถุ (Objects) คือ สิ่งใดๆ ก็ตามซึ่งมีคุณลักษณะ (Attribute) บ่งบอกถึงความเป็นตัวของตัวเองในขณะนั้น และสามารถแสดงพฤติกรรม (Behavior) ของตัวเองออกมาได้ ในชีวิตประจำวันเมื่อมองดูรอบตัว เราก็จะพบเห็นวัตถุต่างๆ มากมาย ไม่ว่าจะเป็นวัตถุที่เราสามารถมองเห็นได้และ จับต้องได้ (Tangible Objects) เช่น โต๊ะ รถยนต์ คอมพิวเตอร์ หรือแม้แต่ตัวเราเอง ในขณะเดียวกัน ในโลกของเราก็ยังมีวัตถุที่มีอยู่จริง แต่ไม่สามารถจับต้องได้ (Intangible Objects) เช่น กฎหมาย (ในที่นี้หมายถึงกฎข้อบังคับที่รัฐบัญญัติขึ้น ไม่ใช่ตัวเล่มรัฐธรรมนูญ ซึ่งถือเป็น Tangible Object) เวลา หรือความรู้ต่างๆ เป็นต้น กิจกรรมต่างๆ ที่เกิดขึ้นในชีวิตเรานั้น ล้วนแล้วแต่เกิดจากการมีความสัมพันธ์ (Relationship) และการมีปฏิสัมพันธ์ (Interaction) ระหว่างออบเจ็กต์ 2 ตัวขึ้นไป

การเขียนโปรแกรมเชิงวัตถุจะทำการมองปัญหาของโปรแกรมเป็นแบบออบเจ็กต์ โดยมองว่าโปรแกรมหนึ่งจะประกอบไปด้วยหลายๆ ออบเจ็กต์เข้าด้วยกัน โดยในหลักการก็คือ ออบเจ็กต์แต่ละตัวจะเป็นเอกเทศ (Modularity) มีความสมบูรณ์ในตัวเองและทำหน้าที่ต่างกันไป ออบเจ็กต์แต่ละตัวประกอบไปด้วยส่วนที่เป็นชุดข้อมูล (Data member) หรือบางครั้งเรียกว่าเป็น attributes และเมทธอด (Methods) ที่ใช้ในการจัดการกับข้อมูล บางครั้งเมทธอดเหล่านี้อาจมองว่าเป็นพฤติกรรม (Object Behavior) ของออบเจ็กต์การจัดรูปแบบโครงสร้างแบบออบเจ็กต์นี้ (Object-Oriented Structure) ช่วยทำให้การออกแบบ การพัฒนา การแก้ไขข้อผิดพลาด และการนำออบเจ็กต์ไปใช้งานใหม่สามารถทำได้ง่ายขึ้น ตัวอย่างของการวิเคราะห์โครงสร้างของออบเจ็กต์มีดังนี้

Object Cat

Attributes: name, color

Behavior: walk, run, sleep, eat

Object Car

Attributes: make, model, year, color, engine size, speed

Behavior: move, stop, brake, change gear, start up engine, accelerate

Object Student

Attributes: id, name, last name, phone number, address, grade, GPA

Behavior: take course, take exam, study, show grade, show GPA, display

จากตัวอย่างจะเห็นว่าโครงสร้างของออบเจ็กต์จะถูกวิเคราะห์เป็นสองส่วนสำคัญเสมอ สำหรับคุณสมบัติหรือบางครั้งเรียกว่าเป็นคุณสมบัติ Property ของออบเจ็กต์และสำหรับ Behavior ได้แก่ การกระทำ หรือเป็นการแสดงให้เห็นว่าออบเจ็กต์สามารถกระทำอะไรได้ ซึ่งอาจจะเรียกว่าเป็น Action หรือเหตุการณ์ (Event) ที่สามารถเกิดขึ้นกับออบเจ็กต์ได้

หัวใจสำคัญของการพัฒนาโปรแกรมเชิงวัตถุ ก็คือ ออบเจ็กต์ “Object” หรือวัตถุ ซึ่งหมายถึง Entity หรือสิ่งที่มีตัวตน และนำไปใช้ในการประมวลผลข้อมูล โดยในออบเจ็กต์จะต้องมีข้อมูล (Attribute) ที่ใช้ในการอธิบายตนเองและการกระทำต่าง ๆ (Behavior) ที่ออบเจ็กต์สามารถกระทำได้ ออบเจ็กต์สามารถเป็นได้ทั้งสิ่งที่มองเห็นได้ เช่น รถ เรือ สินค้า หรือเป็นเหตุการณ์ต่าง ๆ เช่น การขายสินค้า การประชุมเชิงวิชาการ การสอบแข่งขันเข้ามหาวิทยาลัย หรือแม้กระทั่งเป็นตัวโปรแกรมที่ใช้เขียนขึ้นมา ก็ถือว่าเป็นออบเจ็กต์เช่นกัน หรือร้านขายยาร้านหนึ่ง มีโปรแกรมเกี่ยวกับสินค้าคงคลังสำหรับขายสินค้าในร้านตัวอย่าง เช่น ยา 1 ชนิด ถือว่าเป็น 1 ออบเจ็กต์ภายในยาแต่ละชนิดก็จะมีชื่อยา จำนวนที่เหลือ ราคาขาย เลขที่ยา จุดสั่งซื้อและรายละเอียดอื่นๆ เหล่านี้ถือว่าเป็น “Attribute” หรือข้อมูลที่ใช้อธิบายยาแต่ละชนิด นอกจากนั้นตัวยาแต่ละชนิดก็จะมีขั้นตอนการทำงานเพื่อจัดเก็บข้อมูลของตน เช่น การแก้ไขข้อมูล การเพิ่มข้อมูล การประเมินว่าจะถึงจุดสั่งซื้อสินค้าใหม่หรือยัง ขั้นตอนในการทำงานดังกล่าวเรียกว่า “Behavior” ซึ่งกล่าวได้ว่าออบเจ็กต์จะต้องประกอบด้วยคุณสมบัติและพฤติกรรมนั่นเอง

3.1 สถานะของออบเจ็กต์ (Object States)

การที่ออบเจ็กต์มีคุณสมบัติเป็นของตัวเองพร้อมทั้งการกระทำที่สามารถทำหรือเกิดขึ้นได้ เมื่อมีการเปลี่ยนแปลงใดๆ เกิดขึ้นก็ย่อมมีผลกระทบต่อตัวออบเจ็กต์เอง หรือเรียกได้ว่าสถานะของออบเจ็กต์ได้เปลี่ยนไป ดังเช่นในตัวอย่างของออบเจ็กต์รถ ซึ่งเมื่อออบเจ็กต์ถูกสร้างขึ้นมาใหม่จะมีสถานะเป็นหยุดนิ่ง และเมื่อมีการเร่งเครื่องยนต์หรือมีการเบรก ก็จะมีผลกระทบกับสถานะของออบเจ็กต์ซึ่งอาจจะใช้วิธีตรวจสอบจากความเร็วของรถเป็นหลักว่า เช่น รถในขณะนี้มีสถานะเป็นกำลังเคลื่อนที่หรือกำลังหยุดนิ่ง สถานะของออบเจ็กต์อาจจะแบ่งออกเป็นสถานะที่ไม่ซับซ้อน (simple state) เช่น รถหยุดนิ่ง ไฟสัญญาณเป็นสีเขียว วีดิโอเทปกำลังเล่นเทป หรือตู้เย็นมีอุณหภูมิเป็นศูนย์ และในบางครั้งสถานะของออบเจ็กต์ก็อาจจะซับซ้อนอย่างเช่น ในตัวอย่างของออบเจ็กต์ตัวหมากรุกที่ต้องมีสถานะได้เปรียบหรือเสียเปรียบและต้องขึ้นอยู่กับออบเจ็กต์ของตัวหมากรุกของฝั่งตรงข้าม ทั้งนี้จะเห็นได้ว่าคุณสมบัติของออบเจ็กต์จะเป็นตัวกำหนดสถานะของออบเจ็กต์นั้นๆ หากว่าเป็นออบเจ็กต์ที่มีสถานะที่ซับซ้อนก็อาจจะมีจำนวนของคุณสมบัติที่เพิ่มขึ้น และการที่มีการเปลี่ยนแปลงค่าในคุณสมบัติหนึ่งก็อาจส่งผลให้มีการเปลี่ยนแปลงในคุณสมบัติที่เหลือด้วย

ดังนั้นวัตถุประสงค์ที่สำคัญอย่างหนึ่งของการออกแบบออบเจ็กต์ก็คือ พยายามทำให้ออบเจ็กต์มีโครงสร้างที่สามารถจัดการได้ง่าย เพื่อลดการใช้คุณสมบัติและเมทธอดที่ซับซ้อน อย่างไรก็ตามในกรณีที่ไม่สามารถหลีกเลี่ยงการสร้างออบเจ็กต์ที่มีความซับซ้อน ก็อาจจะใช้วิธีการแบ่งออบเจ็กต์ให้เป็นออบเจ็กต์ย่อยหลายๆ ออบเจ็กต์เป็นต้น การแบ่งเป็นออบเจ็กต์ย่อยหลายๆออบเจ็กต์นี้มีข้อดี คือ สามารถจัดการกับออบเจ็กต์แต่ละส่วนได้ง่ายขึ้นทั้งการดูแลรักษา (Code maintenance) และการตรวจสอบข้อผิดพลาด (Error debugging) ข้อเสียก็คือ จะต้องมีการสร้างความสัมพันธ์ระหว่างออบเจ็กต์ (Interrelationships) ซึ่งอาจจะทำให้โปรแกรมดูซับซ้อนมากขึ้น

3.2 โครงสร้างของคลาส (Class Structure)

แนวการพัฒนาโปรแกรม การสร้างออบเจ็กต์จะอาศัยการออกแบบชนิดของข้อมูลที่เรียกว่า “คลาส” (Class) ซึ่งคลาสในความหมายของโปรแกรมภาษาก็คือ ออบเจ็กต์นั่นเอง สำหรับโครงสร้างโดยพื้นฐานของคลาสประกอบไปด้วยสมาชิกของคลาส (Data member) และเมทธอด (Methods)

```
public class <class_name> {
      // Data Members
     <data_type> data_member1;
     <data_type> data_member2;
     <data_type> data_member3;

  // Function Members
     void Method1() {
                ...
     }

      // Function Members
     void Method2() {
                ...
     }
}
```

การประกาศสมาชิกของคลาส สามารถอยู่ในตำแหน่งใดๆ ก็ได้ภายในคลาส แต่โดยปรกติจะนิยมประกาศให้อยู่ในกลุ่มเดียวกันในตอนเริ่มต้นหรือตอนท้ายของคลาส สำหรับแนวคิดของการเขียนโปรแกรมเชิงวัตถุนั้นเหมาะกับการนำเอาปัญหาที่เกิดขึ้นจริงมาจำลองการทำงาน โดยมองว่าปัญหาต่างๆ นั้นอยู่ในรูปของวัตถุ ตัวอย่างเช่น ถ้าต้องการเขียนโปรแกรมเพื่อแสดงการทำงานของเครื่องวิดีโอ เราอาจจะมองว่าเครื่องวิดีโอหนึ่งเครื่องนี้เป็นออบเจ็กต์หนึ่งออบเจ็กต์ สำหรับการวิเคราะห์หาองค์ประกอบของออบเจ็กต์นั้นเรามองว่าเครื่องวิดีโอประกอบไปด้วยปุ่มและไฟที่ทำหน้าที่ในการแสดงผลการทำงานเช่นเมื่อเครื่องทำงาน หรือเมื่อเล่นเทป คุณสมบัติเหล่านี้ก็คือข้อมูลของออบเจ็กต์วิดีโอนั่นเอง สิ่งสำคัญอีกอย่างหนึ่งของออบเจ็กต์ได้แก่ ฟังก์ชันการทำงานของวิดีโอ เช่น การใส่เทปวิดีโอ (InsertTape) การเล่นวิดีโอ (Play) การหยุดชั่วคราว (Pause) การหยุด (Stop) และการนำเอา

วิดีโอออกจากเครื่องเล่น (Eject) จากการวิเคราะห์โครงสร้างของออบเจ็กต์ วิดีโอนี้ เราสามารถออกแบบโครงสร้างของออบเจ็กต์วิดีโอได้ดังนี้

```
Object Video:
    Data Member:  Light, Tape Status
    Methods:      Power On, Play, Stop, Rewind, Forward, Pause
```

จะเห็นได้ว่าออบเจ็กต์วิดีโอนี้มีคุณสมบัติคือ ไฟที่แสดงสถานะของเครื่องเล่น สถานะของเทป และในส่วนของ พฤติกรรมของวิดีโอ (ในตัวอย่างนี้แสดงถึงว่าวิดีโอสามารถทำอะไรได้บ้าง) ก็คือ ความสามารถในการเปิด/ปิดเครื่อง การเล่นเทป การหมุนเทปไปข้างหน้าหรือถอยหลัง การหยุดเล่น และการหยุดชั่วคราว โดยจะสามารถนำมาเขียนให้อยู่ในรูปแบบของคลาสได้ดังต่อไปนี้

```
class Video {
      private boolean Power;
      private String Light;
      private boolean VideoTape;

      public void PowerOn(boolean) { // Implementation }
      public void InsertTape() { // Implementation }
      public void Play() { // Implementation }
      public void Stop()  { // Implementation }
      public void Pause()  { // Implementation }
      public void Eject()  { // Implementation }
      public boolean hasVideo() { // Implementation }
}
```

จากตัวอย่างนี้เป็นการสร้างคลาสที่ชื่อ Video ขึ้นมาโดยประกอบด้วยสมาชิกของคลาส Power, Light และ VideoTape โดย Power มีชนิดเป็นแบบตรรกะ (boolean) มีค่าเป็นจริงหรือเท็จ เพื่อบ่งบอกว่าวิดีโอนี้ทำงานหรือไม่ทำงาน สำหรับสัญญาณไฟ Light ใช้สำหรับแสดงไฟสีต่างๆสำหรับการทำงานตามฟังก์ชันต่างๆ และ VideoTape เป็นตรรกะในการตรวจสอบเมื่อมีเทปวิดีโอ ในส่วนของเมทธอดที่แสดงถึงฟังก์ชันการทำงานของเครื่องเล่นวิดีโอประกอบไปด้วยเมทธอด PowerOnOff(), InsertTape(), Play(), Stop(), Pause(), Eject() และการตรวจสอบว่ามีเทปอยู่ในเครื่องเล่น HasVideo() ซึ่งเมทธอดเหล่านี้อาจมีผลต่อการเปลี่ยนแปลงค่าของสมาชิกภายในคลาส ตัวอย่างเช่น การเรียกใช้งานเมทธอด PowerOnOff() ซึ่งทำให้เครื่องเล่นวิดีโอทำงานหรือหยุดทำงานซึ่งขึ้นอยู่กับสถานะของเครื่องเล่นในขณะนั้น

```
public void PowerOnOff() {
     if (Power == false)
         Power = true;
     else
         Power = false;
    }
```

สิ่งสำคัญอย่างหนึ่งในการออกแบบออบเจ็กต์คือ การซ่อนข้อมูลหรือสมาชิกภายในออบเจ็กต์ เพื่อไม่ให้ออบเจ็กต์อื่นๆ สามารถที่จะมองเห็นหรือเข้าถึงสมาชิกที่อยู่ภายในออบเจ็กต์ได้โดยตรง และการที่จะเข้าถึงสมาชิกเหล่านี้ได้จะต้องอาศัยเมทธอดที่เรียกว่า getters และ setters methods ซึ่งได้แก่ เมทธอดที่มีชื่อขึ้นต้นด้วยคำว่า get กับ set

```
...
public boolean getPowerOnOff() { return Power; }
public void setPowerOnOff(boolean power) { this.Power = power; }
public String getLight() { return Light; }
public void setLight(String light) { this.Light = light }
public boolean getVideoTape() { return VideoTape; }
public void setVideoTape(boolean video) { this.VideoTape = video; }
...
```

จากตัวอย่างนี้ จะเห็นได้ว่าการทำงานฟังก์ชัน get() ทำหน้าที่ในการส่งค่าปัจจุบันของสมาชิกกลับ และ set() ฟังก์ชันทำหน้าที่ในการกำหนดค่าที่รับเข้ามาจากพารามิเตอร์เพื่อกำหนดค่านั้นให้กับสมาชิกภายในออบเจ็กต์

3.3 อินสแตนซ์ของคลาส (Class Instance)

ในการพัฒนาโปรแกรมเชิงวัตถุ คลาสเปรียบเสมือนเป็นการสร้างชนิดของข้อมูลขึ้นมาอีกชนิดหนึ่ง เป็นชนิดที่มีโครงสร้างซับซ้อน (Complex data type) หรือมีโครงสร้างที่เป็นแบบ แอ็บสแต็ก (Abstract data type) เนื่องจากว่าคลาสนั้นประกอบไปด้วยส่วนที่เป็นข้อมูลที่เป็นสมาชิกของคลาสและพร้อมกับเมทธอด แต่ก่อนที่จะนำคลาสไปใช้งานได้จำเป็นที่จะต้องมีการสร้างอินสแตนซ์ของคลาสขึ้นมาก่อนเรียกว่า class instantiation ซึ่งคลาสอินสแตนซ์ก็คือ ออบเจ็กต์ที่ถูกสร้างขึ้นมาใหม่นั่นเอง ดังนั้นเราอาจจะเรียกได้ว่าคลาสนั้นทำหน้าที่เป็นพิมพ์เขียว (Template) ของออบเจ็กต์ การสร้างอินสแตนซ์ของคลาสในภาษาจาวาจะต้องอาศัยคำสั่ง new ตัวอย่างเช่น หากต้องการสร้างออบเจ็กต์วิดีโอขึ้นมาหนึ่งออบเจ็กต์จากคลาสวิดีโอหนึ่งออบเจ็กต์

```
class Program {
     public static void main(String args[]) {
          Video myVideo;
                myVideo = new Video();
      }
 }
```

จากตัวอย่างแสดงให้เห็นถึงเมทธอด main() ซึ่งเป็นเมทธอดสำคัญในการเป็นจุดเริ่มต้นทำงานของโปรแกรม โดยการทำงานเริ่มต้นจากการประกาศตัวแปร (Variable declaration) และการสร้างเป็นออบเจ็กต์ใหม่ขึ้นมาโดยอาศัย new

```
myVideo = new Video();
```

สำหรับ myVideo ในตัวอย่างนี้เป็นตัวแปรที่ทำหน้าที่อ้างอิงไปยังออบเจ็กต์ (Object reference) Video ซึ่งเราสามารถใช้ตัวแปรนี้ในการอ้างอิงถึงสมาชิกหรือเมทธอดที่อยู่ภายในออบเจ็กต์

3.4 การส่งข้อความระหว่างออบเจ็กต์ (Message Passing between Objects)

ในการพัฒนาโปรแกรมเชิงวัตถุ โปรแกรมที่เราสร้างขึ้นนั้นอาจประกอบไปด้วยหลายๆ ออบเจ็กต์ที่ทำงานร่วมกัน การที่ออบเจ็กต์จะสามารถทำงานร่วมกันได้นั้นจะต้องมีการสื่อสารระหว่างออบเจ็กต์ (Object interactions) โดยอาศัยเทคนิคเรียกว่า Message Passing นั่นก็คือ การส่งข้อความจากออบเจ็กต์หนึ่งไปยังอีกออบเจ็กต์ (Sending message) หรือการรับข้อความที่ถูกส่งมาจากอีกออบเจ็กต์หนึ่ง (Receiving message) ซึ่งสามารถทำได้โดยผ่านเมทธอดของออบเจ็กต์

การรับและส่งข้อความระหว่างออบเจ็กต์ถือว่าเป็นหัวใจสำคัญในการที่จะทำให้ออบเจ็กต์สื่อสารระหว่างกันได้ โดยการสื่อสารระหว่างกันนั้น ออบเจ็กต์ที่ออกแบบจะต้องกำหนดในเบื้องต้นว่าจะให้ออบเจ็กต์บริการ (Service) อะไรให้แก่ออบเจ็กต์อื่นๆได้บ้าง ยกตัวอย่างออบเจ็กต์ต่อไปนี้

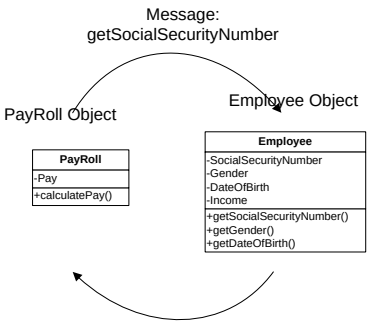
object Employee ประกอบด้วย

attribute	: SocialSecurityNumber, Gender , DateOfBirth, Income
method	: getSocialSecurityNumber, getGender, getDateOfBirth

object Payroll ประกอบด้วย

attribute	: Pay
method	: calculatePay

ออบเจ็กต์ Employee มีคุณสมบัติหมายเลขบัตร เพศ วันเดือนปีเกิดและรายได้ แต่คุณสมบัติเหล่านี้มักจะถูกซ่อน (private data member) เพื่อไม่ให้สามารถมองเห็นได้จากออบเจ็กต์อื่นๆ (อธิบายในบทที่ 4.2) ดังนั้นการที่จะให้ออบเจ็กต์อื่นๆสามารถที่จะเข้าถึงหรือสามารถเรียกใช้ Data Member ภายในได้ จะต้องเรียกผ่านเมทธอดที่ให้บริการเท่านั้น เช่นหากต้องการข้อมูลรหัส Social Security Number ของพนักงานก็จะต้องเรียกผ่านเมทธอด getSocialSecurityNumber ดังแสดงในรูป 3.1



รูป **3.1** แสดงการส่งค่าร้องขอข้อมูลระหว่างออบเจ็กต์

ในลักษณะเดียวกัน หากออบเจ็กต์ PayRoll ต้องการที่ทราบว่าพนักงานนั้นมีวันเดือนปีเกิดเป็นอย่างไร ก็จะต้องเรียกผ่านเมทธอด getDateOfBirth แต่ถ้ากรณีที่ออบเจ็กต์ PayRoll ต้องการข้อมูลเฉพาะ วันที่เกิด หรือปีที่เกิดของพนักงาน แต่ออบเจ็กต์ Employee ไม่ได้มีบริการในส่วนนี้ให้ก็จะไม่สามารถทำได้

ซึ่งจากตัวอย่างดังกล่าวจะเห็นได้ว่าการเรียกใช้งานเมทธอดที่ให้บริการรับและส่งค่ามักจะเริ่มต้นด้วยคำว่า get[MethodName] และคำว่า set[MethodName] ซึ่ง get จะหมายถึงเมทธอดที่ทำหน้าที่ส่งค่าจากออบเจ็กต์มาให้อีกออบเจ็กต์หนึ่ง และ set จะหมายถึงการที่เมทธอดนั้นทำหน้าที่ในการกำหนดค่าใดค่าหนึ่งให้แก่ออบเจ็กต์ ตัวอย่างเช่นในกรณีของออบเจ็กต์วิดีโอ ซึ่งเราสามารถเรียกเมทธอด

```
bool status = myVideo.getPowerOnOff();
```

เพื่อที่จะรับข้อความจากออบเจ็กต์วิดีโอด้วยการตรวจสอบสถานะของเครื่องเล่นเทป จากตัวอย่างนี้จะเห็นได้ว่าการสื่อสารกับออบเจ็กต์นั้นสามารถทำได้ด้วยการส่งข้อความหรือรับข้อความจากออบเจ็กต์โดยผ่าน เมทธอดที่สร้างไว้ ซึ่งการใช้งานเมทธอดนั้นจะใช้เครื่องหมายจุด (Period) ตามด้วยชื่อของเมทธอดดังตัวอย่างต่อไปนี้

```
Video myVideo = new Video();
myVideo.setPowerOnOff(true);
myVideo.setLight("Green");
myVideo.setVideoTape(true)
myVideo.Play();
myVideo.Stop();
myVideo.Eject();
myVideo.setPowerOnOff(false);
```

3.5 การประกาศค่าตัวแปรของออบเจ็กต์

จากตัวอย่างที่ผ่านมาจะเห็นว่าการสร้างอินสแตนซ์ของออบเจ็กต์นั้นจะต้องเริ่มต้นจากการสร้างตัวแปรของออบเจ็กต์ myVideo เราเรียกว่าเป็นตัวแปรที่ยังไม่ได้มีการกำหนดค่า (Uninitialize

variable) ซึ่งในจุดนี้ตัวแปร myVideo ไม่ได้มีค่าเป็นศูนย์แต่เป็นลักษณะของ null ซึ่งเหมือนกับว่าเป็นตัวแปรที่อ้างอิงถึงออบเจ็กต์ว่างที่ยังไม่ได้มีการสร้างขึ้นมา แต่ถ้าอยากให้ myVideo เป็นตัวแปรที่อ้างอิงถึงออบเจ็กต์วิดีโอก็จำเป็นต้องอาศัย new เช่น

```
Video myVideo;
myVideo = new Video();
```

หรือในกรณีที่ต้องการกำหนดตัวแปรและทำการสร้างอินสแตนซ์ของคลาสขึ้นมาใช้งานพร้อมกัน

```
Video myVideo = new Video();
```

3.6 การทำลายออบเจ็กต์ (Object Destruction)

การสร้างออบเจ็กต์ขึ้นมาใหม่โดยใช้คำสั่ง new นั้น เป็นการขอเนื้อที่ (Memory Allocation) จากหน่วยความจำให้กับออบเจ็กต์และเมื่อสิ้นสุดการใช้งานออบเจ็กต์นั้นแล้วควรจะต้องมีการทำลายออบเจ็กต์ที่สร้างเพื่อเป็นการล้างและคืนหน่วยความจำ (Memory Deallocation) ให้กับระบบปฏิบัติการ ในการพัฒนาโปรแกรมโดยทั่วไปนั้น การล้างหน่วยความจำจะต้องอาศัยโปรแกรมที่เขียนขึ้น หากไม่ทำอย่างนั้นแล้วก็อาจเกิดข้อผิดพลาดในการใช้งานหน่วยความจำได้เช่น หน่วยความจำรั่วไหล (Memory Leak) แต่สำหรับการเขียนโปรแกรมแบบออบเจ็กต์โอเรียนเท็ดนี้ การสร้างและการยกเลิกการใช้งานออบเจ็กต์จะเกิดขึ้นบ่อยครั้ง ดังนั้นเพื่อไม่ให้เกิดปัญหาของหน่วยความจำรั่วไหล ไม่ว่าจากสาเหตุใดก็ตาม กลไกควบคุมขยะอินสแตนซ์ (อินสแตนซ์ คือ พื้นที่ในหน่วยความจำซึ่งสร้างตามแบบโครงร่างของคลาสใดๆ ดังนั้น อินสแตนซ์คือ พื้นที่ที่สามารถเก็บข้อมูลและประมวลผลตามส่วนที่ประมวลผลได้นั่นเอง) เกิดขึ้น เนื่องจากเมื่อมีการสร้างโปรแกรมจำเป็นต้องมีการสร้าง อินสแตนซ์หลายๆ อินสแตนซ์เพื่อใช้งาน อินสแตนซ์บางอินสแตนซ์ถูกอ้างด้วยตัวแปรอ้างถึง แต่บางอินสแตนซ์ก็ไม่ถูกอ้างด้วยตัวแปร อ้างถึง ดังนั้นเมื่อโปรแกรมถูกทำงานเป็นระยะเวลาหนึ่ง อาจทำให้อินสแตนซ์ที่ไม่ถูกอ้างถึงเกิดเป็นขยะขึ้นและไม่สามารถนำกลับมาใช้งานหรืออ้างถึงได้อีกต่อไป

```
Date today;
today = new Date( ); //First instance
...
today = new Date( ); //Second instance
...
today = new Date( ); //Third instance
```

จากตัวอย่างข้างต้นจะเห็นว่าในโปรแกรมนี้มีการสร้างอินสแตนซ์ขึ้นมาทั้งสิ้น 3 อินสแตนซ์ (สังเกตจากการใช้คำสั่ง new ถึง 3 จุด) แต่มีการสร้างตัวแปรอ้างถึงเพียงตัวเดียว คือ today ตัวแปรอ้างถึงดังกล่าวมีการย้ายการเชื่อมโยงไปยังอินสแตนซ์ทั้งสามตัว ทำให้ช่วงสุดท้ายมีอินสแตนซ์ที่เป็นขยะ คือ อินสแตนซ์ที่หนึ่ง และ อินสแตนซ์ที่สอง ส่วนอินสแตนซ์ตัวที่สาม เป็นตัวที่ยังคงถูกใช้งาน ดังนั้นอินสแตนซ์ที่ค้างอยู่ในหน่วยความจำทำให้สิ้นเปลืองพื้นที่ หากมีอินสแตนซ์ที่เป็นขยะเป็นจำนวนมากๆ แล้ว สามารถทำให้โปรแกรมหยุดการทำงานได้ เนื่องจากไม่มีหน่วยความจำเหลือให้ใช้งาน

ต่อไป ผู้ใช้โปรแกรมอาจพบข้อความรายงานข้อผิดพลาดขึ้นมากมาย เช่น Insufficial Memory หรือ Out of Memory เป็นต้น

ภาษาที่สนับสนุนการทำงานแบบออบเจ็กต์จะมีกลไกพิเศษที่สามารถกำจัดขยะอินสแตนซ์ที่ไม่ถูกใช้งานเหล่านี้ออกจากหน่วยความจำ โดยทำงานในแบบฉากหลัง (Background Process) โดยผู้เขียนโปรแกรมไม่จำเป็นต้องเขียนคำสั่งกำจัดขยะด้วยตนเอง และผู้ใช้โปรแกรมก็ไม่รู้เลยว่ามีการกำจัดขยะอินสแตนซ์ขึ้นเมื่อใด เพราะเป็นกลไกที่ทำงานโดยอัตโนมัติ ซึ่งไม่เหมือนกับภาษาเชิงวัตถุตัวอื่นๆ ที่ผู้เขียนโปรแกรมต้องระมัดระวังปัญหาในลักษณะนี้ โดยต้องคืนพื้นที่ให้กับหน่วยความจำทุกครั้งที่ไม่มีการใช้งานอินสแตนซ์ เช่น ในภาษา C++ ต้องใช้คำสั่ง free เพื่อกำจัดอินสแตนซ์ทิ้งออกจากหน่วยความจำทุกครั้ง

โปรแกรมภาษาที่สนับสนุนการทำงานแบบออบเจ็กต์ในปัจจุบันจะมีตัวช่วยในการจัดการล้างหน่วยความจำให้อัตโนมัติด้วยเทคนิคเรียกว่า Garbage Collection การล้างหน่วยความจำจะเกิดขึ้นอัตโนมัติเมื่อสิ้นสุดการใช้งานออบเจ็กต์ โดยปรกติแล้วจะกระทำเมื่อโปรแกรมออกจากบล็อก (Scope) การทำงานของโปรแกรม ตัวอย่างเช่นโค๊ดต่อไปนี้

```
if (valueA ==  valueB)
{
    Video myVideo = new Video();
    myVideo.setPowerOnOff(true);
    myVideo.Play();
    myVideo.setPowerOnOff(false);
}
```

ในตัวอย่างนี้ ออบเจ็กต์ myVideo จะถูกสร้างขึ้นเมื่อค่าของ valueA เท่ากับ valueB ซึ่งการทำงานเกิดขึ้นและสิ้นสุดภายในบล็อก {...} และหลังจากออกจากบล็อกการทำงานของ if ออบเจ็กต์นี้ก็จะถูกทำลายอัตโนมัติ สำหรับการทำงานของ Garbage Collection นั้นจะคล้ายกับการควบคุมการมองเห็นของตัวแปรในการเขียนโปรแกรมทั่วไป และสำหรับการสร้างออบเจ็กต์ในเมทธอด main ซึ่งถือว่าเป็นบล็อกขอบเขตที่ใหญ่ที่สุด ออบเจ็กต์จะถูกทำลายก็ต่อเมื่อสิ้นสุดการใช้งานของโปรแกรมนั้น

3.7 คอนสตัคเตอร์ (Constructor)

คอนสตัคเตอร์เป็นเมทธอดที่มีชื่อเดียวกับชื่อคลาส ซึ่งจะถูกเรียกทุกครั้งที่มีการสร้างออบเจ็กต์ใหม่ขึ้นมาโดยอัตโนมัติ เมื่อออบเจ็กต์ใดๆ ถูกสร้างขึ้นมาภายใต้คลาสหนึ่งนั่นคือ โปรแกรมจะต้องไปเรียกใช้เมทธอดที่มีชื่อเดียวกันกับชื่อคลาสทันที และเรียกเมทธอดที่มีชื่อเดียวกันกับคลาสว่าคอนสตัคเตอร์เมทธอด ซึ่งประโยชน์ของการใช้งานได้แก่การเรียกใช้คำสั่งที่ต้องกระทำเสมอเมื่อมีการสร้างออบเจ็กต์ขึ้นมาใหม่ ตัวอย่างเช่นการกำหนดค่าเริ่มต้นให้กับตัวแปรแต่ทั้งนี้มีข้อจำกัดคือ

- คอนสตัคเตอร์เมทธอดจะต้องมีชื่อเดียวกันกับชื่อของคลาส
- ลักษณะของการประกาศคอนสตัคเตอร์เมทธอดต้องไม่มีค่าส่งกลับ หรือแม้กระทั่งคีย์เวิร์ด “void”

ดังนั้นถ้าชื่อเมทธอดใดๆในคลาสเป็นชื่อเดียวกันกับ คลาสดัวยแสดงว่าเมทธอดนั้นเป็น คอนสตัคเตอร์

ตัวอย่างการสร้างคอนสตัคเตอร์เช่น ในโปรแกรมสำหรับเกมส์บางอย่าง จำเป็นต้องเรียกใช้ตัวเลขสุ่ม ทุกครั้งที่เริ่มต้นเกมส์ใหม่ (เป็นการกำหนดค่าเริ่มต้นให้กับโปรแกรมทุกครั้งที่เรียกใช้โปรแกรม) แต่ทั้งนี้ไม่จำเป็นที่ทุกโปรแกรมจะต้องมีการกำหนดคอนสตัคเตอร์เมทธอดไว้ด้วย แต่เพื่อให้มีมาตรฐานเดียวกัน จึงนิยมเขียนคอนสตัคเตอร์เมทธอดเอาไว้ในโปรแกรมเสมอ ถึงแม้ว่าไม่มีการทำงานใดๆในคอนสตัคเตอร์เมทธอดก็ตาม โดยจะกำหนดเป็นบรรทัดว่างไว้ เช่น สมมติชื่อของคลาส คือ HelloApp ดังนั้นจะสร้างเมทธอดชื่อ HelloApp เอาไว้ด้วย โดยกำหนดเครื่องหมายปีกกาเปิดและปิดเอาไว้เท่านั้น (โดยไม่มีการทำงานใดๆ)

```
public HelloApp()
{ … }
```

โดยทั่วไปสำหรับโปรแกรมใหญ่ๆ ในคลาสหนึ่งๆ มักจะมีการกำหนดคอนสตัคเตอร์ไว้มากกว่า 1 คอนสตัคเตอร์โดยจะขึ้นอยู่กับค่าของพารามิเตอร์ที่ส่ง เรียกว่าโอเวอร์โหลด Overloading มาใช้งานร่วมกับคอนสตัคเตอร์ได้ โอเวอร์โหลดคือการประกาศเมทธอดที่มีชื่อเหมือนกันแต่การส่งจำนวนค่าพารามิเตอร์หรือชนิดของตัวแปรที่ส่งต่างกัน ตัวอย่างเช่น คลาส Circle ต่อไปนี้มีการประกาศคอนสตัคเตอร์อยู่สองตัวในลักษณะที่เป็นโอเวอร์โหลด ให้สังเกตว่าตัวแปรที่ส่งไปยังคอนสตัคเตอร์สองตัวนี้ต่างกัน

ในตอนเริ่มต้นของบทนี้ เราได้กล่าวถึงว่าออบเจ็กต์สามารถมีสถานะเป็นของตัวเองได้ การกำหนดสถานะของออบเจ็กต์นั้นสามารถทำได้ด้วยการเรียกใช้ผ่านเมทธอดที่ทำหน้าที่ในการกำหนดค่าให้กับสมาชิกภายในออบเจ็กต์ ซึ่งอาจจะต้องเรียกเมทธอดหลายๆ เมทธอดทำให้สิ้นเปลืองจำนวนเมทธอดสำหรับการ set() แต่อีกวิธีหนึ่งก็คือการใช้งานคอนสตัคเตอร์ (Constructor) ซึ่งเป็นการกำหนดค่าเริ่มต้นให้กับออบเจ็กต์เมื่อมีการสร้างออบเจ็กต์ขึ้นมาใหม่โดยอาศัย new คอนสตัคเตอร์ ก็คือเป็นเมทธอดอีกชนิดหนึ่งซึ่งจะถูกเรียกใช้ทันทีเมื่อออบเจ็กต์ถูกสร้างขึ้นมาใหม่ สำหรับการสร้างคอนสตัคเตอร์นั้นก็เหมือนกับการสร้างเมทธอดปกติทั่วไป เพียงแต่คอนสตัคเตอร์นั้นกำหนดไว้ว่าจะต้องมีชื่อเป็นชื่อเดียวกันกับชื่อของคลาส และเนื่องจากคอนสตัคเตอร์เป็นฟังก์ชันแรกที่ถูกเรียกใช้เมื่อออบเจ็กต์ถูกสร้าง เราจึงนิยมกำหนดค่าเริ่มต้นต่างๆ ไว้ในฟังก์ชันนี้

```
class Video {
    private boolean Power;
    private String Light;
    private boolean VideoTape;

    // Constructor
    Video(boolean Power, String Light, boolean VideoTape)
    {
        this.Power = Power;
            this.Light = Light;
        this.VideoTape = VideoTape;
    }
}
```

จะเห็นว่าในคอนสตัคเตอร์นี้มีการใช้คำสั่ง this เพื่อเป็นการอ้างอิงถึงสมาชิกที่เป็นของออบเจ็กต์ปัจจุบันเพื่อให้สามารถแยกระหว่างสมาชิกของออบเจ็กต์และค่าของพารามิเตอร์ที่ถูกส่งเข้ามา

3.7.1 การใช้คอนสตัคเตอร์ในคลาสที่สืบสกุล

คลาสที่ถูกสร้างขึ้นมาใหม่ด้วยวิธีการสืบสกุลนั้นสามารถที่จะมีคอนสตัคเตอร์เป็นของตัวเองเพื่อใช้สำหรับกำหนดค่าเริ่มต้นให้กับออบเจ็กต์ แต่อย่างไรก็ตามการกำหนดค่าเริ่มต้นนั้นจะใช้สำหรับสมาชิกที่เป็นของตัวเองเท่านั้น ส่วนสมาชิกที่เป็นของคลาสพื้นฐาน ก็จะต้องใช้วิธีเรียกใช้จากคอนสตัคเตอร์ที่เป็นของคลาสพื้นฐานด้วยคีย์เวิร์ด super() และส่งข้อมูลเป็นลักษณะของอาร์กิวเมนต์ไปให้

```
public class Employee
{
    String Name, LastName, Address, Phone;
    float Salary;

    Employee(String n, String l, String a, String p, float sal)
    {
        Name = n;
        LastName = l;
        Address = a;
        Phone = p;
        Salary = sal;
    }
}

class Manager extends Employee
{
   float Bonus;
   Manager(String n, String l, String a, String p,
            float sal, float b)
    {
        super(n, l, a, p, sal);// call constructor in base class
        Bonus = b;
    }
}

class MyProgram
{
     public static void main(String args[])
     {
          Manager = new Manager("John", "Doe", "USA",
          "245678", 1500.0, 100.0);
     }
}
```

จากตัวอย่างนี้ จะเห็นได้ว่าคลาส Employee และ คลาส Manager ต่างมีคอนสตัคเตอร์เป็นของตัวเอง ในกรณีที่สร้างอินสแตนซ์ของคลาส Manager ขึ้นมาใหม่จำเป็นจะต้องส่งอาร์กิวเมนต์ 6 ตัว

โดยห้าตัวแรก (“John”, “Doe”, “USA”, “245678”, 1500.0) จะส่งต่อไปให้กับคอนสตัคเตอร์ที่อยู่ในคลาสพื้นฐาน และอาร์กิวเมนต์ตัวสุดท้าย ซึ่งเท่ากับ 100.0 จะกำหนดให้กับตัวแปร Bonus ที่อยู่ในคลาส Manager

3.7.2 ดีฟอลต์คอนสตัคเตอร์ (Default Constructor)

การเรียกใช้คอนสตัคเตอร์ที่มีการรับค่าตัวแปรจะเป็นการบังคับให้เมทธอดที่เรียกจะต้องส่งค่าพารามิเตอร์มาให้ครบตามจำนวนที่กำหนด ซึ่งการส่งค่าพารามิเตอร์ในคอนสตัคเตอร์นั้นส่วนใหญ่จะนิยมใช้เพื่อการกำหนดค่าตั้งต้นให้แก่ออบเจ็กต์ เช่นคลาส Windows อาจจะมีคอนสตัคเตอร์เพื่อกำหนดตำแหน่ง สีและขนาด X1, Y2, X2, Y2 ของ Windows เพื่อเป็นค่าเริ่มต้น แต่ในบางครั้งผู้ใช้งานอาจจะไม่รู้ล่วงหน้าว่าค่าตั้งต้นควรจะเป็นค่าอะไร หรือไม่ต้องการที่จะระบุค่าตั้งต้น ดังนั้นภายในคอนสตัคเตอร์เองควรจะมีการกำหนดค่าตั้งต้นเป็นค่า Default ให้ไว้ เพื่อเป็นการกำหนดค่าตั้งต้นอัตโนมัติ

สำหรับคอนสตัคเตอร์ลักษณะนี้เรียกว่า Default Constructor เนื่องจากการประกาศ ไม่จำเป็นต้องระบุค่าพารามิเตอร์ใดๆ ดังตัวอย่างต่อไปนี้

```
class Boat {
int Speed;

   Boat() {    //Constructor
  Speed = 0;
   }
}
```

จากตัวอย่างนี้จะเห็นว่าคลาส Boat มีการประกาศเมทธอดที่เป็นคอนสตัคเตอร์ที่ปราศจากพารามิเตอร์ และเมื่อมีการสร้างอินสแตนซ์ของออบเจ็กต์ Boat ขึ้นมาใหม่ เช่น Boat myBoat = new Boat(); เมทธอดที่ทำหน้าที่เป็นคอนสตัคเตอร์จะถูกเรียกใช้อัตโนมัติโดยจะกำหนดค่าความเร็วของเรือ Speed ให้เท่ากับศูนย์ และถ้าหากต้องการให้โปรแกรมมีความยืดหยุ่นมากยิ่งขึ้นโดยจะให้มีการกำหนดค่าเริ่มต้นให้กับ Speed หรือไม่ก็ได้นั้น เราอาจจะสร้างคอนสตัคเตอร์ขึ้นมาอีกตัวหนึ่งในลักษณะของการโอเวอร์โหลดดังตัวอย่างต่อไปนี้

```
class Boat {
int Speed;

Boat() {
   Speed = 0;
    }

 Boat(int Speed) {
     this.Speed = Speed;
    }
}
```

ในตัวอย่างนี้จะเห็นว่ามีการประกาศคอนสตัคเตอร์สองตัวโดยที่ตัวแรกจะทำหน้าที่เป็น default คอนสตัคเตอร์ในกรณีที่ไม่มีการส่งค่าไปให้ และอีกตัวทำหน้าที่เป็นคอนสตัคเตอร์ที่ยอมให้ผู้ใช้กำหนดค่าของ Speed ได้เอง ซึ่งการสร้างคอนสตัคเตอร์ให้มีรูปแบบการทำงานที่ต่างกันนั้นทำให้โปรแกรมที่พัฒนามีความยืดหยุ่นในตัวเองมากยิ่งขึ้น

3.7.3 ดีสตัคเตอร์ (Destructor)

ดีสตัคเตอร์ (Destructor) ทำหน้าที่ในทางตรงกันข้ามกับคอนสตัคเตอร์ นั่นคือจะเป็น เมทธอดที่ถูกเรียกใช้อัตโนมัติเมื่อออบเจ็กต์ถูกทำลาย ซึ่งการประกาศคอนสตัคเตอร์ในแต่ละภาษาอาจจะต่างกัน ตัวอย่างเช่น ภาษาจาวาจะใช้ชื่อว่า finalize() แต่สำหรับภาษา C++ ดีสตัคเตอร์จะมีชื่อเดียวกับชื่อคลาสแต่จะมีเครื่องหมาย ‘~’ นำหน้า การใช้งานดีสตัคเตอร์นั้นมักนิยมใช้สำหรับการล้างหน่วยความจำ การปิดไฟล์ หรือยกเลิกการเชื่อมต่อกับฐานข้อมูล ตัวอย่างเช่น ในโปรแกรมขนาดใหญ่ที่ต้องมีการใช้ตัวแปรพอยน์เตอร์ หรือตัวแปรที่เป็นไดนามิกโดยการขอจองหน่วยความจำขนาดใหญ่ๆ จากระบบ เมื่อใดก็ตามที่เลิกใช้ข้อมูลนั้นๆ แล้ว จำเป็นจะต้องมีการคืนค่าที่จองไว้กลับคืนสู่ระบบด้วย เป็นการล้างหน่วยความจำ (Memory Clean-up) ให้กับระบบมีหน่วยความจำเพียงพอสำหรับพร้อมใช้งานต่อไปอย่างมีประสิทธิภาพ สำหรับ Destructor นั้นจะทำงานโดยมีจุดประสงค์ตรงข้ามกับ Constructor คือจะทำงานทุกครั้งที่วัตถุพ้นขอบเขตการทำงานของตัววัตถุนั้นๆ สำหรับคลาส x ใดๆ นั้นจะมี destructor ชื่อ ~x() ซึ่งกฎเกณฑ์การเรียกชื่อจะเหมือนกับคอนสตัคเตอร์ นั่นเอง

ในการเขียนโปรแกรมเชิงวัตถุ ดีสตัคเตอร์เป็นฟังก์ชันที่จะถูกเรียกใช้ก่อนที่ออบเจ็กต์จะถูกทำลายโดย Garbage collection โดยภาพรวมแล้วดีสตัคเตอร์นั้นจะทำหน้าที่ในทางตรงกันข้ามกับคอนสตัคเตอร์ ในที่นี้ก็คือ ดีสตัคเตอร์จะเป็นฟังก์ชันที่ถูกเรียกใช้เมื่อออบเจ็กต์ถูกทำลาย สำหรับการสร้างฟังก์ชันให้เป็นดีสตัคเตอร์ในภาษาจาวานั้นสามารถทำได้โดยใช้กำหนดชื่อฟังก์ชันเป็น finalize () ดังตัวอย่างต่อไปนี้

```
class Video {
       // Constructor
      public void Video(boolean Power, String Light, boolean VideoTape)
{
          this.Power = Power;
          this.Light = Light;
          this.VideoTape = VideoTape;
      }

       // Destructor
      void finalize() {
           // clean up memory
           // close file(s)
      }
}
```

โดยหลักการทำงานโดยทั่วไปนั้น ดีสตัคเตอร์จะใช้สำหรับการดำเนินการขั้นตอนสุดท้ายก่อนที่ออบเจ็กต์จะถูกทำลายเช่น การล้างหน่วยความจำ หรือปิดไฟล์ที่เปิดเอาไว้ระหว่างการทำงาน อย่างไรก็

ตามการที่อาศัยดีสตัคเตอร์ในการทำงานบางอย่างอาจจะไม่ใช่สิ่งที่ดีเสมอไปเนื่องจาก ดีสตัคเตอร์จะถูกเรียกใช้ก็ต่อเมื่อออบเจ็กต์จะถูกทำลาย ซึ่งจัดการโดย Garbage Collector ของภาษาจาวา ซึ่งเราอาจจะไม่สามารถคาดการณ์ล่วงหน้าได้ว่าถูกเรียกใช้งานเมื่อใด ซึ่งการล้างหน่วยความจำอัตโนมัติโดย Garbage Collector นั้นจะทำให้ช่วยลดความผิดพลาดที่เกิดจากปัญหาหน่วยความจำไม่เพียงพอ Memory Leak หรือ Out of Memory ได้

3.8 โอเวอร์โหลด (Overloading)

ในการเขียนโปรแกรมเชิงวัตถุนี้ ฟังก์ชันที่ประกาศใช้ภายในออบเจ็กต์เดียวกันสามารถมีชื่อซ้ำกันได้ เราเรียกการประกาศฟังก์ชันแบบนี้ว่าเป็นการโอเวอร์โหลดฟังก์ชัน (Function Overloading) แต่ทั้งนี้ค่าพารามิเตอร์ที่ส่งระหว่างฟังก์ชันจะต้องไม่เหมือนกัน คือจะส่งตัวแปรจำนวนเท่ากันก็ได้แต่ว่าชนิดของตัวแปรนั้นจะมีชนิดที่ต่างกัน

```
class Student {
            public void Display() {
                  System.out.println("The default name is "John");
            }


            public void Display(String Name) {
                  System.out.println("The name is: " + Name);
            }
    }
```

จากตัวอย่างนี้จะเห็นได้ว่ามีประกาศโอเวอร์โหลดฟังก์ชันที่ชื่อ Display() ฟังก์ชันหนึ่งสามารถเรียกใช้ได้โดยไม่ต้องส่งค่า และอีกฟังก์ชันหนึ่งจะรับค่าที่เป็นตัวแปรชนิดของ String โดยในการเรียกใช้ฟังก์ชันนี้จะสามารถทำได้โดย

```
Student s = new Student()
s.Display();        //Display() without any parameter
s.Display("Sam");    //Display(String) with one parameter
```

ในการเรียกใช้ฟังก์ชันที่ทำการโอเวอร์โหลด ถึงแม้ว่ามีการเรียกใช้ฟังก์ชันที่มีชื่อเหมือนกัน แต่คอมไพเลอร์ก็จะรู้ว่าจะต้องเรียกใช้ฟังก์ชันตัวไหน โดยใช้วิธีตรวจสอบจากค่าพารามิเตอร์ที่ส่ง

3.9 โอเวอร์โหลดคอนสตัคเตอร์ (Overloading Constructor)

จากการทำโอเวอร์โหลดฟังก์ชัน เราสามารถนำเอาหลักการเดียวกันนี้มาทำใช้กับการสร้างคอนสตัคเตอร์ เรียกว่าเป็นการทำโอเวอร์โหลดคอนสตัคเตอร์ โดยหลักการก็เช่นเดียวกันคือยอมให้ฟังก์ชันที่ทำหน้าที่เป็นคอนสตัคเตอร์มีชื่อที่ซ้ำกันได้ แต่อย่างไรก็ตามจำนวนและชนิดของค่าพารามิเตอร์ ที่ส่งก็จะต้องแตกต่างกัน

```
class Student
{
   private String Name;
   private String Address;
   Student()
   {
        //Default constructor
        this.Name = "";
        this.Address = "";
    }

//Overloading constructor
    Student(String Name, String Address) {
        this.Name = Name;
        this.Address = Address;
    }
   }
```

จะเห็นว่าคอนสตัคเตอร์ถูกทำการโอเวอร์โหลด โดยที่คอนสตัคเตอร์จะถูกเรียกใช้ก็ต่อเมื่อมีการสร้างออบเจ็กต์ใหม่ขึ้นมา และเมื่อออบเจ็กต์ถูกสร้างขึ้นมาใหม่ คอนสตัคเตอร์ที่สร้างไว้ก็จะถูกเรียกใช้ ซึ่งถ้ามีการประกาศฟังก์ชันที่เป็นคอนสตัคเตอร์มากกว่าหนึ่งตัว โปรแกรมจะเรียกใช้ คอนสตัคเตอร์ที่ถูกต้องซึ่งจะตรวจสอบจากจำนวนและชนิดของพารามิเตอร์ที่ส่งเข้ามา

```
Student s1 = new Student();
Student s2 = new Student("Fernandez","12/29St. Steven Street");
```

เมื่อออบเจ็กต์ s1 ถูกสร้างขึ้นมา คอนสตัคเตอร์ที่หนึ่งจะถูกเรียกใช้เนื่องจากไม่มีการส่งค่าให้ ในที่นี้ข้อมูลชื่อและที่อยู่ของ s1 จะถูกกำหนดให้เป็นว่าง สำหรับการสร้างออบเจ็กต์ s2 โดยส่งค่าสองค่านี้จะทำให้คอนสตัคเตอร์ตัวที่สองถูกเรียกใช้ โดยจะกำหนดค่า “Fernandez” ให้กับ Name และ “12/29 St. Steven Street" ให้กับ Address

3.10 สมาชิกและเมทธอดแบบสถิต (Static Members and Methods)

คลาสถูกออกแบบขึ้นมาเพื่อใช้สำหรับเป็นพิมพ์เขียวเพื่ออธิบายโครงสร้างออบเจ็กต์ และออบเจ็กต์จะถูกสร้างโดยคำสั่ง new เรียกว่าเป็นการสร้างออบเจ็กต์อินสแตนซ์ โดยออบเจ็กต์จะสามารถขึ้นมาได้ไม่จำกัดจำนวน ซึ่งขึ้นอยู่กับหน่วยความจำของเครื่องคอมพิวเตอร์ ซึ่งการออกแบบคลาสเพื่อให้สามารถสร้างอินสแตนซ์ของออบเจ็กต์นั้นอาจจะเหมาะกับในหลายๆกรณีที่มีความต้องการสร้างออบเจ็กต์ได้เรื่อยๆแบบ Dynamic ตามความต้องการ เช่นออบเจ็กต์ Car, Student, People แต่ในกรณีออบเจ็กต์อาจจะถูกสร้างขึ้นมาใช้งานเพียงแค่ครั้งเดียว และไม่มีความจำเป็นที่จะต้องมีการสร้างเพิ่มขึ้นมามากกว่า 1 ออบเจ็กต์ เช่นออบเจ็กต์ StringUtility ซึ่งเป็นออบเจ็กต์ที่รวมเอาเมทธอดช่วยเหลือสำหรับการจัดการกับ String อาจจะประกอบด้วย

```
class StringUtility {
       String Concatenation (String, String) { }
       String Convert(int) { }
       String Reverse(String) { }
}
```

ซึ่งการใช้งานในลักษณะนี้ไม่จำเป็นต้องมีการสร้างอินสแตนซ์มากกว่าหนึ่งอินสแตนซ์ ดังนั้นเราควรจะใช้การกำหนดโครงสร้างเมทธอดเป็นแบบสถิต (Static) ซึ่งการกำหนดเมทธอดแบบสถิตนี้มีลักษณะการทำงานที่ต่างจากเมทธอดทั่วไป คือเมทธอดแบบสถิตนี้จะถูกสร้างขึ้นมาทันทีโดยอัตโนมัติเมื่อโปรแกรมเริ่มต้นทำงาน เพื่อให้เมทธอดนี้สามารถพร้อมให้เรียกใช้งานโดยที่ไม่จำเป็นต้องมีการสร้างเป็นอินสแตนซ์ขึ้นมาใหม่ ซึ่งตัวอย่างที่เห็นได้ชัดเจนก็คือการประกาศเมทธอด Main() ในภาษาจาวา

```
public static void Main(String argv[]) { }
```

เมทธอด Main นี้จะถูกสร้างขึ้นมาเพื่อให้เรียกใช้งานได้ทันที สำหรับกรณีคลาส StringUtility นี้จะสามารถสร้างขึ้นมาโดยใช้หลักการของเมทธอดสถิตดังต่อไปนี้

```
class StringUtility {
       static String Concatenation (String, String) { }
       static String Convert(int) { }
       static String Reverse(String) { }
}
```

การที่มีการประกาศใช้งานเมทธอดที่เป็นแบบสถิตนี้จะทำให้คลาสที่ประกาศจะเป็นคลาสแบบสถิต (Static Class) ดังนั้นการเรียกใช้คลาสและเมดธอดที่เป็นแบบสถิตนี้จะสามารถเรียกใช้งานได้ทันทีโดยวิธีการอ้างจากชื่อของคลาส ตามด้วยเครื่องหมายจุด และชื่อของเมทธอดที่ต้องการเรียกใช้ โดยที่ไม่จำเป็นต้องสร้างเป็นอินสแตนซ์ขึ้นมาก่อน ดังตัวอย่างต่อไปนี้

```
String message1 = "Information";
String message2 = "Retrieval";
String result = StringUtility.Concatenation(message1, message2);
```

สำหรับการประกาศตัวแปรที่เป็นแบบสถิตนั้น มีหลักการทำงานในลักษณะเดียวกัน คือตัวแปรที่อยู่ภายในแต่ละคลาสนั้นเป็นการประกาศชื่อสมาชิกที่เป็นของแต่ละออบเจ็กต์ ดังนั้นเมื่อออบเจ็กต์แต่ละตัวถูกสร้างขึ้นมา สมาชิกเหล่านี้ก็จะถูกสร้างขึ้นมาพร้อมกับออบเจ็กต์แต่ละตัวด้วย และเช่นเดียวกันกับการทำลายออบเจ็กต์ สมาชิกของออบเจ็กต์ก็จะถูกทำลายไปด้วย สำหรับเมทธอดหรือสมาชิกที่ประกาศเป็นแบบสถิตจะมีลักษณะที่ต่างไปจากสมาชิกแบบปกติดังนี้ คือตัวแปรที่เป็นแบบสถิตจะถูกสร้างขึ้นมาเพียงครั้งเดียว ไม่ว่าจะมีการสร้างอินสแตนซ์ของคลาสกี่ครั้งก็ตาม ในลักษณะเดียวกันกับการใช้เมทธอดที่เป็นแบบสถิต นั่นก็คือจะเป็นเมทธอดที่สามารถมองเห็นและเรียกใช้ได้จากคลาสอื่นๆ ดังนั้นเมทธอดที่เป็นแบบสถิตจะสามารถเข้าถึงตัวแปรที่เป็นแบบสถิตได้เท่านั้น

```
static int Counter;
static void IncrementCounter() {
      Counter++;
}
```

ข้อดีของการประกาศตัวแปรและเมทธอดที่เป็นแบบสถิตคือ จะสามารถเรียกใช้โดยไม่ต้องอาศัยการสร้างอินสแตนซ์ของคลาสขึ้นมาใหม่ หรือในกรณีที่ต้องการใช้งานตัวแปรหรือเมทธอดร่วมกันระหว่างคลาสต่างๆ แต่ข้อเสียของตัวแปรสถิตในที่นี้คือตัวแปรแบบนี้จะเหมือนกับตัวแปรที่เข้าถึงได้ทั่วไป (Global) ซึ่งเมทธอดอื่นสามารถมองเห็นและแก้ไขได้ หากมีข้อผิดพลาดจะทำให้ตรวจสอบได้ยากกว่า ซึ่งจะต้องมีการออกแบบการทำงานให้ดี

3.11 แอ็บสแต็กคลาสและแอ็บสแต็กเมทธอด (Abstract Class and Abstract Method)

จากตัวอย่างของการสืบสกุลระหว่างคลาส Employee และคลาส Manager กับ คลาส Sales ที่ผ่านมา จะเห็นว่าไม่มีการสร้างอินสแตนซ์ของคลาส Employee เกิดขึ้น เนื่องจากในโจทย์ถูกระบุไว้ว่า “ไม่มีพนักงานคนไหนเลยที่เป็นพนักงานทั่วไป'” คือ พนักงานจะต้องถูกจัดอยู่ในกลุ่มใดกลุ่มหนึ่ง จะเห็นได้ว่าคลาส Employee ทำหน้าที่เป็นแค่พิมพ์เขียวให้คลาสอื่นสืบสกุลเท่านั้น ในกรณีอย่างนี้เราอาจจะประกาศคลาส Employee ให้เป็นลักษณะของแอ็บสแต็ก (Abstract) คลาสเพื่อเป็นการบ่งบอกว่าจะไม่มีการสร้างอินสแตนซ์ของคลาสนี้เกิดขึ้น ลักษณะรูปแบบของคลาสที่เป็นแอ็บสแต็ก คือ ใช้คีย์เวิร์ด abstract ตามด้วยชื่อของคลาสเช่น

```
public abstract class Employee {
      // Class Body
}

public class Manager extends Employee {
     // Class Body
}
```

การจะเรียกใช้งานคลาสที่เป็นแอ็บสแต็กได้นั้นจะทำได้โดยการสร้างคลาสขึ้นมาใหม่เพื่อสืบสกุลจากคลาสแอ็บสแต็กนั้น ดังในตัวอย่างหลังจากที่คลาส Manager สืบสกุลจากคลาส Employee แล้ว ก็จะสามารถสร้างเป็นอินสแตนซ์ของคลาส Manager ได้

```
Employee e = new Employee();        // Error
Manager m = new Manager();          // Ok
```

คลาสที่เป็นแอ็บสแต็กนั้นสามารถมีเมทธอดภายในเป็นแบบแอ็บสแต็กได้อีกด้วย แอ็บสแต็กเมทธอด คือ ฟังก์ชันที่มีการประกาศเฉพาะชื่อและโครงสร้างเท่านั้น แต่จะไม่มีการเขียนโค๊ดคำสั่งภายใน (Implementation)

```
public abstract class Employee {
       abstract float CalculateBonus();  // Abstract Method
}
```

จะเห็นได้ว่าภายในคลาส Employee มีการประกาศแอ็บสแต็กเมทธอด CalculateBonus() โดยคีย์เวิร์ด abstract ก่อนการประกาศเมทธอด และสังเกตว่าในส่วนของฟังก์ชันการทำงานจะไม่มีการสร้างไว้ โดยเขียนในรูปแบบ CalculateBonus(); สำหรับในตัวอย่างนี้เป็นการแสดงถึงว่าการเก็บข้อมูลพนักงานภายในบริษัท จะมีการคำนวณเงินโบนัสให้กับพนักงาน แต่รายละเอียดขั้นตอนหรือสูตรการคำนวณสำหรับพนักงานแต่ละกลุ่มอาจจะไม่เหมือนกัน ดังนั้นคลาส Employee ที่เป็นแอ็บสแต็กจึงทำได้แค่เพียงประกาศเป็นแอ็บสแต็กเมทธอดไว้ให้เพื่อที่เวลาคลาสที่สืบสกุลจะสามารถโอเวอร์ไรด์เป็นเมทธอดที่มีฟังก์ชันการทำงานเฉพาะเป็นของตัวเองได้

```
public class Manager {
      public float CalculateBonus()
      {
             float Bonus = Salary * 0.10;
                 return Bonus;
          }
    }

     public class Sales {
        // Overriding CalculateBonus()
        public float CalculateBonus() {
             float Bonus = Salary * 0.5;
             return Bonus;
          }
    }
```

จากที่ผ่านมา เราพอจะเห็นวิธีการทำงานของแอ็บสแต็กคลาสบ้างแล้ว พอสรุปได้ว่าแอ็บสแต็กคลาสและเมทธอดทำหน้าที่เป็นพิมพ์เขียวของโครงสร้างให้สำหรับคลาสที่จะมาสืบสกุลต่อไป ซึ่งจะทำให้รูปแบบและโครงสร้างของคลาสที่ออกแบบชัดเจนมากยิ่งขึ้น แต่อย่างไรก็ตามการใช้งานแอ็บสแต็กคลาสก็ยังมีข้อจำกัดซึ่งพอจะสรุปได้ดังนี้

1. แอ็บสแต็กคลาสจะไม่สามารถนำไปสร้างเป็นอินสแตนซ์ได้ แต่จะสามารถใช้เป็นพิมพ์เขียวให้กับคลาสอื่นได้ด้วยการทำการสืบสกุลจากคลาสแอ็บสแต็ก
2. เมทธอดที่ประกาศเป็นแบบแอ็บแสต็กจะไม่สามารถใช้ร่วมกับคีย์เวิร์ด static ได้ เนื่องจาก static เมทธอดเป็นการสร้างเมทธอดแบบมองเห็นได้สำหรับทุกคลาส ถ้าเมทธอด static เป็นแอ็บสแต็กจะทำให้ไม่สามารถสร้างเมทธอดโอเวอร์ไรด์ได้
3. การประกาศเมทธอดที่เป็นแบบแอ็บสแต็กจะไม่สามารถประกาศเป็นแบบ private ได้ เนื่องจากการใช้งานคลาสที่เป็นแอ็บสแต็กจำเป็นที่จะต้องมีการสืบสกุลและทำการ โอเวอร์ไรด์เมทธอด ถ้าหากว่าเมทธอดที่อยู่ในแอ็บสแต็กคลาสประกาศเป็นแบบ private แล้วจะทำให้ไม่สามารถมองเห็นจากข้างนอกคลาสได้ ทำให้ไม่สามารถโอเวอร์ไรด์ได้เช่นกัน

3.12 เมทธอดแบบซ้อนทับ (Override Method)

ในการสืบสกุลระหว่างคลาสพื้นฐานและคลาสสืบสกุล จะเห็นได้ว่าการสืบสกุลทำให้คลาสที่สืบสกุลนั้นได้รับเอาคุณสมบัติของคลาสพื้นฐานทั้งหมดมาเป็นของตัวเอง ทั้งส่วนที่เป็นสมาชิกและส่วนที่เป็นเมทธอด แต่ในบางครั้งของการทำงาน เมทธอดที่สืบทอดมาจากคลาสพื้นฐาน อาจจะไม่สามารถทำงานได้ตรงกับความต้องการของคลาสที่สืบสกุล ดังนั้นการเขียนโปรแกรมแบบออบเจ็กต์ จะยอมให้คลาสสืบสกุลสร้างเมทธอดขึ้นมาใหม่ภายในคลาสเองและเมทธอดที่สร้างนั้นให้มีชื่อเดียวกันกับเมทธอดที่อยู่ในคลาสพื้นฐานได้ เรียกว่าเป็นการสร้างเมทธอดแบบซ้อนทับ (Overriding Method)

```
class Employee {
        String Name, LastName, Address, Phone;
        float Salary;

         void Display() {
            System.out.println(Name + " " + LastName);
        }
 }

 class Manager extends Employee {
        float Bonus;
        void Display() {   // Overriding Method
             // Calling Display() in Employee
             super.Display();
             system.out.println("Bonus is: " + Bonus);
       }
 }
```

จากตัวอย่าง จะเห็นว่าคลาส Manager สร้างเมทธอด Display() ขึ้นมาอีกตัวหนึ่งซึ่งเป็นการซ้อนทับเมทธอด Display() ที่อยู่ในคลาสพื้นฐาน ซึ่งเมทธอด Display() ที่ถูกสร้างขึ้นมาใหม่สามารถแสดงผลลัพธ์ได้มากกว่า และสำหรับการเรียกใช้งานเมทธอด Display() ที่อยู่ในคลาส Manager ก็สามารถทำได้โดยการเรียกเมทธอด Display() ได้เลย แต่หากต้องการเรียกใช้งาน Display() ที่อยู่ในคลาสพื้นฐานจะต้องอาศัยคีย์เวิร์ด super เพื่อบ่งชี้ถึงเมทธอดที่อยู่ในคลาสแม่ (Parent Class) เช่น super.Display();

3.13 ไฟนอลคลาส ไฟนอลเมทธอด และตัวแปรแบบไฟนอล (Final)

ไฟนอลในภาษาจาวาสำหรับการประกาศตัวแปรที่เป็นค่าคงที่ (constant) ซึ่งเป็นตัวแปรที่ไม่สามารถเปลี่ยนแปลงค่าได้ สำหรับตัวแปรที่เป็นไฟนอลนั้นอาจจะเขียนโดยใช้ตัวอักษรตัวใหญ่ทั้งหมดเพื่อให้ดูแตกต่างจากตัวแปรประเภทอื่นๆ

```
final int PI = 3.141592;
final float DOLLAR = 25.0;
```

นอกจากไฟนอลจะใช้สำหรับระบุตัวแปรที่เป็นค่าคงที่แล้ว ไฟนอลยังใช้สำหรับระบุคลาส ที่ไม่ยอมให้เกิดการสืบสกุล และเมทธอดที่ไม่ยอมให้เกิดการโอเวอร์ไรด์ได้อีกด้วย ตัวอย่างเช่น ถ้าต้องการสร้างคลาส TrafficLight ขึ้นมาซึ่งจะมีเพียงคลาสเดียวและไม่มีการสืบสกุลจากคลาสนี้อีก ซึ่งควรระบุคลาสนี้ให้เป็นแบบ final เพื่อไม่ให้เกิดการสืบสกุลขึ้นอีก ตัวอย่างเช่น

```
final class TrafficLight {
      // Class Body
}
```

และเมื่อมีการสร้างคลาสขึ้นมาใหม่เพื่อสืบทอดจากคลาส TrafficLight เช่น

```
public class MyTrafficLight extends TrafficLight
```

โปรแกรมคอมไพล์เลอร์ก็จะฟ้องข้อผิดพลาดทันที

หรือในอีกกรณีหนึ่งก็คือ การใช้ไฟนอลเพื่อควบคุมไม่ให้เกิดการโอเวอร์ไรด์เมทธอด เช่น เมื่อมีการสืบสกุลเกิดขึ้น เมทธอดที่อยู่ในคลาสแม่อาจจะถูกโอเวอร์ไรด์ได้ หากว่าเราไม่ต้องการให้มีการโอเวอร์ไรด์ในเมทธอดใดๆในคลาสนั้นอีก เราก็ควรจะระบุเมทธอดนั้นในคลาสกลายเป็นเมทธอดไฟนอลไปด้วย ดังตัวอย่างต่อไปนี้ เมทธอด calculateArea() จะเป็นเมทธอดสุดท้าย ซึ่งจะไม่สามารถโอเวอร์ไรด์ได้อีก

```
public class Circle {
        final float PI = 3.141592;
        int radius;

        final float calculateArea() {
            return (PI * radius * radius);
        }
}
```

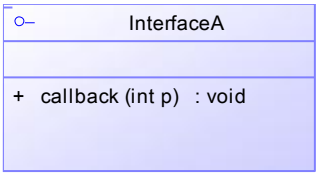
3.14 อินเตอร์เฟส (Interface)

อินเตอร์เฟสทำหน้าที่คล้ายกับ “Abstract Class” (แต่จะเป็น Abstract Class ที่สมบูรณ์กว่า) โดยการนิยามอินเตอร์เฟสนั้นจะเหมือนกับการนิยามคลาสเพียงแต่ภายใน Interface จะมีเฉพาะตัวแปรที่มีค่าคงที่และ Abstract Method เท่านั้น ทั้งนี้เมื่อโปรแกรมใดก็ตามที่มีการใช้งาน “interface” จะหมายถึงการกำหนดว่าทุกคลาสในโปรแกรมนั้นจะต้องสร้างเมทธอดที่มีชื่อตามที่ระบุไว้ใน Interface ด้วย โดยเนื้อหาของเมทธอดอาจจะแตกต่างกันไปในแต่ละคลาสขึ้นอยู่กับความต้องการในการใช้งาน (แต่ชื่อเมทธอดต้องเป็นชื่อเดียวกัน)

ประโยชน์ของการนำอินเตอร์เฟสมาใช้นั้น ก็คือ การทำ “Multiple inheritance” สำหรับการเขียนโปรแกรมเชิงวัตถุ (Object Oriented Programming) การสร้าง Polymorphism เมทธอดและการกำหนดค่าคงที่ให้กับทุกคลาสที่มีในโปรแกรมนั้น

3.14.1 โครงสร้างของอินเตอร์เฟส

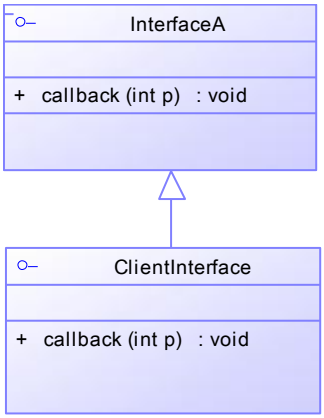
การประกาศคลาสที่เป็น Interface นี้จะใช้คำว่า “interface” (แทนการระบุคำว่า “Class”) รูปแบบของการประกาศคลาสที่เป็นอินเตอร์เฟส จะสังเกตว่าเมทธอดที่อยู่ภายในอินเตอร์เฟสจะกำหนดเป็น Abstract Method



รูป **3.2** รูปแบบการประกาศ **interface**

```
interface A
{
    void callback (int param) ;       // Abstract method
}
```

กำหนด interface ชื่อว่า “A” และกำหนดว่าในทุก ๆ Class ที่ implements “A” จะต้อง Override Method “callback” เอาไว้ด้วย (แต่รายละเอียดของเมทธอดอาจแตกต่างกันไปได้) การใช้งานอินเตอร์เฟสสามารถทำได้โดยการระบุคำว่า “implements” แล้วตามด้วยชื่ออินเตอร์เฟสไว้ในบรรทัดของการกำหนดชื่อคลาสจากนั้นจึงกำหนดชื่อเมทธอดพร้อมรายละเอียดไว้ในคลาสนั้น



รูป **3.3** การ implement interface

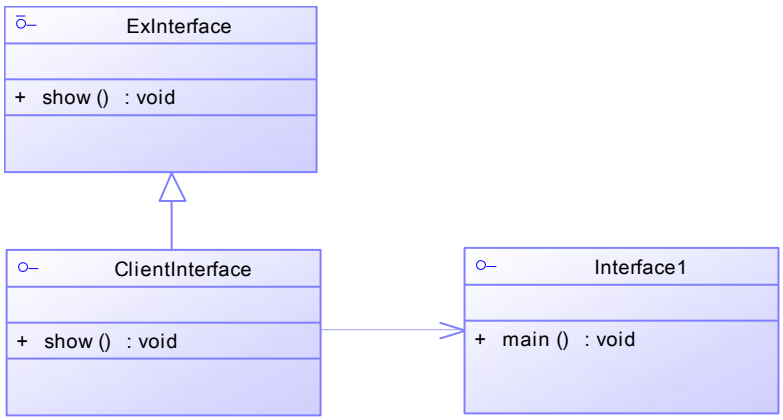
```
class client implements A
{
    public void callback (int p);
{
```

จากตัวอย่างนี้ A เป็นชื่อของอินเตอร์เฟสและ callback เป็นเมทธอดที่ทำการ Override เมทธอดที่ประกาศไว้ก่อนเดิมภายใน Interface ซึ่งการประกาศเมทธอดที่ใช้ในการอ้างอิงอินเตอร์เฟส

ควรระบุค่า Accessibility เป็น “public” เพื่อป้องกันกรณีที่สร้าง interface ไว้ต่าง Folder กันกับโปรแกรมที่ ใช้งาน (แต่ในกรณีที่อยู่ภายใน Folder เดียวกัน ก็ไม่จำเป็นต้องกำหนดไว้เป็น “public” และสำหรับอินเตอร์เฟสไม่ควรมีการสร้าง Constructor Method)

3.14.2 การเรียกใช้งานอินเตอร์เฟส

อินเตอร์เฟสมีลักษณะคล้าย ๆ กับ Abstract Class คือใช้ interface อ้างอิง (reference) ในการสร้างออบเจ็กต์แต่จะสร้างออบเจ็กต์จาก instance ไม่ได้ และภายในโปรแกรมหากมีการสร้าง interface ไว้จะต้องสร้าง Class ที่มี implementation ของ Abstract Method ใน interface ให้ครบเสียก่อน (Override Method) จึงจะสามารถนำ Class นั้นไปใช้งานเป็น Subclass ของ Class ได้



รูป **3.4** การสืบทอด **interface** เพื่อทำการ **implement** เป็นคลาสใหม่

ตัวอย่าง แสดงการนำ interface มาสร้าง Class วงกลม (circle) แล้วให้ Class อื่นทำการ implement ต่อ

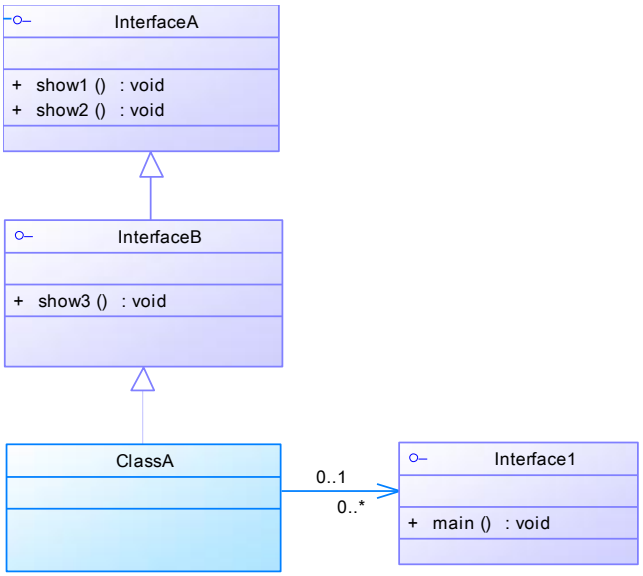
```
public class interface3 {
    public static void main (String [] args) {
    Circle a = new CricleA(5.3);
    a.showArea();
    }
}

interface Circle {
    public final double PI = 3.14159;
    public double findArea(double r);
    public void showArea();
}

class circleA Implements Circle
{
    private double r;
```

```
        circleA (double r) { //constructor
            this.r = r;
        }

        public double findArea(double r) {
        return Pi * r * r;
        }

        public void showArea() {
            system.out.println("ค่ารัศมี : " + r);
        system.out.println("พื้นที่ของวงกลม : " + findArea(r));
        }
    }
```

จากตัวอย่างข้างต้นนี้เป็นการสร้างอินเตอร์เฟส “circle” เพื่อใช้คำนวณพื้นที่วงกลมและให้แสดงพื้นที่ที่คำนวณได้ โดยภายในอินเตอร์เฟสจะมีตัวแปร PI ซึ่งกำหนดให้เป็นค่าคงที่ เพื่อใช้สำหรับคำนวณพื้นที่ทั้งนี้ได้กำหนด Abstract Method ไว้ 2 เมทธอด เมื่อคลาสที่จะอิมพลีเมนท์ ทำการ Override หากออบเจ็กต์ที่สร้างขึ้นอ้างอิงกับอินเตอร์เฟสภายในคลาสที่ทำการอิมพลีเมนท์ จะไม่สามารถสร้างเมทธอดอื่นนอกเหนือจากที่ได้กำหนดไว้ในอินเตอร์เฟส เช่น หากในอินเตอร์เฟส ไม่ได้กำหนดเมทธอด “showArea()” ไว้ก็จะไม่สามารถสร้างเมทธอด“showArea()” ในคลาสที่ทำการอิมพลีเมนท์ และหากออบเจ็กต์ที่สร้างขึ้นมาไม่ได้อ้างอิงกับอินเตอร์เฟส ก็สามารถสร้างเมทธอดที่ไม่ได้กำหนดไว้เป็น Abstract Method ได้ แต่จะไม่สามารถนำออบเจ็กต์ที่สร้างขึ้นไปใช้งานกับ Polymorphism ได้ เพราะออบเจ็กต์ที่สร้างขึ้นมาไม่ได้อ้างอิงกับคลาสในระดับสูงสุดซึ่งก็คืออินเตอร์เฟสนั่นเอง

3.14.3 การสืบทอดในอินเตอร์เฟส

อินเตอร์เฟสสามารถใช้การถ่ายทอดคุณสมบัติได้เช่นเดียวกันกับคลาสโดยใช้คีย์เวิร์ด “extends” ทั้งนี้คลาสที่เรียกใช้จะต้องทำการอิมพลีเมนท์ทุกเมทธอดที่ระบุใน interface ด้วย ตัวอย่างเช่น

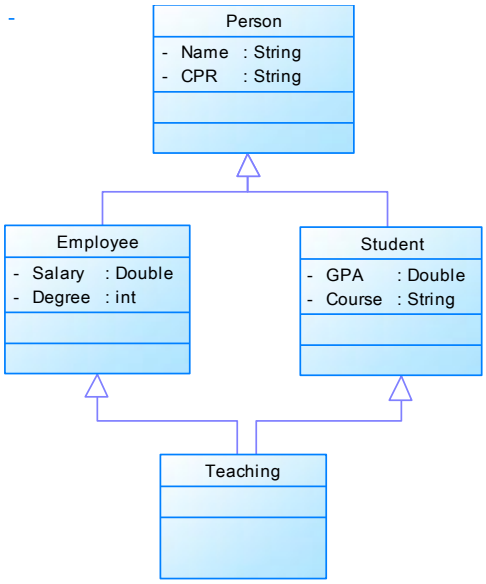


รูป 3.5 การสืบทอดสืบสกุลกรณีคลาสที่เป็น interface

```
public class ClassA {
     public static void main(String [] args)
     {
         interfaceB a = new A();
         a.show1();
         a.show2();
         a.show3();
     }
}

interface interfaceA {
    void show1();
    void show2();
}

interface interfaceB extends interfaceA {
    public void show3();
}

class A implements interfaceB {
    public void show1(){
        system.out.println("Method show1() Override
              จาก inter B*");
    }

    public void show2(){
        system.out.println("Method show2() Override
            จาก inter B*");
     }

     public void show3(){
        system.out.println("Method show3() Override
            จาก inter B*");
    }
 }
```

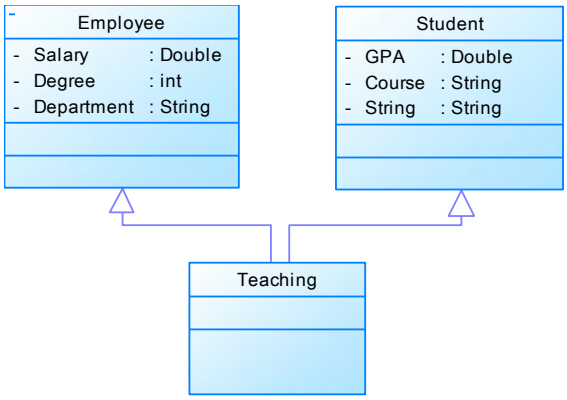
สำหรับอินเตอร์เฟส “interA” มี 2 แอ็บสแต็กเมทธอดส่วนอินเตอร์เฟส “interB” มีการสืบทอดมาจาก “interA” ดังนั้น ในอินเตอร์เฟส“interB” จึงมี 3 แอ็บสแต็ก หากคลาสใดทำการ implement “interB” ก็จะต้องทำการ Override Abstract ทั้ง 3 ตัว ให้ครบ จึงจะสามารถนำคลาสดังกล่าวไปสร้างออบเจ็กต์ได้ การสร้างออบเจ็กต์จะต้องอ้างอิงกับ “interB” เพราะหากอ้างอิงกับ “interA” Object ที่สร้างขึ้นจะไม่สามารถอ้างอิงถึง Method”show3()”ได้

ภาษาที่สนับสนุนการทำงานแบบออบเจ็กต์บางภาษา เช่น C++ สามารถทำการสืบทอดแบบหลาย Parent มีลักษณะเป็น Multiple Inheritance คือสับคลาสสามารถสืบสกุลจากซูเปอร์คลาสได้มากกว่าหนึ่งคลาส เช่น



รูป **3.6** การสืบทอดแบบ **multiple inheritance** ของ **interface**

คลาส Teaching สืบสกุลมาจากคลาส Employee และ Student ดังนั้นคลาส Teaching จะประกอบด้วยสมาชิก คือ name, cpr, salary, degree, gpa และ courses แต่บางครั้งก็ก่อให้เกิดปัญหา เช่นตัวอย่างของการสร้างคลาสเพื่อการสืบสกุลแบบ multiple inheritance ดังต่อไปนี้



รูป **3.7** การสืบทอดแบบ **multiple inheritance** ของ **interface** ที่อาจก่อให้เกิดปัญหา

เมื่อมีการสร้างออบเจ็กต์ ของคลาส Teaching ซึ่งจะเกิดความสับสนในเรื่องของการที่มีชื่อซ้ำกันของชื่อสมาชิกที่สืบทอดมาจากแต่ละคลาส ตัวอย่างเช่น การอ้างอิงถึง department ก็จะหมายถึง department จาก Employee หรือ Student สำหรับภาษาที่สนับสนุนการทำงานแบบออบเจ็กต์ในบางภาษาจะไม่ยอมให้มีการสืบทอดจากหลายคลาสเพื่อลดปัญหานี้เช่น ในภาษาจาวาที่ยอมให้มีการสืบสกุลแบบ Single Inheritance เนื่องจาก มีแนวคิดว่า Multiple Inheritance ไม่จำเป็นต่อโครงสร้างของการ สืบสกุลซึ่งจะทำให้เกิดปัญหามากกว่าผลดี และอีกอย่างคือจาวามีคลาสอินเตอร์เฟสอยู่แล้ว ซึ่งสามารถนำมาใช้ในการทำโครงสร้างที่คล้ายกับ Multiple Inheritance ได้เช่นเดียวกัน เช่น คลาสลูกสามารถสืบสกุลจากซูเปอร์คลาส ได้เพียงคลาสเดียว แต่สามารถ implement ได้หลายๆ interface

การสร้างอินเตอร์เฟสจะคล้ายกับการสร้างคลาสแอ็บสแต็ก แต่อินเตอร์เฟสจะมีข้อจำกัดมากกว่าคลาสแอ็บสแต็ก ดังนี้

1. อินเตอร์เฟสไม่สามารถมีตัวแปรสมาชิก (data member) แต่สามารถมีค่าคงที่ (ตัวแปรค่าคงที่)ได้
2. เมทธอดที่สร้างภายในอินเตอร์เฟสเป็นแอ็บสแต็กเมทธอด เท่านั้น ซึ่งแตกต่างแอ็บสแต็กคลาสซึ่งสามารถมีเมทธอดปกติได้
3. ค่าคงที่และแอ็บสแต็กเมทธอด ในอินเตอร์เฟสต้องเป็น Public เท่านั้น

การใช้งานอินเตอร์เฟสจะคล้ายกับแอ็บสแต็กคลาส คือไม่สามารถสร้าง อินสแตนซ์ของ interface ได้ แต่จะต้องสร้างคลาสที่มีการอิมพลีเมนท์ แอ็บสแต็กเมทธอด ในอินเตอร์เฟสให้ครบเสียก่อน แล้วจึงจะสามารถนำเอาคลาสนั้นไปใช้ได้ ลักษณะของภาษาที่มีการสืบสกุลแบบ Single Inheritance จะสืบสกุลจากซูเปอร์คลาสได้เพียง 1 คลาสเท่านั้น แต่ด้วยกลไกของอินเตอร์เฟสจะทำให้สามารถเข้าถึงการกำหนดนิยามในคลาสอื่น ได้หลายๆ คลาสในขณะที่ยังคงสืบสกุลจากซูเปอร์คลาสเพียงคลาสเดียว

บทที่ 4 - องค์ประกอบแนวคิดเชิงวัตถุ

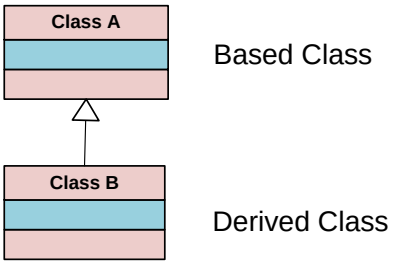
การพัฒนาซอฟต์แวร์โดยอาศัยเทคโนโลยีเชิงวัตถุนั้นมีองค์ประกอบที่สำคัญคือ 1) การวิเคราะห์ปัญหาโดยอาศัยแนวคิดเชิงวัตถุ (Object-Oriented Analysis: OOA) ซึ่งเป็นการวิเคราะห์ความต้องการของผู้ใช้งานและนำมาวิเคราะห์และสร้างแบบจำลองเชิงวัตถุ 2) การออกแบบเชิงวัตถุ (Object-Oriented Design: OOD) เป็นกระบวนการออกแบบโดยอาศัยแนวคิดเชิงวัตถุ ซึ่งในขั้นตอนนี้เป็นการพัฒนาโมเดลที่ผ่าน การวิเคราะห์ไปสู่การออกแบบซึ่งผลลัพธ์ที่ได้อาจอยู่ในรูปของไดอะแกรมต่างๆเช่น UML (Unified Modeling Language) และการพัฒนาซอฟต์แวร์เชิงวัตถุ (Object-Oriented Programming: OOP) ซึ่งเป็นกระบวนการการพัฒนาซอฟต์แวร์ เพื่อให้เกิดเป็นซอฟต์แวร์ที่สามารถทำงานได้จริง ตรงตามความต้องการของผู้ใช้

4.1 การสืบทอด (Inheritance)

หลักการสำคัญอย่างหนึ่งของการพัฒนาโปรแกรมเชิงวัตถุ คือ การนำเอาโปรแกรมที่มีอยู่แล้วกลับมาใช้อีกเรียกว่า Code Reusability หรือ Program Reusability และเป็นหัวใจหลักที่ทำให้การพัฒนาโปรแกรมเป็นไปอย่างรวด การสร้างคลาสใหม่ขึ้นมาด้วยการสืบสกุลจากคลาสที่มีอยู่เดิมทำให้โปรแกรมที่พัฒนามีความยืดหยุ่นสูง การสืบสกุลของคลาสในการพัฒนาโปรแกรมเชิงวัตถุ คือ การยอมให้ออบเจ็กต์สามารถสืบทอดเอาคุณสมบัติต่างๆที่มีอยู่ในคลาสพื้นฐาน (Primitive class)[3] มาสู่คลาสที่สืบสกุล (Derived class) โดยหลักการของการพัฒนาโปรแกรมเชิงวัตถุนั้น การสืบสกุลมีอยู่สองลักษณะใหญ่ได้แก่

1. **การสืบสกุลแบบชั้นเดียว (Single Inheritance)** จะเป็นการสืบสกุลจากคลาสพื้นฐานเพียงคลาสเดียวเท่านั้น ดังแสดงในรูป 4.1

[3] สำหรับการเรียกชื่อระหว่างออบเจ็กต์ตั้งต้นและออบเจ็กต์ที่ทำการสืบสกุล ในตำราบางเล่มอาจจะใช้ชื่อเรียกที่ต่างกัน เช่น Based Class/Derived Class หรือ Super Class/Sub Class หรือ Parent Class/Child Class ซึ่งแท้จริงแล้วมีความหมายเหมือนกัน

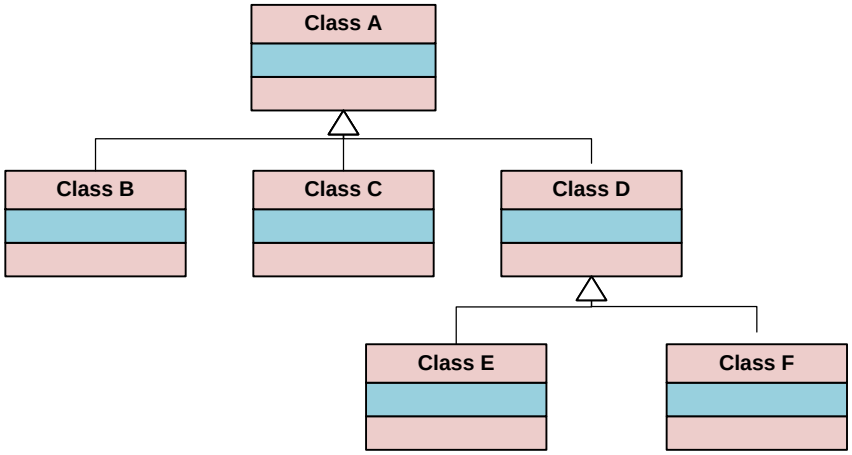


รูป **4.1** การสืบสกุลแบบชั้นเดียว

จากตัวอย่างนี้จะเป็นว่าคลาส B เป็นคลาสที่ถูกสร้างขึ้นมาใหม่โดยสืบสกุลจากคลาสพื้นฐาน A ซึ่งทำให้คลาส B มีสืบทอดเอาคุณสมบัติทุกอย่างทั้งที่เป็นคุณสมบัติและการกระทำของออบเจ็กต์ มาเป็นของตัวเองซึ่งเราสามารถเขียนเป็นภาษาจาวาได้ดังนี้

```
public class A {
       private String data;
        public void setData(String s) {
            data = s;
        }

        public String getData() {
            return (data);
        }
    }

    public class B extends A {
        public void display() {
           setData("Hello World");
           System.out.println(getData());
        }
    }
```

2. การสืบสกุลแบบหลายลำดับชั้น (Multiple Inheritance) ในการสืบสกุลของคลาสภาษาจาวาจะใช้คำสั่ง extends เพื่อบ่งบอกชื่อของซูเปอร์คลาสที่ต้องการจะสืบสกุล จะเห็นได้จากตัวอย่างนี้ คลาส B ที่สืบสกุลจากคลาส A สามารถเรียกใช้ฟังก์ชันต่างๆ ที่มีอยู่ในคลาส A ได้เหมือนกับว่าเป็นฟังก์ชันของตัวเอง สำหรับการสืบสกุลนี้คลาสที่ทำหน้าที่เป็นสับคลาส ก็สามารถเป็นซูเปอร์คลาสให้กับคลาสอื่นๆ ต่อไปได้ จะเห็นได้จากตัวอย่างในรูป 4.2



รูป **4.2** การสืบสกุลแบบหลายลำดับชั้น

คลาส A นั้นเป็นซูเปอร์คลาสของ คลาส B, C และ D และในชั้นถัดลงไปคลาส D ก็เป็นซูเปอร์คลาสสำหรับคลาส E และ F ได้อีกด้วย ลักษณะของการสืบสกุลแบบที่ผ่านมา เราเรียกว่าเป็นการสืบสกุลแบบชั้นเดียวเนื่องจากเป็นการสืบสกุลจากซูเปอร์คลาสเพียงคลาสเดียวเท่านั้น แต่ในหลักการของการพัฒนาโปรแกรมเชิงวัตถุนั้นยอมให้มีการสืบสกุลจากซูเปอร์คลาสหลายๆ ตัวพร้อมกันก็ได้เรียกว่า Multiple inheritance อย่างไรก็ตามการสืบสกุลแบบ Multiple inheritance นี้จะสามารถทำได้ในบางภาษาเท่านั้นอย่างเช่น ภาษา C++ แต่สำหรับภาษาจาวาแล้วจะไม่ยอมให้มีการสืบสกุลแบบหลายลำดับชั้นเนื่องจากในกรณีที่ซูเปอร์คลาสทั้งสองคลาสมีสมาชิกที่มีชื่อเดียวกัน เวลาสืบสกุลอาจทำให้เกิดความสับสนว่าสมาชิกตัวนี้ควรเป็นของคลาสใด ดังนั้นเพื่อลดความสับสนในการเขียนโปรแกรม ดังนั้นภาษาจาวาจึงไม่สนับสนุนการสืบสกุลจากหลายคลาส นั่นก็คือ จะสามารถสืบสกุลได้จากซุปเปอร์คลาสเพียงคลาสเดียวเท่านั้น สำหรับการออกแบบโครงสร้างของคลาสเป็นแบบสืบสกุลนั้นนิยมใช้รูปแบบของลำดับชั้น (hierarchy) การออกแบบคลาสโดยวิธีการสืบสกุลจะช่วยลดปัญหาของการเขียนโค๊ดซ้ำซ้อน ซึ่งเป็นแนวคิดที่สำคัญอย่างหนึ่งของการพัฒนาเชิงวัตถุคือ หลักการของการนำเอาโปรแกรมกลับมาใช้ใหม่

4.1.2 ความสัมพันธ์ระหว่างออบเจ็กต์

การวิเคราะห์ออบเจ็กต์โมเดลเชิงนามธรรมนั้น ระบบ (System) หนึ่งระบบนั้นประกอบด้วยออบเจ็กต์หลายๆออบเจ็กต์ที่สามารถทำงานร่วมกันได้อย่างเป็นระบบ โดยที่ออบเจ็กต์แต่ละออบเจ็กต์นั้นมีบทบาทเป็นของตัวเองภายในระบบ เรียกว่า “Object Role” ตัวอย่างเช่นรถยนต์ นั้นประกอบไปด้วยเครื่องยนต์ ระบบไฟฟ้า ระบบขับเคลื่อน หรือแม้กระทั่งอุปกรณ์และเครื่องอำนวยความสะดวกต่างๆ ภายในซึ่งหากเราวิเคราะห์ระบบที่เรียกว่ารถยนต์องค์ประกอบต่างๆ อาจจะมองได้ว่าเป็นออบเจ็กต์ ที่มีความเป็นอิสระต่อกัน สามารถถูกผลิตโดยต่างบริษัท ต่างสถานที่ แต่องค์ประกอบเหล่านี้

มีบทบาทการทำงานที่แยกออกจากกันชัดเจนเมื่อนำมาเชื่อมต่อกันเพื่อประกอบกันเป็นรถยนต์ 1 คัน ทั้งนี้ในการวิเคราะห์หาความสัมพันธ์ของออบเจ็กต์เราจะมองความสัมพันธ์ออกเป็น 2 ลักษณะคือ

1. ความสัมพันธ์แบบ Is-A Relationship

ความสัมพันธ์แบบ Is-A นี้เป็นความสัมพันธ์ที่จัดกลุ่มออบเจ็กต์ให้เป็นสมาชิกหรือเครือญาติเดียวกัน ตัวอย่างเช่น ความสัมพันธ์แบบการสืบทอด/สืบสกุล ซึ่งเป็นความสัมพันธ์แบบ Parent-Child ดังเช่นตัวอย่างนี้ จะเห็นว่าออบเจ็กต์รถ รถยนต์ รถบรรทุก รถบัส ทั้ง 4 ออบเจ็กต์นี้มีความสัมพันธ์แบบ Is-A คือรถยนต์ รถบรรทุก และรถบัสจัดว่าเป็นรถประเภทหนึ่ง แต่มีคุณสมบัติพิเศษที่มีอยู่ในตัวรถทั่วไป รถตู้เองก็จัดว่าเป็นรถและรถบัสประเภทหนึ่ง ซึ่งมีคุณสมบัติพิเศษที่มีรายละเอียดปลีกย่อยลงไปอีก ดังนั้นเราจะเห็นว่าความสัมพันธ์ที่เป็นแบบการสืบทอดนั้นจะเป็นแบบ Is-A Relationship หรือบางครั้งเราเรียกความสัมพันธ์แบบ Generalization and Specialization เนื่องจากเมื่อมีการถ่ายทอดคุณสมบัติจาก Parent มาสู่ Child แล้วคุณสมบัติของคลาสที่สืบทอดก็จะมีความเป็นเฉพาะมากขึ้น หรือในทางกลับการคลาสที่ทำหน้าที่เป็น Parent ก็จะมีคุณสมบัติที่เป็นกลาง General มากขึ้น

2. ความสัมพันธ์แบบ Has-A Relationship

ความสัมพันธ์แบบ Has-A เป็นลักษณะของออบเจ็กต์ที่เป็นองค์ประกอบของอีกออบเจ็กต์หนึ่ง ซึ่งในที่นี้เราอาจจะเรียกว่าออบเจ็กต์ A ประกอบไปด้วยออบเจ็กต์ B และ C เพื่อทำให้ออบเจ็กต์ A มีความสมบูรณ์และสามารถทำงานได้ ดังตัวอย่างในเรื่องของรถยนต์ที่ได้กล่าวมาแล้ว ว่ารถยนต์แต่ละคันจะต้องประกอบด้วยชิ้นส่วนอะไรบ้างเพื่อทำให้รถนั้นคุณลักษณะและการทำงานที่เป็นรถยนต์อย่างสมบูรณ์ ซึ่งเราสามารถจำแนกชนิดของของบทบาทความสัมพันธ์ในรายละเอียดได้ดังนี้

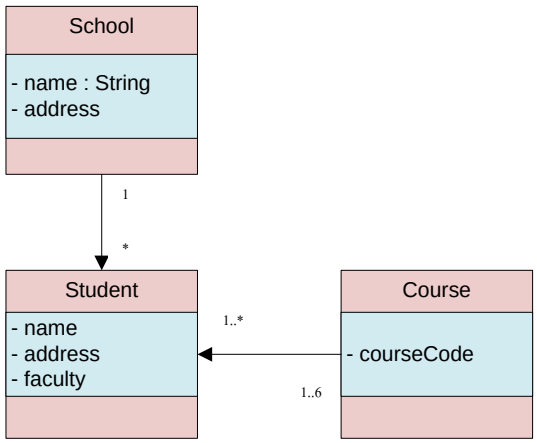
- ดีกรีของความสัมพันธ์ระหว่างออบเจ็กต์ (Multiplicity)

เป็นการระบุในรายละเอียดเพื่อแสดงให้เห็นถึงดีกรีของความสัมพันธ์ระหว่างออบเจ็กต์ว่าสัมพันธ์กันในระดับไหน เช่น นักเรียน 1 คน สามารถลงทะเบียนเรียนได้หลายวิชา หรืออาจารย์หลายคนสามารถร่วมกันสอนในกระบวนวิชาหลายวิชาได้อีกด้วย โดยปกติแล้วเราจะใช้สัญลักษณ์ที่แสดงถึง Multiplicity ได้ดังนี้

1..1	เป็นการแสดงถึงดีกรีความสัมพันธ์แบบหนึ่งต่อหนึ่ง
1..*	เป็นการแสดงถึงดีกรีความสัมพันธ์แบบหนึ่งต่อหลาย
..	เป็นการแสดงถึงดีกรีความสัมพันธ์แบบหลายต่อหลาย

- พาธความสัมพันธ์ (Navigability)

เป็นการแสดงถึงพาธของความสัมพันธ์ระหว่างออบเจ็กต์ ซึ่งการแสดงผลจะใช้สัญลักษณ์ของเส้นที่มีหัวลูกศรชี้ ไปยังคลาสที่จะทำหน้าที่ในการคงความสัมพันธ์



รูป **4.3** แสดงพาธความสัมพันธ์

- ชนิดความสัมพันธ์ (Type)

เป็นการจำแนกดีกรีของความสัมพันธ์ลงในรายละเอียด ว่าออบเจ็กต์นี้มีความสัมพันธ์กันแบบแนบแน่น Strong Relationship หรือแบบ Weak Relationship ซึ่งเราสามารถจำแนกในรายละเอียดได้ดังนี้

ความสัมพันธ์แบบ Association

เป็นความสัมพันธ์ของออบเจ็กต์ที่แสดงถึงบทบาทที่เกี่ยวข้องระหว่างกันเช่นโรงเรียนหนึ่งประกอบด้วยนักเรียนที่ลงทะเบียนเรียน และนักเรียนลงทะเบียนเรียนสามารถเลือกเรียนในกระบวนวิชาที่เปิดสอน

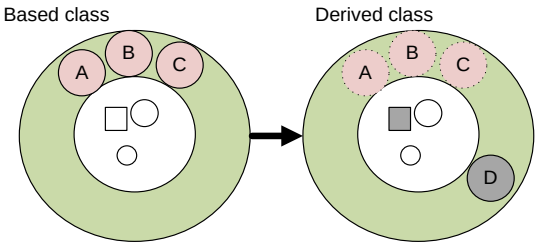
ความสัมพันธ์แบบ Composition

เป็นลักษณะของความสัมพันธ์ที่ผูกติดกันและกัน โดยที่เมื่อออบเจ็กต์ใดออบเจ็กต์หนึ่งถูกทำลายไป ออบเจ็กต์จะไม่คงความเป็นคุณสมบัติของออบเจ็กต์อีกต่อไป เช่น ออบเจ็กต์ Polygon ประกอบไปด้วยออบเจ็กต์ Point หลายๆจุด หากออบเจ็กต์ Polygon ถูกทำลายไป ออบเจ็กต์ Point ก็จะถูกทำลายตามไปด้วย

ความสัมพันธ์แบบ Aggregation

เป็นความสัมพันธ์ที่คล้ายกับ Composition แต่มีดีกรีของความสัมพันธ์ที่ไม่ผูกติดหรือยึดติดต่อกัน และต่างออบเจ็กต์ก็สามารถแยกออกจากกันอย่างเป็นอิสระ คุณสมบัติของการสืบสกุลประกอบด้วยหลักการของ Base Class / Derived Class, Abstraction และ is-a relationship ซึ่งหลักการต่างๆ เหล่านี้จะช่วยให้การออกแบบโครงสร้างโปรแกรมแบบออบเจ็กต์โอเรียนเท็ดทำได้ง่ายขึ้น การสืบสกุลคือ คุณสมบัติที่คลาสหนึ่งๆ สามารถสืบทอดเอาคุณสมบัติทั้ง attribute และ method

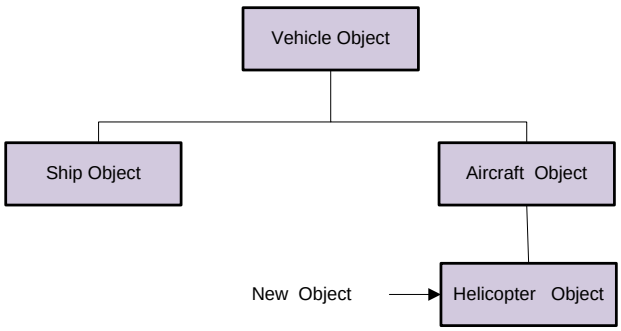
ของ อีกคลาส หนึ่งได้ การทำเช่นนี้ทำให้สามารถ สร้างคลาส ขึ้นมาให้โดยนำสาระสำคัญที่เหมือนกันของ attribute และ behavior (method) จาก class อื่นมาใช้ได้ ดังรูป 4.4



รูป **4.4** การสืบสกุลคุณสมบัติต่างๆจากซูเปอร์คลาสไปยังสับคลาส

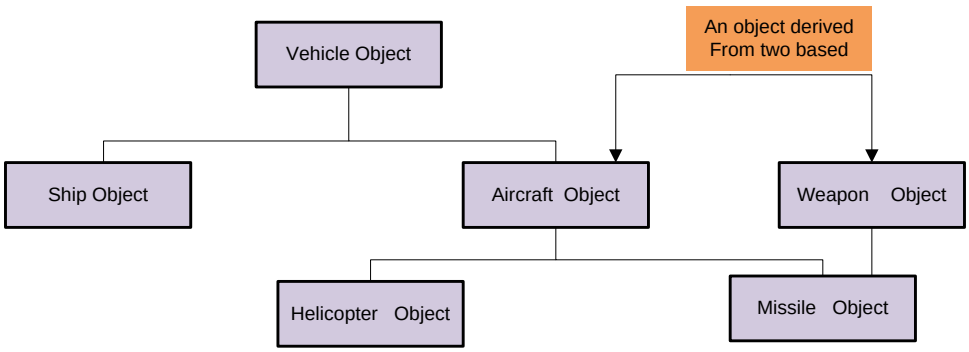
จากรูปข้างต้น A, B และ C เป็นเมทธอดที่ถูกสร้างขึ้นในซูเปอร์คลาส ซึ่งเมื่อมีการสืบสกุลคุณสมบัติต่างๆ ทั้งที่เป็นคุณสมบัติ (Attribute) และการกระทำ (Behavior) ของซูเปอร์คลาสจะถูกถ่ายทอดไปยังสับคลาส ส่วนเมทธอด D เป็นเมทธอดที่ subclass สร้างขึ้นเพื่อแสดงลักษณะความเป็นเฉพาะของตัวเอง ตามลักษณะการสืบสกุลโดยทั่วไปจะมี 2 แบบ คือ

แบบที่ 1 คือลักษณะของการสร้างเป็นออบเจ็กต์ใหม่ 1 ออบเจ็กต์ ถูกสร้างขึ้นมามีการสืบทอดลักษณะจากซูเปอร์คลาส เพียง 1 คลาสจะเรียกว่า Single inheritance จะเห็นได้ว่า object Helicopter จะสืบทอด ลักษณะมาจาก object Aircraft เพียงออบเจ็กต์เดียว ดังรูป 4.5



รูป **4.5** การสืบสกุลแบบชั้นเดียว

แบบที่ 2 บาง object จะมีการสืบทอดลักษณะจาก class แม่ 2 class ขึ้นไปซึ่งจะเรียกว่า “multiple inheritance” ดังรูป 4.6



รูป **4.6** การสืบสกุลจากหลายลำดับชั้น

เพราะฉะนั้น Single inheritance หมายถึง การที่ออบเจ็กต์หนึ่ง ๆ ได้รับการสืบทอดลักษณะจากออบเจ็กต์อื่น เพียง 1 ออบเจ็กต์ Multiple inheritance หมายถึง การที่ออบเจ็กต์หนึ่งๆ ได้รับการสืบทอดลักษณะจาก ออบเจ็กต์อื่นมากกว่า 1 ออบเจ็กต์ จากรูปจะเห็นว่าออบเจ็กต์ missile จะเกิดจากออบเจ็กต์ aircraft และ ออบเจ็กต์ weapon การสืบทอดลักษณะแบบนี้มีข้อควรระวังคือ ชื่อของ attribute ในแต่ละออบเจ็กต์ที่ได้รับการสืบทอดมา ถ้าชื่อซ้ำกันก็จะเกิดความกำกวมขึ้น ซึ่งควรหลีกเลี่ยงการตั้งชื่อซ้ำกันในแต่ละออบเจ็กต์

หลักของการสืบสกุลจะคำนึงถึงความสัมพันธ์ระหว่างคลาส (Superclass และ subclass) ในการถ่ายทอดความสัมพันธ์ของ attribute และ method เป็นหลัก ซึ่งมีรูปแบบความสัมพันธ์เป็น is-a relationship เช่น poodle is a dog เป็นต้น ส่วนคอมโพซิชันจะไม่คำนึงถึงความสัมพันธ์ในลักษณะของการถ่ายทอด แต่จะมองความสัมพันธ์ของออบเจ็กต์ในลักษณะของส่วนประกอบระหว่างกัน ในรูปแบบความสัมพันธ์เป็น has-a relationship เช่น car has an engine หมายความว่า car จะมี engine เป็นส่วนประกอบหนึ่ง โดยที่ทั้ง engine และ car มีลักษณะการทำงานเป็นของตนเองไม่มีความสัมพันธ์กันในลักษณะการถ่ายทอดคุณสมบัติใดๆ นอกจากนี้ออบเจ็กต์ engine ยังอาจประกอบไปด้วยออบเจ็กต์ย่อยๆ ได้อีกหลายออบเจ็กต์เช่น ออบเจ็กต์ piston หรือ oilTank เป็นต้น

ในการออกแบบทั้งสองวิธีนี้ โดยส่วนใหญ่แล้ววิธีคอมโพซิชั่น (Composition) มักจะได้รับความนิยมมากกว่าวิธีของการสืบทอดสืบสกุลเนื่องจากความซับซ้อนของเทคนิคการสืบสกุลที่สามารถสร้างความสับสนให้กับผู้ที่ไม่เชี่ยวชาญในการออกแบบได้ แต่วิธีการออกแบบด้วยเทคนิคการสืบสกุลก็มีข้อดีอยู่ไม่น้อย ถ้าสามารถนำไปใช้ได้อย่างถูกวิธี ก็จะเป็นประโยชน์อย่างมากต่อการเขียนโปรแกรมต่อไปด้วย สรุปได้ว่าสืบสกุล คือ การสืบทอดลักษณะทั้งหมดของคุณสมบัติและเมทธอดจากซูเปอร์คลาสไปยังสับคลาส ซึ่งหลักแนวคิดในการออกแบบออบเจ็กต์ด้วยเทคนิคการสืบสกุลจะเป็นไปตามขั้นตอนต่อไปนี้

- การหาคลาสของระบบ ให้หาความสัมพันธ์จากคลาสที่มีอยู่ก่อน โดยคลาสที่เกิดใหม่จะสืบทอดลักษณะจาก คลาสเดิม

- มองระบบให้ประกอบด้วยระดับชั้น (hierarchy) ของคลาสต่างๆโดยที่ในระดับเดียวกันของการสืบทอดลักษณะจากซูเปอร์คลาสแต่ละคลาสจะต้องมีพฤติกรรมที่ต่างกันอย่างสิ้นเชิง
- การเชื่อมความสัมพันธ์ จะเริ่มจากคลาสแม่เป็นหลัก และเชื่อมความสัมพันธ์กับคลาสลูกไปเรื่อยๆ ในลักษณะต้นไม้ (tree)

ข้อดีการคุณสมบัติการถ่ายทอด	ข้อเสียของคุณสมบัติการถ่ายทอด
1. ทำให้ประหยัดเวลาในการพัฒนาโปรแกรมและการทดสอบโปรแกรม ซึ่งหมายถึงส่วนของโปรแกรมที่พัฒนาไปแล้วหรือใช้อยู่ในซูเปอร์คลาส สามารถสืบทอดมายังสับคลาสได้ โดยไม่จำเป็นต้องเขียนใหม่ และไม่ต้องทำการทดสอบใหม่อีกครั้งด้วย	1. กรณีที่โครงสร้างของการสืบทอดไม่ได้ถูกควบคุมการใช้งานอย่างเข้มงวด การเปลี่ยนแปลงแก้ไขในซูเปอร์คลาสย่อมมีผลกระทบต่อสับคลาสที่มีการถ่ายทอดเอาคุณสมบัติต่างๆไปด้วย
2. การแก้ไขเพิ่มเติมสามารถทำได้ที่ซูเปอร์คลาสเพียงคลาสเดียวเท่านั้น ไม่จำเป็นต้องทำการแก้ไขในทุกๆสับคลาสที่มีการแก้ไขลักษณะเดียวกัน	2. การปรับโครงสร้างของคลาส ที่อาจจะมีการเพิ่มเติมเข้ามาในอนาคตอาจจะไม่ได้เป็นไปตามที่ได้กำหนดหรือออกแบบไว้ตั้งแต่ต้น
3. การสืบทอดมีโครงสร้างที่เอื้อต่อประโยชน์ของ การใช้งานออบเจ็กต์ได้หลายรูปแบบ (Polymorphism) ซึ่งทำให้เกิดประโยชน์อย่างมากต่อผู้ทำการออกแบบ เพราะจะลดความยุ่งยากและความซับซ้อนของโปรแกรมลงไปได้	3. การออกแบบโครงสร้างที่มีการสืบทอดจำเป็นต้องคำนึงถึงคุณสมบัติใดที่สามารถถ่ายทอดได้ การถ่ายทอดบางส่วน ซึ่งทำให้การวิเคราะห์และออกแบบใช้เวลานาน

4.2 การซ่อนคุณสมบัติของออบเจ็กต์ (Data Encapsulation)

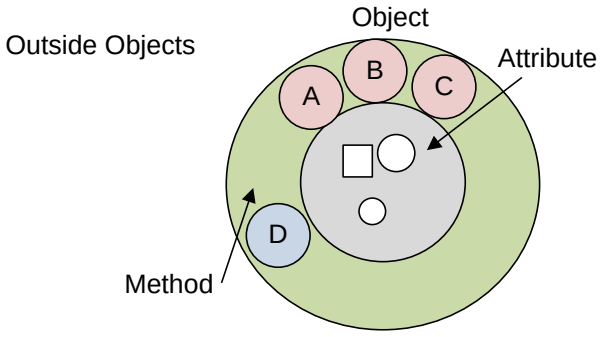
คุณสมบัติที่สำคัญของการทำ Data Encapsulation นั้นก็คือการรวบรวมเอาสิ่งที่เกี่ยวข้องกับขั้นตอนการทำงานเชิงนามธรรมมาจัดกลุ่มกัน โดยที่แบ่งส่วนการทำงานที่สามารถเปิดเผยโดยผ่านทางอินเตอร์เฟส (Interface) หรือสิ่งที่ควรซ่อนรายละเอียดของการทำงานไว้เบื้องหลัง สังเกตว่าคำว่า Encapsulation นั้น มีรากมาจากคำว่า “แคปซูล” หากเรามองว่าแคปซูลยาหนึ่งเม็ด ข้างในอาจประกอบด้วยยาหลายตัว ซึ่งผู้ใช้ยาไม่จำเป็นต้องรู้ แต่สิ่งที่มันทำคือตัวยาเหล่านี้ช่วยรักษาโรคเดียว นั่นคือแนวคิดของการซ่อนและการรวบรวม สำหรับการออกแบบโครงสร้างของโปรแกรมเชิงโครงสร้างนั้น (Procedural Programming Language) มักจะแบ่งโครงสร้างของซอฟต์แวร์ออกเป็นส่วนที่เรียกว่า Interface และส่วนที่เป็น Implementation ซึ่งส่วนที่เป็น Interface นั้นเป็นส่วนที่ยอมให้มีการเข้าถึง

หรือเรียกใช้โดยโมดูลหรือโปรแกรมอื่นได้ และส่วนที่เป็น Implementation นั้นจะซ่อนรายละเอียดของการพัฒนาเอาไว้

ในการพัฒนาซอฟต์แวร์เชิงวัตถุ โครงสร้างของออบเจ็กต์จะถูกนิยามผ่านคลาส โดยคลาสจะเป็นตัวกำหนดโครงสร้างและคุณสมบัติ ที่อธิบายคุณลักษณะของออบเจ็กต์ว่าออบเจ็กต์มีหน้าตาเป็นอย่างไร และส่วนของการกระทำ นั่นคือออบเจ็กต์สามารถทำอะไรได้บ้าง ซึ่งโครงสร้างของคลาสจะแบ่งออกเป็น 2 ส่วนสำคัญคือ Property และ Method นอกจากนี้เรายังสามารถกำหนดขอบเขตของการเข้าถึงและการมองเห็นของคุณสมบัติต่างๆภายในออบเจ็กต์ซึ่งเรียกว่า Access Modifier เราสามารถระบุขอบเขตการเข้าถึงสมาชิกของคลาส ได้แก่ private, protected, public, published

การซ่อนคุณสมบัติของออบเจ็กต์เป็นรากฐานอย่างหนึ่งของแนวความคิดในเชิงออบเจ็กต์โอเรียนเท็ดซึ่งข้อดีคือ การป้องกันคุณสมบัติต่างๆ ของออบเจ็กต์จากความเสียหาย เพราะหากส่วนของโปรแกรมทั้งหมด อนุญาติให้มีการเข้าถึงข้อมูลที่จัดเก็บในแต่ละออบเจ็กต์ได้ตามที่ต้องการแล้วนั้นจะส่งผลให้คุณสมบัติง่ายต่อการถูกใช้อย่างผิดๆ หรือทำให้ค่าเปลี่ยนแปลงไปซึ่งก่อให้เกิดความเสียหายตามมา การซ่อนคุณสมบัติของออบเจ็กต์จะทำหน้าที่ป้องกันไม่ให้ออบเจ็กต์อื่นที่อยู่ภายนอกเข้าถึงออบเจ็กต์หนึ่งๆได้อย่างอิสระจะมีเฉพาะเมทธอดที่อยู่ในออบเจ็กต์ เท่านั้นที่จะสามารถเชื่อมต่อกับคุณสมบัติที่อยู่ในออบเจ็กต์เดียวกันได้เรียกได้ว่าการทำ Data Encapsulation มีคุณสมบัติของการซ่อนข้อมูล (Information hiding)

คุณสมบัติที่สำคัญของ Information hiding คือ การจำกัดการมองเห็นข้อมูลภายในออบเจ็กต์เช่น การกำหนดคุณลักษณะเป็น public เพื่อให้สามารถเชื่อมต่อกับออบเจ็กต์ภายนอกได้หรือการกำหนดเป็น private เพื่อจำกัดคุณสมบัติให้อยู่ภายในออบเจ็กต์เท่านั้น ข้อดีอีกอย่างของการ Encapsulation คือการรวม attribute และ method ไว้ด้วยกันเป็นหนึ่งออบเจ็กต์ ซึ่งถ้ามีการเปลี่ยนแปลงเกิดขึ้นภายในออบเจ็กต์ หนึ่งก็จะไม่ส่งผลกระทบต่อออบเจ็กต์อื่นมีเพียง method และ attribute ของออบเจ็กต์นั้นเท่านั้น ที่จะได้รับผลกระทบนอกจากนี้แล้วเมทธอดของออบเจ็กต์ ก็สามารถถูกเปลี่ยนแปลงได้อย่างอิสระไม่เกี่ยวกับออบเจ็กต์อื่น ดังรูป 4.7



รูป **4.7** แสดงคุณสมบัติของ **encapsulation**

Information hiding เป็นแนวคิดหนึ่งของ Encapsulation คือ การมองเห็นเฉพาะพฤติกรรมที่ให้เห็นว่าทำได้เฉพาะ operation ที่อนุญาติ นอกจากนั้นจะทำการเปลี่ยนแปลงหรือเข้าถึงไม่ได้ กล่าวคือการกระทำกับ data จะต้องทำผ่าน operation เท่านั้น เนื่องจากโปรแกรมอาจจะประกอบด้วยหลายๆคลาสและแต่ละคลาส จะมีการกำหนดสิทธิในการเรียกใช้เมทธอดของออบเจ็กต์และไม่ให้มองเห็นส่วนที่เป็นคุณสมบัติของออบเจ็กต์

ในการออกแบบคลาสที่กล่าวมา ออบเจ็กต์หนึ่งๆจะมีคุณสมบัติทั้งส่วนสมาชิกและเมทธอดอีก ทั้งออบเจ็กต์ยังยอมให้เกิดการสืบสกุลระหว่างคลาสพื้นฐานและคลาสสืบสกุลได้อีกด้วย การควบคุมการมองเห็นหรือการเข้าถึงตัวแปรที่เป็นสมาชิกและเมทธอดของออบเจ็กต์จึงถือเป็นเรื่องสำคัญ การควบคุมการมองเห็นนี้เรียกว่า Encapsulation หรือ Information Hiding เป็นการควบคุมว่าจะยอมให้ออบเจ็กต์อื่นๆมองเห็นสมาชิกตัวไหนหรือเมทธอดตัวใดภายในออบเจ็กต์ปัจจุบัน การมองเห็นในที่นี้หมายความว่าสามารถแก้ไขตัวแปรหรือเรียกใช้เมทธอดได้ สำหรับสิทธิของการควบคุมการมองเห็นนั้นจะมีอยู่ 3 ประเภทหลักได้แก่

1. แบบทั่วไป (Public)
2. แบบส่วนตัว (Private)
3. แบบป้องกัน (Protected)

หากวิเคราะห์ถึงแนวคิดของการทำ Encapsulation ในการการพัฒนาโปรแกรมเชิงวัตถุนั้น เราจะเห็นว่าแนวคิดนี้มีคุณสมบัติเด่นที่เอื้อประโยชน์ต่อการพัฒนาโปรแกรมมีรูปแบบ อีกทั้งช่วยให้การพัฒนาซอฟต์แวร์ที่มีขนาดใหญ่ และการพัฒนาแบบทีมเป็นไปอย่างมีระบบ ซึ่งเราสามารถสรุปคุณสมบัติของ Encapsulation ได้ดังนี้

- ความเป็นเอกภาพ (Modularity)

เมื่อเราสามารถแบ่งส่วนโปรแกรมออกเป็นคลาสต่างๆ และในโปรแกรมหลัก ซึ่งมองคลาสเหล่านี้เป็นลักษณะของกล่องดำทำให้ โปรแกรมหลักเขียนง่ายขึ้น ลดความซับซ้อนของโครงสร้างโปรแกรม ซึ่งในลักษณะของการทำงานที่เป็นทีมงานคลาสต่างๆเหล่านี้สามารถนำมารวมกันเป็น คอมโพเนนท์หรือแพคเกจ และสามารถที่จะแยกให้แต่ละทีมงานพัฒนาและตรวจสอบข้อผิดพลาด (Debuggin) และการทดสอบ (Testing) โปรแกรมในแต่ละส่วนได้

- การนำกลับมาใช้งานได้ (Reusability)

เมื่อเรามองคลาสเหล่านี้ในลักษณะของโมดูล หรือคอมโพเนนท์ ที่แยกการทำงานอย่างเป็นอิสระแล้ว เราสามารถที่จะนำเอาโปรแกรมโค๊ดเหล่านี้กลับมาใช้ได้อีก หรือสามารถนำเอาคอมโพเนนท์ต่างๆ มาประกอบกันเป็นโปรแกรมใหม่ได้ ทำให้ลดความซ้ำซ้อนของการพัฒนา และทำให้การพัฒนาซอฟต์แวร์ใหม่ๆสามารถทำได้อย่างรวดเร็ว

- ความเสถียรภาพ (Robustness)

การในประโยชน์จากแนวคิดของ Data Encapsulation ทำให้โปรแกรมโค๊ดที่พัฒนามีความเป็นเอกภาพมากขึ้น การควบคุมการเข้าถึงและการมองเห็นทำให้ลดข้อผิดพลาดที่อาจจะเกิดขึ้นจากการแก้ไขคุณสมบัติของออบเจ็กต์โดยที่ไม่ได้ตั้งใจ หรือกรณีที่ออบเจ็กต์นั้นผ่านการทดสอบและแก้ไขจนไม่มีข้อผิดพลาดเกิดขึ้นแล้ว เราสามารถมั่นใจได้ว่าออบเจ็กต์นั้นสามารถนำไปใช้พัฒนาต่อยอด หรือประกอบกันเป็นซอฟต์แวร์ใหม่ได้

4.2.1 แบบทั่วไป (Public)

การกำหนดสิทธิแบบทั่วไปนั้นเป็นการยอมให้คลาสอื่นๆสามารถมองเห็นตัวแปรและเมทธอดของคลาสปัจจุบันได้ โดยรูปแบบของการกำหนดแบบทั่วไปจะใช้คีย์เวิร์ด public นำหน้าเมื่อประกาศคลาส ประกาศตัวแปร หรือสร้างเมทธอด ซึ่งปรกติแล้วหากไม่มีการระบุสิทธิเริ่มแรก (default) จะถูกกำหนดให้เป็นแบบทั่วไป

```
class CarPart {
     String Model;
     public setModel(String Model) {
         this.Model = Model;
     }
}

class MyProgram {
     public static void main(String args[]) {
         CarPart mycar = new CarPart();
         mycar.Model = "Mazda";   // Accessing data member directly
         mycar.setModel("Mazda");  // Accessing method
     }
}
```

จากตัวอย่างแสดงถึงการเข้าถึงตัวแปร Model และเมทธอด setModel() ที่อยู่ในคลาส CarPart สามารถทำได้จากภายนอกของคลาสในที่นี้เราเรียกใช้จากฟังก์ชัน main() จะเห็นได้ว่าการกำหนดสิทธิแบบทั่วไปของคลาสเป็นการเปิดให้คลาสอื่นๆสามารถมองเห็นข้อมูลและฟังก์ชันการทำงานภายในคลาสได้ ซึ่งถ้าเปรียบเทียบกับการเขียนโปรแกรมโดยทั่วไปที่ไม่ใช่ออบเจ็กต์ก็จะคล้ายกับการใช้งานโครงสร้างของข้อมูลแบบเรคคอร์ด (Record) ที่มีประกอบด้วยฟังก์ชันทำงานที่เกี่ยวข้องกับข้อมูลเรคคอร์ดนั่นเอง การออกแบบออบเจ็กต์โดยให้สิทธิแก่คลาสอื่นๆ เข้ามาจัดการกับข้อมูลของคลาสทำให้การพัฒนาโปรแกรมค่อนข้างง่ายในระดับหนึ่ง แต่เมื่อโปรแกรมมีขนาดใหญ่ขึ้นการป้องกันข้อมูลรวมทั้งการจัดการการเข้าถึงเมทธอดภายในคลาส ควรจะมีการกำหนดสิทธิที่เข้มงวดกว่าแบบทั่วไปซึ่งเราจะกล่าวถึงในหัวเรื่องต่อไป

4.2.2 แบบส่วนตัว (Private)

การกำหนดสิทธิแบบส่วนตัวเป็นการกำหนดให้สมาชิกสามารถมองเห็นได้เฉพาะภายในคลาสเดียวกันเท่านั้น รูปแบบของสิทธิแบบส่วนตัวนั้นจะใช้คีย์เวิร์ด private เป็นตัวกำหนด จุดประสงค์ก็เพื่อ

เป็นการซ่อนเมทธอดเท่านั้น ในกรณีนี้เมทธอดที่นิยมสร้างในออบเจ็กต์ได้แก่ เมทธอด set() และ get() โดย set() จะเป็นเมทธอดที่ทำหน้าที่ในการกำหนดค่าให้กับตัวแปรส่วนตัว โดยค่าจะมาจากการส่งเป็นอาร์กิวเมนต์ ส่วนเมทธอด get() ก็จะทำหน้าที่ในการส่งค่าของตัวแปรส่วนตัวกลับไปให้เมทธอดที่เรียกใช้ (caller function)

```
public class Circle {
    private float radius;       //private attribute which can only be
                                //seen within Circle class
    public void setRadius(float r) {
        this.radius = r;
    }

    public float getRadius() {
        return (radius);

   }
}
```

ในทำนองเดียวกัน การประกาศสิทธิแบบส่วนตัวก็ยังสามารถใช้กับเมทธอดของคลาสได้อีกด้วย และถ้าในกรณีนี้ เมทธอดที่เป็นแบบส่วนตัวจะสามารถถูกเรียกใช้ได้จากเฉพาะภายในคลาสเดียวกันเท่านั้น

```
class Circle {
    final float PI = 3.141592;
    private float radius = 50;
    private float area;

    private boolean checkRadius() {
         boolean validRadius = false;
         if ((radius > 0) && (radius <= 100))
             validRadius = true;
             return (validRadius);
         }
 }
     public float calculateArea() {
         if (checkRadius())
         return (radius * PI * PI);
     }
 }

 class MyProgram {
      public static void main(String args[]) {
         Circle c = new Circle();
         boolean validRadius = c.checkRadius();      // Error
         float area = calculateArea();               // OK
      }
 }
```

จากตัวอย่างนี้จะเห็นว่าเมทธอด checkRadius() ประกาศเป็นแบบส่วนตัว และเรียกใช้โดยเมทธอด calculateArea() ซึ่งอยู่ในคลาสเดียวกัน แต่ถ้าเรียกใช้จากฟังก์ชัน main() ภายในคลาส MyProgram ก็จะเกิดข้อผิดพลาดขึ้นเพราะถือว่าไม่ได้อยู่ในคลาสเดียวกัน

4.2.3 แบบป้องกัน (Protected)

การกำหนดสิทธิแบบป้องกันนี้จะใช้คีย์เวิร์ด protected นำหน้าชื่อตัวแปรหรือชื่อเมทธอด ลักษณะการทำงานก็จะคล้ายกับ 2 แบบที่ผ่านมาเพียงแต่ว่าตัวแปรและเมทธอดที่เป็นแบบป้องกันนี้จะสามารถมองเห็นได้เฉพาะภายในคลาสตัวเองและคลาสที่สืบสกุลเท่านั้น

```
class A {
       protected int number = 0;
       protected void displayNumber() {
            System.out.println("Number is : " = number);
       }
 }

 class B extends A {
        void display()
             number = 45;
             displayNumber();
        }
 }
```

4.3 การใช้งานได้หลายรูปแบบ (Polymorphism)

คำว่า Poymorphism เป็นคำมาจากภาษากรีกแปลว่า “Having many shapes” หมายถึง การที่คลาสหนึ่งๆ สามารถแปรเปลี่ยนไปได้หลายรูปร่างขึ้นอยู่กับสภาพแวดล้อมหรือสถานการณ์ในขณะนั้น ตัวอย่างการทำงานของเครื่องคอมพิวเตอร์ที่มีปุ่ม F1 เพื่อแสดงถึงความต้องการช่วยเหลือ ซึ่งปุ่ม F1 ที่เรากดในแต่ละครั้งอาจจะได้รับผลลัพธ์ของการช่วยเหลือไม่เหมือนกัน ทั้งนี้จะขึ้นอยู่กับว่า ณ ขณะนั้นเรากำลังทำงานอยู่กับโปรแกรมอะไร หรือว่าอยู่ในหน้าจอที่เกี่ยวข้องกับเรื่องใด ทั้งนี้เราจะเห็นว่าหลักการของ โพลีมอร์ฟิซึมในออบเจ็กต์พยายามที่จะทำให้การใช้งานระบบคอมพิวเตอร์สะดวกต่อผู้ใช้งาน การกดปุ่ม F1 แต่ละครั้ง ผู้ใช้งานแทบจะไม่ต้องรู้ในรายละเอียดเลยว่า F1 ทำงานอย่างไรในแต่ละโปรแกรม

ลักษณะเดียวกันกับการใช้งานคลาสหรือออบเจ็กต์ การเรียกใช้งานเมทธอดที่มีชื่อเดียวกัน แต่การทำงานอาจจะไม่เหมือนกันซึ่งจำแนกตามสถานะของออบเจ็กต์ (Object's state) ณ ขณะนั้น ตามชนิดของข้อมูล หรือ Massage ที่ฟังก์ชันจะได้รับขณะทำงาน เมื่อฟังก์ชันได้รับข้อมูลที่ต้องการแล้วก็จะไปเลือกฟังก์ชันที่เหมาะสมกับข้อมูลนั้นๆ ทำงานต่อไป

หลักการทำงานของโพลีมอร์ฟิซึมจะเป็นใช้งานเมทธอดที่เขียนทับ Overridden method จากคลาสที่มีการสืบทอด ซึ่งโปรแกรมจะทำการเลือกเมทธอดที่เหมาะสมเพื่อทำงานในขณะที่มีการเรียกใช้งานเท่านั้น โดยอาศัยเทคนิคเรียกว่า Binding ซึ่งหมายถึง การเชื่อมต่อระหว่าง Method call (คำสั่งที่

ใช้เรียก Method มาทำงาน) กับ Method body (คำสั่งต่างๆ ภายใน method) ถ้า Binding ถูกทำงานขณะที่รันโปรแกรม (โดย compiler) จะเรียกว่า “early binding” แต่เทคนิคของการเลือกใช้เมทธอดขณะที่รันโปรแกรมว่า “Dynamic Binding” หรือ “Late binding” หรือ “Run-time binding” หรืออาจเรียกอีกอย่างหนึ่งตามลักษณะของออบเจ็กต์ โพลีมอร์ฟิซึมถือได้ว่าเป็นความสามารถในการส่ง Massage เดียวกันให้กับออบเจ็กต์ของซูเปอร์คลาสหรือสับคลาสเพื่อทำการตอบสนองต่อ Massage อันเดียวกันในลักษณะที่แตกต่างกันที่เหมาะสมกับคลาสและสถานะในขณะนั้นและเรียกใช้เมทธอดได้อย่างถูกต้อง โดยมีกลไกในการตรวจสอบ กล่าวโดยสรุปก็คือ คอมไพเลอร์จะไม่รู้ว่าออบเจ็กต์นั้นเป็นออบเจ็กต์ชนิดใดล่วงหน้า แต่จะตรวจสอบและเรียกใช้ Method body ที่ถูกต้องมาทำงานโดยอัตโนมัติ

โพลีมอร์ฟิซึมทำให้ออบเจ็กต์แต่ละออบเจ็กต์ ตอบสนองต่อ Massage เดียวกันในลักษณะที่เหมาะสมกับคลาสของตัวเอง เช่น DisplayInformation() ถูกนิยามโดยให้แสดงค่าของข้อมูลบางตัวของ ออบเจ็กต์ ทำให้ DisplayInformation นี้สามารถส่งให้ออบเจ็กต์ทุกๆ ออบเจ็กต์ของซูเปอร์คลาสหรือสับคลาส ที่ทำให้ออบเจ็กต์นั้นรู้จัก DisplayInformation และออบเจ็กต์ของคลาสที่ต่างกันจะตอบสนองต่อเมสเสจต่างกันไป ซึ่งความสามารถในการใช้เมสเสจที่เหมือนกันส่งให้ออบเจ็กต์ต่างชนิดกันได้สามารถเปรียบเทียบได้กับความคิดของมนุษย์คือไม่เป็นธรรมชาติ เนื่องจากมนุษย์เรามีอาการตอบสนองกับสิ่งต่างๆ ที่เข้ามาในที่ไม่เหมือนกัน ขึ้นอยู่กับสถานการณ์หรือสภาพแวดล้อมที่ต่างกัน

บทที่ 5 – ภาษาเชิงวัตถุ

ปัจจุบันโปรแกรมภาษาที่รองรับการทำงานแบบออบเจ็กต์มีอยู่หลายภาษาด้วยกัน แต่ละภาษาต่างๆ มีข้อดีและข้อด้อยที่แตกต่างกัน ซึ่งในบทนี้จะกล่าวถึงภาษาคอมพิวเตอร์ที่ใช้กันอยู่ทั่วไป เพื่อให้เห็นถึงว่าแต่ละภาษาได้มีการนำเอาแนวคิดของการพัฒนาโปรแกรมเชิงวัตถุที่ได้กล่าวไปแล้ว การพัฒนางานเชิงวัตถุนั้นเป็นการนำเอาสิ่งที่มีอยู่แล้วนำมาใช้ประโยชน์ ปัจจุบันภาษาจาวาเป็นภาษาหนึ่งที่สนับสนุนการพัฒนาในเชิงวัตถุได้เป็นอย่างดี เราสามารถใช้รูปแบบการเขียนโปรแกรมแบบภาษาจาวารวมไปถึงกลุ่มคลาสที่จาวามีให้เราใช้ได้ บางภาษาเป็นภาษาที่มีอยู่เดิมแต่ได้มีการพัฒนาโดยเพิ่มคุณสมบัติทางด้านออบเจ็กต์เข้าไปเพื่อให้สามารถรองรับการทำงานเชิงวัตถุได้ ตัวอย่างเช่นภาษา Delphi ที่มีการพัฒนามาจากภาษา Object Pascal (Pascal + OOP) ซึ่งจัดเป็นภาษาที่ใช้เขียนโปรแกรมแบบโครงสร้างเชิงวัตถุ (Object-Oriented Programming Language) และ Delphi มีลักษณะเป็น Hybrid OOP Language เหมือนกับ C++ คือ นอกจากการสร้างหรือนิยามคลาสแล้วยัง สามารถประกาศ ตัวแปร และ สร้าง procedure หรือ function ได้ตามปกติ ซึ่งต่างจากภาษาจาวา ที่เป็นลักษณะ Pure OOP Language มีการบังคับว่าการประกาศตัวแปรหรือการสร้างฟังก์ชันจะต้องกระทำภายในคลาสเท่านั้น ปัจจุบัน Delphi สามารถทำงานได้ทั้งบนระบบปฏิบัติการ Windows และ Linux โดยบนระบบ Linux จะใช้ชื่อว่า Kylix ภาษา Perl เองก็เช่นกันที่ได้มีการพัฒนาความสามารถของตนเองเพื่อให้สามารถทำงานเชิงวัตถุได้ ซึ่งเราเรียกลักษณะของภาษาว่า Object Perl

ภาษา C# (อ่านว่า ซี-ชาร์ป) เป็นภาษาเชิงวัตถุเช่นเดียวกับ C++ และ Java C# เป็นภาษาเชิงวัตถุที่ทันสมัย ช่วยให้นักพัฒนาสามารถสร้างแอพพลิเคชั่นหลายๆ รูปแบบในแพลตฟอร์ม .NET ของไมโครซอฟท์ ซึ่งเป็นแพลตฟอร์มที่มีเครื่องมือ และไลบรารี ที่สนับสนุนการพัฒนาอย่างมากมายทั้งการประมวลผลและเทคโนโลยีใหม่ๆที่รองรับภาษานี้ ความง่ายของภาษา Visual Basic ที่ C# ได้รับมาคือ สามารถสร้าง Windows Form ด้วยภาษา C# ได้ง่ายพอๆ กับการเขียนฟอร์มด้วยภาษา Visual Basic รวมไปจนถึงการสร้างคอมโพเนนท์ก็สามารถทำได้จากการสร้างเป็น Package และเนื่องจากภาษา C# ได้พัฒนามาจากภาษา C++ ดังนั้นโปรแกรมจึงเขียนด้วยภาษา C++ ไวยากรณ์ของภาษา C# มีความคล้ายคลึงกับภาษาจาวามาก รวมไปจนถึงแนวคิดก็มีลักษณะคล้ายๆ กัน (เพราะเป็นภาษาแบบ OOP เหมือน) ทำให้สามารถเรียนรู้ภาษา C# ได้เร็ว โดยทั้งภาษา C# และ Java เหมือนตรงที่มีพื้นฐานมาจากภาษา C, C++ และผนวกกับคุณสมบัติที่สำคัญของออบเจ็กต์มาด้วย เช่น garbage collection, exception handing เป็นต้น ซึ่งภาษาที่รองรับการพัฒนาอย่างรวดเร็วนี้เรียกว่าการพัฒนาแบบวิชวลโปรแกรม (Visual Programming) ที่ช่วยให้สามารถพัฒนาซอฟต์แวร์ได้อย่างรวดเร็วและเพิ่มประสิทธิภาพต่อการพัฒนาต้นแบบ ทำให้ผู้ใช้สามารถพัฒนาโปรแกรมในส่วนที่ติดต่อกับผู้ใช้

(User Interface) ได้อย่างรวดเร็ว ซึ่งซอฟต์แวร์ที่สนับสนุนงานประเภทนี้เช่น Microsoft Visual Studio, Borland Suite, Delphi และ Netbean

5.1 เครื่องมือพัฒนาที่สนับสนุนการพัฒนาเชิงวัตถุ

ปัจจุบันเครื่องมือที่สนับสนุนการพัฒนาซอฟต์แวร์เชิงวัตถุมีการพัฒนาการและมีประสิทธิภาพมากกว่าในอดีต เครื่องมือเหล่านี้เรียกว่า IDE (Integrated Development Environment) นั่นก็คือ Program Editor ในสมัยก่อน แต่ปัจจุบันเครื่องมือเหล่านี้ได้มีพัฒนาการไปอย่างทันสมัยและมีความฉลาดในตัวเองเพื่อที่จะทำให้นักพัฒนาโปรแกรมสามารถที่จะพัฒนาซอฟต์แวร์ได้อย่างรวดเร็ว

5.1.1 เครื่องมือสำหรับพัฒนา Visual Programming

เครื่องมือเหล่านี้สนับสนุนการทำงานที่เป็นแบบวิชวล เราเรียกว่า Visual Programming คือสามารถมองเห็นผลลัพธ์ของซอฟต์แวร์ขณะที่ทำการออกแบบหรือระหว่างการพัฒนา และสนับสนุนการพัฒนาโปรแกรมโค๊ดที่ซ่อนอยู่เบื้องหลังในลักษณะที่เป็น Code Behind เครื่องมือเหล่านี้ทำงานโดยรวบรวมเครื่องมือต่างๆที่จำเป็นสำหรับการพัฒนาโปรแกรม เช่น Visual Tools, Editor, Compiler, Linker, Debugger ฯลฯ เข้าด้วยกัน และทุกอย่างสามารถทำได้จากการลากและวาง (Drag and Drop) การพัฒนาโปรแกรมที่เป็นลักษณะของ Project Base ที่เครื่องมือเหล่านี้จะทำการเตรียมสภาวะแวดล้อม หรือโปรแกรมโค๊ดตั้งต้นในการสร้างเค้าโครงของโปรแกรม ประกอบกับเครื่องมือวิซาร์ด (Wizard Tool) ที่ช่วยให้คำแนะนำแก่นักพัฒนา ต่างๆเหล่านี้ ทำให้นักพัฒนาในปัจจุบันได้รับความสะดวกจากเครื่องมือเหล่านี้มาก ซึ่งโปรแกรมเหล่านี้จัดอยู่ในประเภท RAD (Rapid Application Development) ซึ่งทำให้ในปัจจุบัน ซอฟต์แวร์สามารถถูกผลิตและพัฒนาออกมาได้อย่างรวดเร็ว

5.1.2 การจัดการกับโมดูลและคอมโพเนนท์

การจัดการกับซอฟต์แวร์ที่มีขนาดกลางถึงขนาดใหญ่โดยที่มีนักพัฒนาโปรแกรมทำงานร่วมกันเป็นทีมนั้นนิยม การออกแบบสถาปัตยกรรมของซอฟต์แวร์นิยมจัดเก็บกลุ่มโปรแกรมที่มีลักษณะการทำงานคล้ายกันอยู่ในกลุ่มเดียวกัน เรียกว่า โมดูลหรือไลบรารี ซึ่งในภาษา C นั้นเราจะเห็นว่ามีการจัดกลุ่มโปรแกรมเป็นไลบรารี่ ในภาษา Delphi นั้น การจัดกลุ่มโปรแกรมจะจัดอยู่ในรูปของ Unit ซึ่งภาษา Pascal มักแยกส่วนของ source program ที่มีการใช้งานร่วมกันออกเป็น unit เพื่อให้สามารถถูกเรียกใช้งานร่วมกันได้ โดยไม่ต้องเขียนใหม่ทุกครั้ง โปรแกรมโค๊ดที่อยู่ใน Unit อาจเป็นการกำหนดค่าคงที่ นิยามชนิดข้อมูล รวมถึงการนิยามคลาสต่างๆ การประกาศตัวแปร Procedure และ Function สิ่งที่อยู่ใน Unit อาจเป็น source program ซึ่งไฟล์นั้นจะมีนามสกุลเป็น .pas หรืออาจเป็น source program ที่ผ่านการคอมไพล์แล้ว ซึ่งไฟล์นั้นจะมีนามสกุลเป็น .dcu (Delphi compiled unit) การเรียกใช้สิ่งที่อยู่ใน unit จะใช้คำสั่ง “uses” เช่น uses MyUnit;

ในลักษณะเดียวกันกับภาษา C# C++ หรือภาษา Java ซึ่งมีไลบรารีขนาดใหญ่มารองรับการพัฒนาซอฟต์แวร์ที่มีขนาดใหญ่ ตัวอย่างเช่น ภาษา C# Visual Basic และ C++ นั้นพัฒนาขึ้นมาบนพื้นฐานของ .NET Framework หรือหากเป็นเวอร์ชั่นก่อนหน้าเทคโนโลยี .NET นี้ทางไมโครซอฟท์เองจะสนับสนุนคลาสไลบรารีขนาดใหญ่ชื่อว่า Microsoft Foundation Class (MFC) ส่วนภาษา Java เองจะจัดไลบรารีเหล่านี้ให้อยู่ในรูปของ Java Package ในการเรียกใช้งานไลบรารีเหล่านี้ก็สามารถทำได้ง่ายขึ้นโดยอาศัยเครื่องมือ IDE ที่อาศัยการลากและวาง หรือเครื่องมือประเภท WYSIWYG (What-You-See-Is-What-You-Get) ที่สามารถช่วยในการพัฒนาโปรแกรมเป็นแบบวิชวลโปรแกรมมิ่ง

เมื่อเราสร้างคลาสขึ้นมาตามความต้องการของระบบ จนกระทั่งจำนวนของคลาสที่พัฒนาเพิ่มขึ้นเรื่อย ในภาษาจาวามีวิธีการจัดการคลาสที่ได้ให้เป็นหมวดหมู่อย่างเป็นระเบียบโดยการกำหนดกลุ่มของคลาสขึ้นมาเรียกว่า แพคเกจ (Package) โดยทำหน้าที่คล้าย Folder หรือ Directory ของระบบคอมพิวเตอร์สำหรับเก็บไฟล์ และในการอ้างถึงคลาสก็จะใช้แพคเกจเป็นตัวอ้างถึงตำแหน่งที่อยู่ของคลาสด้วย เช่น

```
package com.Company;
class Staff {
    int StaffId;
    String StaffName;

    void setStaffId(int staffId){
           this.StaffId = staffId
    }
    …
}
```

จากตัวอย่างนี้หมายถึง เรามีการสร้างคลาส Staff อยู่ในแพคเกจชื่อว่า com.Company เมื่อเราได้กำหนดแพคเกจสำหรับระบบที่พัฒนาแล้วและได้มีการกำหนดคลาสเข้าไปอยู่ในแพคเกจ ที่เหมาะสมแล้วในการเรียกใช้งานสามารถทำได้ดังนี้

```
class Tester{
       public static void main(String agrv[]){
              com.Company.Staff employee = new com.Company.Staff();
              employee.setStaffId(678);
       }
}
```

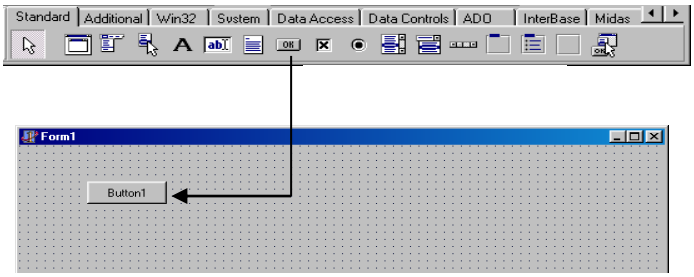
เราสามารถอ้างถึงคลาสผ่านแพคเกจได้ดังคลาสตัวอย่างนี้ com.Company.Staff เป็นการอ้างถึงคลาส Staff ที่อยู่ในแพคเกจ com.Company สำหรับการสร้างไลบรารีขึ้นมาใช้ในภาษา C# นั้นเราเรียกว่า Namespace ซึ่งใช้ในการกำหนดกลุ่มของคลาสให้อยู่ในไลบรารีเดียวกัน และวิธีการเรียกใช้งานไลบรารีจะอาศัยคำสั่ง use <Namespace> โดยที่ระบุชื่อของ Namespace ที่ต้องการเรียกใช้งาน

```
using System;
namespace LibraryA
{
    using LibraryB.lib;
```

```
    class MainProgram {
           public static void Main()
           {
                  SayHello hello = new SayHello();
                  hello.SayIt();
           }
    }
}
Namespace LibraryB.lib
{
    class SayHello
    {
           public void SayIt()
           {
                  Console.WriteLine("Hello, World");
           }
    }
}
```

การพัฒนาโปรแกรมด้วยภาษา Delphi จะใช้ภาษาออบเจ็กต์ Pascal ซึ่งมีหลักการเขียนที่คล้ายคลึงกับภาษาที่สนับสนุนการทำงานเชิงวัตถุอื่นๆ ซึ่งอาศัยการทำงานแบบออบเจ็กต์เป็นหลักโดยแต่ละออบเจ็กต์จะมีองค์ประกอบที่สำคัญคือ ข้อมูล (ตัวแปรภายใน) Data Member และพฤติกรรม (การทำงาน) Method โดยออบเจ็กต์ที่พัฒนาขึ้นจะถูกนำมาประกอบกันเป็นโปรแกรมเพื่อทำงานตามที่ต้องการ สำหรับภาษา Delphi ออบเจ็กต์จะถูกแบ่งออกเป็น 2 กลุ่มคือ ออบเจ็กต์ปกติและคอมโพเนนท์ โดยมีรายละเอียดของข้อแตกต่างระหว่างกันดังนี้

- คอมโพเนนท์จัดได้ว่าเป็นออบเจ็กต์ที่สร้างขึ้นจากคลาสที่ได้ถูกพัฒนาไว้ก่อนหน้าแล้วในไลบรารีซึ่ง Delphi เรียกคลาสเหล่านั้นว่า VCL (Visual Component Library) ส่วนออบเจ็กต์นั้นจะถูกสร้างจากคลาสที่ผู้ใช้สร้างขึ้นเอง
- คอมโพเนนท์เป็นออบเจ็กต์ ที่ถูกสร้างอย่างอัตโนมัติโดยภาษา Delphi เอง เพียงแค่ผู้ใช้ใช้เครื่องมือ IDE ในการลากและวางคอมโพเนนท์จากคอมโพเนนท์ palette มาวางบน Form และเมื่อ Form ถูกทำลายคอมโพเนนท์ก็จะถูกทำลายโดยอัตโนมัติ ส่วนออบเจ็กต์ จะถูกสร้างโดยใช้คำสั่งในขณะที่โปรแกรมทำงาน (Runtime) เท่านั้น และผู้ใช้จะต้องเป็นผู้ทำลายออบเจ็กต์เอง



- โดยเมื่อผู้ใช้งานลากคอมโพเนนท์วางลงบน Form โปรแกรม Delphi จะทำการสร้างคอมโพเนนท์ ให้โดยอัตโนมัติ ซึ่ง Delphi จะทำการสร้างคอมโพเนนท์จากคลาสของคอมโพเนนท์ ที่ถูกสร้างไว้แล้วใน VCL (Visual Component Library) และผู้ใช้สามารถ

กำหนดคุณสมบัติของคอมโพเนนท์ที่สร้างขึ้นในขณะออกแบบโปรแกรม โดยผ่านทางออบเจ็กต์ Inspector

- คอมโพเนนท์ เป็นออบเจ็กต์ที่ผู้ใช้สามารถจัดการได้ในขณะออกแบบ (Design time) โดยผ่านออบเจ็กต์ Inspector และผู้ใช้ก็สามารถจัดการกับคอมโพเนนท์ในขณะที่โปรแกรมทำงานโดยใช้คำสั่งได้ ส่วนออบเจ็กต์จะถูกจัดการโดยใช้คำสั่งในขณะที่โปรแกรมทำงานเท่านั้น

ทั้งออบเจ็กต์และคอมโพเนนท์ต่างก็มีคุณลักษณะเหมือนกับภาษาออบเจ็กต์ทั่วไปคือประกอบด้วย

- Field คือ Data Member ซึ่งเป็นข้อมูลภายในหรือตัวแปรภายในของออบเจ็กต์
- Method คือ พฤติกรรมหรือการทำงานของออบเจ็กต์ (ในรูปของ Procedure หรือ Function)

นอกจากนี้คอมโพเนนท์ใน Delphi ยังประกอบด้วย

- Property คือ กลไกในการเข้าถึงข้อมูลภายในของออบเจ็กต์ทั้งการอ่าน / เขียน (read/write) ทั้งในขณะออกแบบและขณะที่โปรแกรมทำงานซึ่งจริงๆ แล้วก็เป็นการเข้าถึงข้อมูลภายในโดยเรียกผ่านเมทธอดนั่นเองซึ่งตามแนวคิดของออบเจ็กต์แล้ว Property ใน Delphi คือลักษณะของ Method อย่างหนึ่งของคอมโพเนนท์นั่นเอง เพื่อประโยชน์ในการเข้าถึง (Accessor) ข้อมูลภายใน (Field) ของคอมโพเนนท์ซึ่งอาจจะเป็นอ่านข้อมูลจากหรือเป็นการเขียนข้อมูลลงในฟิลด์ (Read/Write) โดยผ่าน Getters และ Setters นั่นเอง ดังนั้นอาจจะกล่าวได้ว่า Property เป็นการใช้คุณลักษณะของ Encapsulation ใน OOP เพื่อป้องกันการเข้าถึงข้อมูลภายในฟิลด์ของออบเจ็กต์โดยตรง ตัวอย่างเช่น Property ชื่อ Color จะเป็นการเข้าถึงข้อมูลสีในคอมโพเนนท์ ซึ่งข้อมูลสีในคอมโพเนนท์จริงๆ จะอยู่ในฟิลด์ชื่อว่า Fcolor แต่ปกติแล้วผู้ใช้จะไม่สามารถเข้าถึงฟิลด์ของออบเจ็กต์ได้โดยตรงต้องกระทำผ่าน Property
- นอกจากนี้ภาษา Delphi ยังสนับสนุนการพัฒนาโปรแกรมแบบ Event-driven programming คือ การเขียนโปรแกรมเพื่อตอบสนองต่อเหตุการณ์ต่างๆ ที่อาจเกิดขึ้นกับคอมโพเนนท์ ซึ่งมีหลักการ เขียนชุดของคำสั่งที่จะกระทำในรูปแบบของ procedure เมื่อเกิดเหตุการณ์ใดๆ ขึ้นกับคอมโพเนนท์ ก็จะมีการไปเรียกใช้ (Trigger) procedure นั้นๆ เพื่อให้ทำงาน ซึ่งชุดคำสั่งนี้สามารถแบ่งออกได้เป็น
  - การเรียกใช้คำสั่ง procedure หรือ function
  - การเข้าถึง property ของ คอมโพเนนท์ใดๆ หรือ การเข้าถึง field ของออบเจ็กต์ใดๆ
  - การเรียกใช้ method ของคอมโพเนนท์หรือออบเจ็กต์ใดๆ

แต่เมื่อเปรียบเทียบกับอีกภาษาหนึ่งเช่นภาษา Perl จะพบว่าการจัดโมดูลในภาษา Perl นั้นจะอยู่ในรูปของ Package ซึ่งคล้ายกับภาษาจาวานั่นเอง ซึ่งเป็น reusable package ที่ได้นิยามไว้ใน library file ที่มีชื่อเดียวกันกับชื่อของ package (ที่มีนามสกุล .pm) อย่างไรก็ตามออบเจ็กต์ ที่อยู่ในภาษา Perl จะมีพื้นฐานการอ้างอิงถึงฟังก์ชันที่อยู่ภายใน Package นั่นคือ จะมีการอ้างอิงถึงสิ่งใดก็ตามที่รู้จักภายในคลาสที่อยู่เท่านั้น

5.2 การจัดการกับหน่วยงานจำโดย Garbage Collection

ลักษณะการอ้างอิงที่อยู่ในหน่วยความจำเรียกว่า พอยต์เตอร์ (Pointer) หรือที่เราเรียกว่าเป็น “ตัวชี้” ในโปรแกรมภาษาแบบ Procedural Language ซึ่งเป็นคุณสมบัติสำคัญที่ใช้ในภาษา C หรือภาษา C++ ในการจัดการใช้ประโยชน์จากการใช้หน่วยความจำแบบไดนามิก ซึ่งทำให้โปรแกรมที่พัฒนามีความยืดหยุ่นสูง แต่ก็เป็นจุดบกพร่องอย่างหนึ่งของภาษาเนื่องจากการใช้งานพอยต์เตอร์อย่างไม่ถูกต้องก็ทำให้เกิดปัญหาหน่วยความจำรั่วไหล หรือเกิดความผิดพลาดในโปรแกรม ที่นักพัฒนาอาจจะไม่ได้ตระหนักถึงผลกระทบที่ตามมา สำหรับภาษาที่สนับสนุนการทำงานแบบออบเจ็กต์ในปัจจุบัน ส่วนใหญ่จะไม่สนับสนุนโครงสร้างข้อมูลแบบพอยต์เตอร์โดยตรง แต่จะเรียกการอ้างอิงไปยังหน่วยความจำนี้ว่า Reference หรืออ้างอิง (ซึ่งนัยหนึ่งแล้วมีความคล้ายกับการอ้างอิงไปยังหน่วยความจำเหมือนกับพอยต์เตอร์นั่นเอง) การใช้งาน Reference ในการทำงานของภาษาจาวา หรือภาษา .NET เช่นภาษา C#, Visual Basic .NET หรือ C++ .NET นั้น การจองใช้หน่วยความจำไม่ได้ทำโดยตรงผ่านทางระบบปฏิบัติการโดยตรงเหมือนกับพอยต์เตอร์ แต่จะทำการจองผ่าน CLR (Common Language Runtime) โดยที่ CLR จะทำหน้าที่ในการตรวจสอบและจองหน่วยความจำก่อนกระบวนการคอมไพล์โปรแกรมให้เป็น Native Code ก่อนแล้วจึงแปลงให้เป็นภาษาเครื่องอีกทีหนึ่ง ซึ่งก่อนการคอมไพล์โปรแกรมโมดูลชื่อว่า Verifier จะทำหน้าที่ในการตรวจสอบเพื่อให้แน่ใจว่าโปรแกรมที่เขียนขึ้นนั้นทำงานได้อย่างถูกต้อง เช่น โปรแกรมไม่ไปเขียนทับหน่วยความจำในส่วนอื่นๆ ของระบบ ซึ่งเราจะเรียกโปรแกรมที่ทำงานบน .NET Framework นี้ว่าเป็น Safe Program เช่นเดียวกันกับโปรแกรมภาษาจาวาซึ่งจะจัดการกับพอยต์เตอร์ผ่านทาง Reference หรือเรียกว่าเป็น Reference Pointer ซึ่งจะถูกจัดการโดย Java Virtual Machine (JVM) โดยที่ผู้ใช้งานไม่สามารถเรียกดูตำแหน่งที่อ้างอิงของพอยต์เตอร์นั้นได้โดยตรง ดังนั้นการใช้งานโปรแกรมภาษาเชิงออบเจ็กต์ ที่มีการใช้งานหน่วยความจำที่ยืดหยุ่น ภาษาเหล่านี้จึงมีตัวจัดการกับหน่วยความจำให้กับนักพัฒนาเรียกว่า Garbage Collection

กลไกควบคุมขยะอินสแตนซ์ (อินสแตนซ์ คือ พื้นที่ในหน่วยความจำซึ่งสร้างตามแบบ โครงร่างของคลาสใดๆ ดังนั้น อินสแตนซ์คือ พื้นที่ที่สามารถเก็บข้อมูลและประมวลผลตามส่วนที่ประมวลผลได้) เกิดขึ้นเนื่องจากขณะที่โปรแกรมทำงาน โปรแกรมจำเป็นต้องมีการสร้างอินสแตนซ์หลายๆอินสแตนซ์เพื่อใช้งาน อินสแตนซ์บางอินสแตนซ์ถูกอ้างด้วยตัวแปรอ้างถึง แต่บางอินสแตนซ์ก็ไม่ถูกอ้างด้วยตัวแปรอ้างถึง ดังนั้นเมื่อโปรแกรมถูกทำงานเป็นระยะเวลาหนึ่ง ดังเช่นในกรณีที่โปรแกรมโค๊ดออกนอกกรอบการทำงาน Out-of-Scope อาจทำให้อินสแตนซ์ที่ไม่ถูกอ้างถึงเกิดเป็นขยะขึ้นและไม่สามารถนำกลับมาใช้งานหรืออ้างถึงได้อีกต่อไป สำหรับภาษาที่สนับสนุนการทำงานเชิงออบเจ็กต์ จะมีกลไกพิเศษที่

สามารถกำจัดขยะอินสแตนซ์ที่ไม่ถูกใช้งานเหล่านี้ออกจากหน่วยความจำ เรียกว่า Garbage Collector ซึ่งจะทำการกำจัดขยะเหล่านี้ ทำงานในฉากหลัง (Background Process) โดยผู้เขียนโปรแกรมไม่จำเป็นต้องเขียนคำสั่งกำจัดขยะด้วยตนเอง และผู้ใช้โปรแกรมก็ไม่รู้เลยว่ามีการกำจัดขยะอินสแตนซ์ขึ้นเมื่อใด เพราะเป็นกลไกที่ทำงานโดยอัตโนมัติ ซึ่งไม่เหมือนกับภาษาเชิงวัตถุตัวอื่นๆ ที่ผู้เขียนโปรแกรมต้องระมัดระวังปัญหาในลักษณะนี้ โดยต้องคืนพื้นที่ให้กับหน่วยความจำทุกครั้งที่ไม่มีการใช้งานอินสแตนซ์ เช่น ในภาษา C++ ต้องใช้คำสั่ง free เพื่อกำจัดอินสแตนซ์ทิ้งออกจากหน่วยความจำทุกครั้ง

สำหรับภาษา Delphi แม้ว่าภาษานี้จะไม่มีกลไกในการทำลายออบเจ็กต์แบบอัตโนมัติอย่างเช่น Garbage Collector ในภาษา Java หรือในภาษา C# แต่ใน Delphi จะใช้แนวคิดที่ว่าออบเจ็กต์ที่เป็น Owner หรือเจ้าของออบเจ็กต์ จะเป็นผู้ทำลายออบเจ็กต์นั้นเอง ตัวอย่างเช่น เมื่อเราทำการสร้างออบเจ็กต์หรือ คอมโพเนนท์โดยใช้คำสั่ง เราจำเป็นต้องระบุชื่อของออบเจ็กต์หรือคอมโพเนนท์ที่เป็นเจ้าของออบเจ็กต์นั้นๆในรูปแบบ

```
object_reference := Class_name.create(Owner);
```

ตัวอย่างเช่น

```
myButton :=  TButton.Create(Form1);
```

โดยเจ้าของออบเจ็กต์ (Owner) หรือคอมโพเนนท์ myButton คือ Form1 ดังนั้นเมื่อ Form1 ถูกทำลายออบเจ็กต์ myButton ก็จะถูกทำลายโดยอัตโนมัติ หรืออีกกรณีหนึ่งคือเราสามารถสั่งทำลายออบเจ็กต์ใดๆได้โดยไม่ต้องรอให้เจ้าของออบเจ็กต์นั้นถูกทำลายโดยใช้เมทธอด free ของออบเจ็กต์นั้น เช่นmyButton.free;

ภาษา Perl ถูกสร้างขึ้นมาโดยมีแนวคิดเพื่อให้สามารถใช้งานง่าย โดยจะมีระบบที่เรียกว่า “Garbage Collection System” อยู่ ซึ่งเมื่อมีการอ้างอิงของออบเจ็กต์ใดๆ ก็ตามและเมื่อสิ้นสุดการทำงานของโปรแกรมแล้วออบเจ็กต์ จะถูกทำลายเข้าไปอยู่ใน garbage collection ในระบบปฏิบัติการ UNIX จะมองว่า garbage collection นั้นจะเป็นขยะ แต่ในความเป็นจริงแล้ว ระบบนี้มีความจำเป็นที่จะฝังหรือ embedded ลงในระบบ หรือในสภาพแวดล้อมที่เป็น Multithreaded

สำหรับภาษา Perl ออบเจ็กต์ที่ถูกสร้างขึ้นมาจะมีการเก็บค่าอ้างอิง (Reference) ไปยังออบเจ็กต์ต่างๆ เพื่อที่จะทำการคืนหน่วยความจำให้กับระบบ ซึ่งออบเจ็กต์เหล่านี้จะถูกทำลายไปโดยอัตโนมัติ หลังจากที่โปรแกรมสิ้นสุดการทำงาน (Program Exit) หรือเมื่อโปรแกรมออกจาก Scope หรือ Block ที่กำหนด แต่ถ้าต้องการที่จะควบคุมการคืนหน่วยความจำ เราสามารถสร้างเมทธอด DESTROY() ขึ้นมาเองภายในคลาส โดยสังเกตว่าชื่อของเมธอดนี้จะในอักษรตัวพิมพ์ใหญ่ทั้งหมด

```
sub DESTROY {
#
# Add code here
#
}
```

โดยเราสามารถกำหนดให้มันทำงานเมื่อใดก็ตามที่มีความจำเป็นที่ต้องมีการทำลายออบเจ็กต์ไป แต่ DESTROY( ) นั้นจะไม่สามารถเรียกใช้ DESTROY( ) ในลักษณะของ nested function ได้

5.3 การนิยามคลาสและออบเจ็กต์ในภาษาเชิงวัตถุ

สำหรับภาษาที่สนับสนุนการทำงานแบบออบเจ็กต์นั้น ออบเจ็กต์จะถูกสร้างขึ้นมาจากการนิยามของคลาส (Class Definition) ซึ่งออบเจ็กต์จะถูกสร้างขึ้นมาจากคลาสในลักษณะที่เป็น Static หรือ Dynamic ซึ่งเกิดขึ้นขณะที่โปรแกรมมีการทำงาน โดยคลาสที่กำหนดขึ้นมานั้นอาจจะเป็นคลาสที่สร้างขึ้นใหม่ หรือเป็นคลาสที่เกิดขึ้นจากการสืบทอดจากอีกคลาสหนึ่งก็ได้ ภาษาจาวาเป็นภาษาที่มีคุณสมบัติทางออบเจ็กต์ที่เข้มงวด โดยมีข้อกำหนดอยู่ว่าในหนึ่งโปรแกรมต้องมีคลาสอย่างน้อยหนึ่งคลาส การสร้างตัวแปรหรือเมทธอดของจาวาจำเป็นต้องระบุให้อยู่ในคลาส โดยที่การประกาศคลาสในภาษาจาวามีข้อกำหนดดังนี้

```
[modifier] class ClassName {
    [class member]
}
```

ความหมายของส่วนต่าง ๆ ของคลาสมีดังนี้

[modifier] เป็นตัวที่อธิบายถึงลักษณะของคลาส เช่น public static abstract final เป็นต้น

class เป็นคำที่กำหนดว่าสิ่งที่สร้างคือ คลาส

ClassName เป็นชื่อของคลาสส่วนใหญ่จะเป็นคำที่สื่อให้รู้ถึงเนื้อหาที่อยู่ภายในคลาส

[class member] เป็นสมาชิกของคลาสประกอบด้วย Data และ Method

ตัวอย่างเช่น

```
public ClassRoom {
    int roomId;
    String commonName;

        public String getCommonName(){
        return this.commonName;
    }
}
```

สำหรับภาษา C# ในลักษณะเดียวกันคลาสซึ่งเป็นแนวคิดหนึ่งของการทำชนิดข้อมูลเชิงนามธรรม (Abstract Data Type: ADT) และแนวทางการพัฒนาเชิงวัตถุ โดยมีการจัดทำเป็นต้นแบบของออบเจ็กต์ในลักษณะของคลาส ซึ่งจะสังเกตเห็นได้ว่าโครงสร้างของภาษา C# นั้นมีความคล้ายคลึงกัน

กับโครงสร้างของภาษา Java มาก สืบเนื่องมาจาก 2 ภาษานี้มีภาษาต้นแบบมาจากภาษา C หรือ C++ นั่นเอง

```
using System;
public class Producer
{
    private int NoOfDish;

    Producer(int n) {
        this.NoOfDish = n;
    }
}
```

ซึ่งหลังจากที่มีการประกาศคลาสแล้วเราสามารถนำเอาคลาสนั้นมาสร้างเป็นออบเจ็กต์เพื่อนำไปใช้งานในโปรแกรมได้

```
public class Consumer
{
    public static void Main(){
        Producer p;
     p = new Producer(15);
    }
}
```

สำหรับการสร้างออบเจ็กต์ในภาษา Delphi จำเป็นต้องทำการสร้างหรือนิยามคลาสขึ้นมาก่อนแล้วจึงจะสามารถทำการสร้างออบเจ็กต์ขึ้นมาจากนิยามนั้นได้ ซึ่งคลาสในที่นี้คือ ข้อกำหนดที่ระบุรูปแบบโครงสร้างและพฤติกรรมหรือการทำงานของออบเจ็กต์ หรืออาจกล่าวได้ว่าคลาสคือต้นแบบของออบเจ็กต์ การสร้างคลาสในภาษา Delphi จะอาศัยชนิดของข้อมูล (Type) ซึ่งจะคล้ายกับข้อมูลชนิดที่เป็น Record ที่ประกอบไปด้วยฟิลด์ข้อมูล แต่คลาสในภาษา Delphi จะต่างจากเรคคอร์ดตรงที่ข้อมูลของคลาส นอกจากจะมีฟิลด์ข้อมูลแล้วยังสามารถมีพฤติกรรม (Method) ซึ่งเป็นคำสั่งที่เขียนขึ้นในรูปแบบของ Procedure หรือ Function ดังนั้นคลาสในภาษา Delphi ก็คือชนิดของข้อมูล (Type) ชนิดหนึ่ง ซึ่งลักษณะการตั้งชื่อคลาสจะใช้ตัวอักษร T นำหน้า เช่น TComponent, TButton, TLabel, TColor, TFont สำหรับการนิยามคลาสใน Delphi จะใช้รูปแบบ ดังนี้

```
type
      Class_Name = class(Ancestor_Class)
      Member List
end;
```

- Class_Name คือ ชื่อคลาสที่ต้องการสร้าง
- Ancestor_Class คือ ชื่อคลาสบรรพบุรุษหรือคลาสต้นแบบ (Based Class) ที่จะสืบทอดสิ่งต่างๆมาใช้ในคลาสที่กำลังสร้าง แต่ถ้าหากไม่ระบุ Ancestor คลาส จะถือว่าคลาสนั้นสืบทอดมาจากคลาส TObject ใน Delphi คลาส TObject จะเป็นคลาสบรรพบุรุษของทุกคลาส
- Member List คือ สมาชิกของคลาส ได้แก่

- Field คือ ข้อมูลหรือตัวแปรภายในของออบเจ็กต์
- Method คือ พฤติกรรมของออบเจ็กต์เป็นชุดคำสั่งที่เขียนในรูปแบบ Procedure หรือ Function Method

- Properties คือ interface เป็นวิธีการเข้าถึง Filed ข้อมูลซึ่งเป็นตัวแปรภายในของออบเจ็กต์ การเข้าถึงจะใช้ access specifies read หรือ write เพื่อระบุว่าจะเข้าไปอ่าน หรือ เขียน (แก้ไข) ข้อมูลภายในของออบเจ็กต์ ซึ่ง Properties มีเฉพาะออบเจ็กต์ที่เป็นคอมโพเนนท์เพื่อใช้เข้าถึงข้อมูลภายในคอมโพเนนท์ผ่าน Tab Properties ของออบเจ็กต์ Inspector ในขณะที่ทำการออกแบบโปรแกรม

แต่สำหรับภาษา Perl แล้วคลาสจะอยู่ในระดับเดียวกันกับ Package ซึ่งเมื่อใดที่มีการอ้างอิงถึงคลาสแล้วภาษา Perl จะมองภาพการอ้างอิงเป็น Package ในการออกแบบโครงสร้างคลาสในภาษา Perl นั้นจะคล้ายกับภาษา C Programming ซึ่งทำให้ผู้ที่มีพื้นฐานทางด้านภาษา C มาก่อนสามารถเรียนรู้ภาษา Perl ได้ง่ายขึ้น เนื่องจากมี Syntax ของภาษาที่คล้ายกัน ตัวอย่างเช่น การประกาศคลาสและการสืบสกุลของคลาสจะใช้เครื่องหมาย Double colon “::” ดังตัวอย่างต่อไปนี้

```
Package Cocoa::BigCocoa;
#
#  Just add code here
#
1;
```

จากตัวอย่างเป็นการสร้างคลาส Cocoa โดยอาศัยหลักการเดียวกันกับการสร้าง Package โดยที่ Package ไฟล์ที่สร้างขึ้นมานั้นจะมีชื่อว่า Cocoa.pm โดยที่นามสกุล .pm ที่ใช้นั้นเป็นนามสกุลที่เป็นค่า Default สำหรับ Package ใน Perl Module

5.3.1 การสร้างออบเจ็กต์

การสร้างออบเจ็กต์ขึ้นมาเพื่อใช้งานภายในโปรแกรม แต่ละภาษามีแนวคิดที่คล้ายกันคือออบเจ็กต์จะถูกสร้างขึ้นมาจากคลาส ซึ่งเราเรียกว่า “Instantiation” ในขั้นตอนนี้ระบบจะทำการจัดสรรหน่วยความจำเพื่อรองรับออบเจ็กต์ที่สร้างขึ้นใหม่นี้ หลังจากที่ออบเจ็กต์ถูกสร้างขึ้นมาแล้วเราสามารถเข้าถึงข้อมูลและเมทธอดภายในคลาสได้จาก instance ของคลาสนั่นเอง สำหรับการสร้างออบเจ็กต์ในภาษาจาวานั้น สามารถทำได้โดยอาศัยคีย์เวิร์ด new ดังตัวอย่างนี้

```
MyCircle circle = new MyCircle();
```

หรือกรณีที่ต้องการสร้างออบเจ็กต์ขึ้นมาใช้งานมากกว่า 1 ออบเจ็กต์ สามารถทำได้ดังนี้

```
MyCircle circle1 = new MyCircle();
MyCircle circle2 = new MyCircle();
```

```
MyCircle circle3 = new MyCircle();
```

โดยที่ MyCircle เป็นคลาสที่ได้มีการนิยามไว้ และ circle1, circle2, circle2 เป็นออบเจ็กต์ หรือเรียกได้ว่าเป็นอินสแตนซ์ของออบเจ็กต์ โดยเมื่อสร้างออบเจ็กต์แล้วถ้าต้องการเรียกใช้งานข้อมูลหรือเมทธอดของออบเจ็กต์ MyCircle ก็สามารถเรียกใช้งานผ่านอินสแตนซ์ที่สร้างขึ้นได้ โดยมีรูปแบบการใช้งาน

ObjectName.Attribute หรือ ObjectName.Method()

เช่น

circle1.Area และ circle1.CalculateArea()

ซึ่งถึงแม้ว่าแต่ละอินสแตนซ์ที่สร้างขึ้นทั้ง 3 ตัวนั้นจะมาจากคลาสเดียวกัน แต่ในแต่ละออบเจ็กต์นั้นก็มีความเป็นอิสระไม่ขึ้นอยู่ต่อกัน สำหรับการสร้างออบเจ็กต์ในภาษา Delphi จะมีลักษณะการทำงานคล้ายกับภาษาอื่นๆ โดยจะสร้างออบเจ็กต์ขึ้นมาจากคลาส ซึ่งการสร้างออบเจ็กต์ขึ้นจากคลาสอาจเรียกว่า การสร้าง Instance ของคลาสซึ่งจะเป็นขั้นตอนที่เรียกว่า การ Instantiate โดยมีรูปแบบการสร้างออบเจ็กต์ดังนี้

```
object_referrence := Class_name.create(Owner);
```

โดยที่ object_reference คือ Object reference หรือ ตัวแปร pointer ที่ชี้หรืออ้างอิงไปยัง object
Class_name เป็นชื่อของคลาสที่จะใช้สำหรับสร้างเป็นออบเจ็กต์
Create เป็นเมทธอดของคลาส (Class method) ที่ทำหน้าที่ในการสร้างออบเจ็กต์ในหน่วยความจำ ซึ่งจะเรียกว่าเมทธอดประเภทนี้ว่า Constructor method
Owner เป็นชื่อออบเจ็กต์ ซึ่งเป็นเจ้าของออบเจ็กต์นั้น หากไม่มีการระบุถือว่าเป็น Nil

เช่น การสร้างคอมโพเนนท์ชนิด TButton โดยใช้คำสั่ง

```
var
 myButton  :  TButton;        //  ประกาศออบเจ็กต์ reference
 begin
     myButton := TButton.Create(Form1);
 end;
```

ซึ่งหลังจากนี้การอ้างถึงออบเจ็กต์ โดยใช้ชื่อตัวแปรออบเจ็กต์ reference ซึ่งการเข้าถึงสมาชิกของออบเจ็กต์ ใดๆ จะใช้รูปแบบ object_reference.<Property> หรือ object_reference.<Method>

5.3.2 คอนสตรัคเตอร์ (Constructor)

คอนสตรัคเตอร์จัดเป็นเมทธอดประเภทหนึ่งที่จำเป็นต้องมีในทุกคลาส สำหรับภาษา Java ภาษา C++ และภาษา C# เองมีรูปแบบการใช้งานคอนสตรัคเตอร์ที่เหมือนกัน คือมองว่าคอนสตรัคเตอร์นั้นเป็นเมทธอดหนึ่งๆ ในคลาสที่มีชื่อเดียวกันชื่อของคลาส สามารถทำการโอเวอร์โหลดได้ โดยคอนสตรัคเตอร์มีคุณสมบัติเหมือนกับเมทธอดทั่วไป ยกเว้นที่เมทธอดที่ทำหน้าที่เป็นคอนสตรัคเตอร์จะไม่สามารถส่งค่ากลับได้ ดังตัวอย่าง

```
class Bicycle
{
    int color;
    Bicycle()    //constructor
    {
   …
    }
}
```

แต่ถ้าคลาสใดไม่มีคอนสตรัคเตอร์เมทธอด ในภาษาจาวาคอมไพเลอร์จะสร้างให้หนึ่งเมทธอด โดยปริยายในลักษะของ Default Constructor (เป็นคอนสตรัคเตอร์ที่ไม่มีการผ่านค่าตัวแปร) ดังนั้นคลาสลูกก็จะได้รับการสืบทอดคลาสดังกล่าวมาด้วยถ้าหากคลาสลูกไม่มีการเขียนคอนสตรัคเตอร์ใหม่ โดยจะมีรูปแบบดังนี้

```
class Player  {
 private String name;
    private String address;

    Player() {
        Name  = "John";
        addres = "Norfolk";
    }

    public void showInfo()  {
        System.out.println("In Player");
    }
}


class GoalKeeper extends Player  {
    private String skills;

    GoalKeeper(String s) {
        Skills = s; //No default constructor in Player class
    }
   public void showInfo()  {
        System.out.println("In GoalKeeper");
    }
 }
```

```
class InheritConstructor  {
   public static void main(String[] args)  {
          Player p = new Player();
          GoalKeeper g = new GoalKeeper();
          g.showInfo();
   }
}
```

จากตัวอย่าง คอนสตรัคเตอร์ของ GoalKeeper จะสืบทอดคอนสตรัคเตอร์ของ Player โดยปริยาย อย่างไรก็ตามถ้าซูเปอร์คลาสมีการเขียนทับคอนสตรัคเตอร์ไปแล้ว คอนสตรัคเตอร์ของคลาสลูกต้องอ้างอิงถึงคอนสตรัคเตอร์ของซูเปอร์คลาสให้ถูกต้อง เช่น ถ้ามีการสร้างคอนสตรัคเตอร์ของคลาส Player เป็น Player(String a, String b) คลาส GoalKeeper จะไม่สามารถอ้างอิงถึงคอนสตรัคเตอร์ของคลาส Player ได้ เพราะไม่มีคอนสตรัคเตอร์ที่เป็น Default Constructor การสร้างคอนสตรัคเตอร์โดยอ้างอิงถึงซูเปอร์คลาสคอนสตรัคเตอร์ ต้องใส่คำสั่ง super() เป็นคำสั่งแรกเพื่อเรียกคอนสตรัคเตอร์ที่อยู่ในคลาสแม่ และกำหนดค่าตัวแปรต่างๆให้ตรงกับคอนสตรัคเตอร์ของคลาสแม่ เช่น ถ้าคอนสตรัคเตอร์ของคลาส GoalKeeper เป็น GoalKeeper(String a, String b, String c) คำสั่งที่ต้องใส่คอนสตรัคเตอร์ของคลาส GoalKeeper คือ super(a, b) เพื่อเป็นการเรียกคอนสตรัคเตอร์ที่อยู่ใน Player เป็นต้น

สำหรับ Delphi เมทธอดคอนสตรัคเตอร์ สามารถมีชื่ออะไรก็ได้เนื่องจากการระบุคอนสตรัคเตอร์จะใช้คีย์เวิร์ด “constructor” ในการระบุ ไม่เหมือนบางภาษาที่ใช้ชื่อคลาสในการระบุคอนสตรัคเตอร์ แต่ใน Delphi ก็มักนิยมตั้งชื่อคอนสตรัคเตอร์ว่า “Create” และสิ่งที่ส่งกลับมาจากเมทธอดคอนสตรัคเตอร์คือ reference ของออบเจ็กต์ ตัวอย่างเช่น

```
var  mycircle :  TCircle;
begin
    mycircle :=  TCircle.create;          //  ไม่มีการระบุ owner
end;
```

จากตัวอย่างนี้ เมื่อเรียกใช้คอนสตรัคเตอร์ create mycircle จะชี้ไปที่ ออบเจ็กต์ชนิด TCircle ในหน่วยความจำและฟิลด์ข้อมูลของออบเจ็กต์ mycircle จะถูกกำหนดให้เป็น 0 หรือ Nil (ถ้าเป็น reference) ตัวอย่างของการสร้างคอนสตรัคเตอร์สำหรับคลาส Date

```
type
    TDate = class
    public
          constructor Create (y,m,d:Integer);
    end;

constructor TDate.Create (y,m,d:Integer);
begin
    fDate := EncodeDate (y,m,d);
end;
```

และตัวอย่างการเรียกใช้งาน

```
var
             ADay:TDate;
begin
             ADay := TDate.Create (1,10,2011);
end;
```

จากตัวอย่างข้างต้นการเรียกคอนสตรัคเตอร์จำเป็นต้องมีการส่งค่าพารามิเตอร์เนื่องจากมีเพียงคอนสตรัคเตอร์เดียว และจำเป็นต้องมีการส่งค่า หากต้องการให้มีการกำหนดค่าเริ่มต้น ให้ทำการสร้างคอนสตรัคเตอร์ในลักษณะที่เป็น Overload Constructor ดังตัวอย่างต่อไปนี้

```
type
         Tdate = class
         public
         constructor Create; overload;
         constructor Create (y,m,d:Integer); overload ;
    end;

    constructor TDate.Create (y,m,d:Integer);
    begin
        fDate :=EncodeDate (y,m,d);
    end;

    constructor TDate.Create;
    begin
        fDate :=Date;
    end;
```

สำหรับการคอนสตรัคเตอร์ในภาษา Perl จัดว่าเป็นฟังก์ชันหนึ่งเรียกว่า subroutine ซึ่งจะคล้ายกับฟังก์ชัน หรือซับรูทีนทั่วไป และสามารถส่งค่ากลับในลักษณะของค่าอ้างอิง (Reference) การเรียกใช้ คอนสตรัคเตอร์นั้นจะต้องเรียกผ่าน Subroutine ที่มีชื่อเฉพาะว่า “bless” บางครั้งเราจะเรียก Subroutine นี้ว่า “blessing” ออบเจ็กต์ ตามตัวอย่างโค๊ดต่อไปนี้ที่จะแสดงจะเป็นส่วนที่แสดงให้เห็นถึง syntax ของ bless ฟังก์ชัน

bless YeReference [, classname]

- YeReference จะอ้างอิงไปยัง object ที่กำลัง bless
- Classname ใช้สำหรับระบุชื่อของคลาส จะเป็น optional คือจะมีหรือไม่มีก็ได้ และจะมีการใส่ชื่อของ Package ที่อ้างอิงถึงด้วย ถ้าไม่ใส่ classname ก็จะใช้ชื่อของ Package ที่อยู่ปัจจุบันแทน

```
package Cocoa;

sub new
{
    my $this = { };        #Create an anonymous hash, and #self points to it.
    Bless $this;   #Connect the hash to the package Cocoa.
    return $this;  #Return the reference to the hash.
```

```
}

#!/usr/bin/perl    #Caller Program - Demonstrate how to call package Cocoa

use Cocoa;
$cup = new Cocoa;
```

จากตัวอย่างของการเรียกคอนสตรัคเตอร์จะเห็นว่ามีการสร้าง Package Cocoa และ Subroutine ชื่อว่า new เพื่อทำการส่งค่าอ้างอิงกลับ การเรียกใช้งาน Package นั้นจะต้องใช้คำสั่ง use package โดยการที่จะคอนสตรัคเตอร์จะถูกเรียกใช้เมื่อมีการสร้างออบเจ็กต์ขึ้นมา โดยคำสั่ง

```
$cup = Cocoa -> new();
```

หรือ

```
$cup = Cocoa :: new();
```

ซึ่งมีรูปแบบการใช้งานเหมือนกับภาษา C

5.3.3 การเข้าถึงสมาชิกของออบเจ็กต์ (Object Accessibility)

แนวคิดของการควบคุมการเข้าถึง หรือบางครั้งเราเรียกว่าเป็นแนวคิดของการควบคุมการมองเห็นของคุณสมบัติที่ซ่อนอยู่ภายในออบเจ็กต์ เป็นแนวคิดที่พัฒนาต่อมาจากชนิดข้อมูลที่เป็นนามธรรม (Data Abstraction) ซึ่งเป็นการสร้างมุมมองใหม่ของออบเจ็กต์ที่อาจจะมีรายละเอียด และขั้นตอนการทำงานที่ซับซ้อน ตัวอย่างเช่นออบเจ็กต์เครื่องเล่นวิดีโอ VCR ที่มีกลไกการทำงานที่ซับซ้อน แต่ในมุมมองของผู้ใช้งานเครื่องเล่นวิดีโอนั้น คือเพียงแค่ใส่แผ่นซีดีหรือดีวีดี และกดปุ่ม Play ก็จะทำให้สามารถเครื่องเล่นทำงานได้ โดยที่ผู้ใช้เพียงแค่รู้ว่า VCR นั้นสามารถทำอะไรได้ แต่ไม่จำเป็นต้องรู้ถึงว่าสามารถทำได้อย่างไร

การกำหนดการเข้าถึงสมาชิกหรือเมทธอดภายในออบเจ็กต์เรียกว่า Encapsulation หรือ Data Hiding เป็นรากฐานอย่างหนึ่งของแนวคิดของการพัฒนาซอฟต์แวร์เชิงวัตถุ ซึ่งข้อดีของเอนแคปซูเลชั่นเป็นการปกป้องคุณสมบัติของออบเจ็กต์จากความเสียหาย (ในที่นี้จะหมายถึงการเปลี่ยนแปลงของค่าที่เก็บในออบเจ็กต์) โดยเอนแคปซูเลชั่นจะทำการการห่อหุ้ม (Wrapper) คุณสมบัติ (Data Member) และเมทธอด (methods) เข้าไว้ด้วยกัน จะทำหน้าที่ป้องกันมิให้ออบเจ็กต์อื่นที่อยู่ภายนอกเข้าถึงออบเจ็กต์หนึ่งๆได้อย่างอิสระ ซึ่งออบเจ็กต์จะต้องกำหนดว่ามีคุณสมบัติใดบ้างที่สามารถให้เข้าถึงได้ หรือจะมีเฉพาะเมทธอดที่กำหนดให้เท่านั้นที่จะสามารถติดต่อกับคุณสมบัติภายในออบเจ็กต์ได้ การจำกัดการมองเห็นของคุณสมบัติของออบเจ็กต์นั้นเรียกว่า Visibility หรือในทำนองเดียวกันกับการกำหนด Scope of Variable ในการเขียนโปรแกรมแบบโครงสร้าง ซึ่งมีการกำหนดขอบเขตการมองเห็น Scope of Visibility โดยคีย์เวิร์ด public, private และ protected

1. public หมายถึง ความสามารถในการเข้าถึงได้ทั้งจากภายในออบเจ็กต์ตัวเอง และต่างออบเจ็กต์
2. private หมายถึง สามารถเข้าถึงได้จากเฉพาะภายในออบเจ็กต์เดียวกันเท่านั้น

3. protected หมายถึง ความสามารถที่จะเข้าถึงคุณสมบัติภายในออบเจ็กต์เดียวกัน และออบเจ็กต์ที่เป็นที่สืบทอดเท่านั้น

ประโยชน์ของแนวคิดของเอนแคปซูเลชั่นคือการควบคุมการเข้าถึงคุณสมบัติและเมทธอดของออบเจ็กต์ ทั้งในแง่ของการเรียกใช้ข้อมูล หรือการเปลี่ยนแปลงข้อมูลเกิดขึ้นภายในออบเจ็กต์ ซึ่งจะส่งผลต่อสถานะของออบเจ็กต์ที่เปลี่ยนไป หรือจะไม่ส่งผลกระทบต่อออบเจ็กต์อื่น

ในภาษาจาวา การกำหนดการมองเห็นของคลาสที่มีการสืบทอดสามารถทำได้โดยอาศัยคลาสเมทธอดสามารถเรียกใช้งานผ่านอินสแตนซ์ของออบเจ็กต์ที่สร้างขึ้น ทั้งนี้การอ้างอิงถึง Data Member หรือ Method ภายในคลาสโดยใช้คีย์เวิร์ด super และ this ซึ่งเป็นการอ้างอิงกับเมทธอดหรือตัวแปรในคลาส คลาสที่สืบทอดหรือคลาสที่เป็นซูเปอร์คลาส ในลำดับที่ต่อเนื่องกันเท่านั้น ถ้าลำดับชั้นของคลาสมีมากกว่าสองลำดับชั้น และจำเป็นต้องอ้างอิงถึง ซูเปอร์คลาสในลำดับที่สูงกว่าลำดับของซูเปอร์คลาสที่กำลังอ้างอิงถึง ต้องเขียนโปรแกรมเป็นลูกโซ่เพื่ออ้างอิงถึงเท่านั้น แต่ออบเจ็กต์ที่สร้างโดยการใช้คำสั่ง new สามารถอ้างอิงถึงตัวแปรในแต่ละคลาสได้โดยการแปลงชนิดของออบเจ็กต์ (Object type casting) ไปยังออบเจ็กต์ที่ต้องการ พิจารณาจากตัวอย่างต่อไปนี้

```
class People {
   protect String name;

   People (String n) {
       this.name = n;
   }

   public void showInfo() {
       System.out.println(name);
   }
}

class Student extends People {
   private Decimal GPA;

   Student (String n, Decimal GPA) {
       super(n);
       this.GPA = GPA; //using 'this' keyword to refer to class member
   }

   public void showInfo() {
       super.showInfo();      //calling showInfo in Parent class
       System.out.println(GPA);
   }
}


class Program {
   public static void main(String [] argv) {
          People p = new People("John");
          Student s = new Student("Sam", 3.50);
          p.showInfo();
          p = (Student) s;
          p.showInfo();
       }
```

```
}
```

จากตัวอย่างดังกล่าว จะเห็นได้ว่ามีการสร้างคลาส Student เพื่อทำการสืบทอดจากคลาส People โดยที่มีการส่งค่าพารามิเตอร์ระหว่างคอนสตรัคเตอร์เพื่อให้มีการกำหนดค่าเริ่มต้นให้แก่ตัวแปร ซึ่งตัวแปร Name นั้นเป็นตัวแปรที่ประกาศในคลาส People แต่ก็ยังสามารถถ่ายทอดคุณสมบัตินี้ลงมาสู่คลาส Student ด้วย สังเกตุว่าการอ้างอิงถึงคุณสมบัติที่อยู่ในคลาสแม่จะอ้างอิงโดยใช้คำสั่ง super ในคอนสตรัคเตอร์สำหรับคลาสลูก

สำหรับภาษา Delphi คลาสเมทธอดเป็นเมทธอดที่ผูกติดอยู่กับคลาสนั้นๆ ซึ่งจะสามารถเรียกผ่านคลาสเมื่อไรก็ได้ สำหรับการประกาศคลาสเมทธอดจะต้องใช้คีย์เวิร์ด "class" นำหน้าชื่อของคลาส เช่น

```
type
Class_Name = class(Ancestor_Class)
Private
 ... { private declarations here}
protected
 ... { protected declarations here }
public
... { public declarations here }
published
... { published declarations here }
end;
```

สำหรับนิยามของแต่ละคีย์เวิร์ดที่ใช้สำหรับการควบคุมการมองเห็นนั้น อธิบายได้ดังต่อไปนี้

- Private เป็นการระบุว่าฟิลด์ข้อมูลหรือเมทธอดจะสามารถมองเห็นได้เฉพาะ ภายใน unit หรือ program ที่นิยามคลาส นี้เท่านั้น
- Protected เป็นการกำหนดให้สามารถถูกมองเห็นได้เฉพาะ unit หรือ source program ที่นิยามคลาสนี้และจาก unit หรือ source program อื่นๆ ที่นิยามคลาสที่สืบสกุลจากคลาสนี้เท่านั้น
- Public เป็นการกำหนดให้สามารถถูกมองเห็นได้จากทุก unit และทุก source program
- Published เป็นการกำหนดให้มีขอบเขตการมองเห็นเหมือน public members แต่สิ่งที่แตกต่างคือ จะมีการสร้างตาราง RTTI (RunTime Type Information) ในหน่วยความจำเพื่อให้สามารถเข้าถึง Properties ในขณะออกแบบโดยผ่านออบเจ็กต์ Inspector ดังนั้นสิ่งที่กำหนดในส่วน published คือ property

5.3.4 การสืบทอด (Inheritance)

หัวใจสำคัญของการพัฒนาโปรแกรมเชิงวัตถุคือการใช้ประโยชน์โปรแกรมโค๊ดที่มีอยู่ Code Reusability หรือการนำสิ่งที่มีอยู่มาใช้โดยไม่ต้องสร้างใหม่ ซึ่งคุณสมบัติที่สนับสนุนแนวคิดนี้คือการ

สืบทอดซึ่งเราสามารถสร้างคลาสใหม่ โดยสืบทอดโครงสร้างและพฤติกรรมทั้งหมดจากคลาสต้นแบบหรือซูเปอร์คลาส ซึ่งคลาสใหม่ที่สืบสกุลมานี้อาจจะสืบทอดโครงสร้างหรือพฤติกรรม ของคลาสต้นแบบโดยมีการเพิ่มคุณสมบัติหรือเมทธอด หรือ อาจจะเป็นการแก้ไขพฤติกรรมของคลาสต้นแบบ โดยการแก้ไขเมทธอดที่สืบทอดมาจากคลาส ต้นแบบ การสืบสกุลในภาษา Delphi ไม่จำเป็นต้องใช้คำสั่งพิเศษใดๆ เนื่องจากรูปแบบของการสร้างคลาสมีการระบุชื่อคลาสต้นแบบอยู่แล้ว เช่น

```
type
       Class_Name  = class(Ancestor_Class)
       Member_List
end;

type
   TDog  =  class(TAnimal)
            …..
End;
```

ในตัวอย่างนี้คลาส TDog สืบทอดคุณสมบัติจาก TAnimal แต่ถ้าไม่ระบุคลาสต้นแบบ คลาสนี้จะสืบสกุลมากจากคลาส TObject ซึ่งใน Delphi ถือว่าคลาส TObject เป็นต้นแบบให้กับทุกคลาส การสืบสกุลใน Delphi จะมีลักษณะเป็น Single Inheritance คือ สามารถสืบสกุลจากคลาสต้นแบบได้เพียงคลาสเดียวเท่านั้นแต่ Delphi สามารถอิมพลีเมนท์ได้หลายอินเตอร์เฟสในลักษณะเดียวกันกับภาษาจาวา

สำหรับภาษา Object Perl เองซึ่งเป็นภาษาที่สนับสนุนแนวคิดของการสืบสกุลด้วย paths ที่อยู่ใน `@ISA array` และตัวแปรต่างๆ จะต้องถูกสืบทอดมาด้วย

5.3.5 โอเวอร์โหลดและโอเวอร์ไรด์ (Overloading and Overriding)

การสร้าง Overloading method ใน Delphi จะใช้คีย์เวิร์ด overload ตามหลังชื่อเมทธอด ซึ่งคอมไพเลอร์จะเป็นตัวตัดสินใจว่าจะเรียกใช้เมทธอดไหน โดยพิจารณาจาก Signature ของเมทธอด ดังตัวอย่างนี้ ที่ทำการสร้าง overload method ชื่อ SetValue

```
type
    TDate = class
public

procedure SetValue (y,m,d:Integer);overload;
procedure SetValue (NewDate:TDateTime);overload;
...//the rest of the class declaration

procedure TDate.SetValue (y,m,d:Integer);
begin
    fDate := EncodeDate (y,m,d);
end ;

procedure TDate.SetValue(NewDate:TDateTime);
begin
    fDate := NewDate;
```

```
end;
```

การทำโอเวอร์ไรด์เป็นการการเปลี่ยนแปลงแก้ไขเมทธอดที่มีการสืบทอดมาจากคลาสต้นแบบในคลาสใหม่ ซึ่งเมทธอดที่จะทำการโอเวอร์ไรด์ได้นั้นต้องเป็น Virtual method หรือ Dynamic method เท่านั้น ดังนั้นในขั้นตอนนิยามเมทธอดในคลาส ถ้าต้องการให้เมทธอดนั้นสามารถถูกโอเวอร์ไรด์ได้ ในคลาสที่จะสืบสกุลต่อไป ต้องระบุ directive Virtual หรือ Dynamic และในคลาสที่สืบสกุลถ้าต้องการ override virtual หรือ dynamic method ต้องระบุ override หลังการนิยาม เมทธอดที่ต้องการจะ override ด้วย ดังตัวอย่าง

```
Type
       TMyClass = class            //Derived from TObject
           Procedure   One; Virtual;   //Allow override
       Procedure   Two;               //Static method ทำ override ไม่ได้
End;

TMySubClass = Class(TMyClass)
Procedure  One; Override;         //Will be overrided
Procedure  Two;     //Overriding "Two" from based class
```

5.3.6 โพลีมอร์ฟิซึม (Polymorphism)

โพลีมอร์ฟิซึมหมายถึง การที่เมทธอดมีได้หลายรูปหรือหลายพฤติกรรมตามสถานะของออบเจ็กต์ ซึ่งเป็นคุณสมบัติที่สำคัญอย่างหนึ่งในการพัฒนาโปรแกรมเชิงวัตถุ ซึ่งโพลีมอร์ฟิซึมจะเป็นการใช้กลไกของการสืบทอดและการทำ late binding คือ การเรียกใช้งานเมทธอดในขณะที่โปรแกรมทำงาน โพลีมอร์ฟิซึมมีหลักแนวคิดที่เป็นความสามารถของ 2 ออบเจ็กต์ ขึ้นไปที่ตอบสนองต่อ message เดียวกัน ในวิธีที่ต่างกัน ภาษา Java มีกลไกสืบทอดและการทำ dynamic binding จึงทำให้สามารถสร้างเมทธอดที่มีลักษณะพิเศษอย่างหนึ่งคือสามารถรับพารามิเตอร์เป็นอินสแตนซ์ของคลาสที่ต่างกันได้ และแม้ว่า เมทธอดจะจัดการกับพารามิเตอร์ที่รับมาด้วยโปรแกรมเดียวกันแต่ผลที่ได้จะขึ้นกับว่า พารามิเตอร์นั้นเป็นอินสแตนซ์ของคลาสใด ทำให้ไม่จำเป็นต้องแบ่งแยกประเภทเพื่อจัดการ เรียกแนวความคิดในการจัดการกับอินสแตนซ์ของคลาสที่ต่างกันด้วยเมทธอดเดียวกันว่าโพลีมอร์ฟิซึม แนวความคิดนี้ไม่ได้จำกัดอยู่กับเพียงรูปแบบของเมทธอดเท่านั้น แต่อาจจะขยายเป็นอาร์เรย์ที่เก็บอินสแตนซ์ของคลาสที่ต่างกัน เพื่อให้สามารถใช้ประโยค for จัดการกับสมาชิกในอาร์เรย์นั้นด้วยโปรแกรมเดียวกัน ซึ่งจะเห็นในตัวอย่างต่อไปนี้

```
        // PolyMethod.java
        class A  {  void print ()  {  System.out.println ('A');  }  }
        class B  extends A { void print ()  { System.out.println ('B'); }
}
        class PolyMethod  {
           static void  test (A x)  { x.print (); }
           public static void main (String args [] )  {
                  A a = new A ();
                  test (a);
                  a = new B ();
```

```
                test (a) ;
            }
        }
```

จากตัวอย่าง คลาส A มีเมทธอด print() ที่พิมพ์อักษร ‘A’ ออกมา ส่วนคลาส B ที่ขยายมาจากคลาส A และมีเมทธอด print() ที่พิมพ์อักษร ‘B’ ออกมา ในคลาส PolyMethod มีเมทธอด test () ที่รับอินสแตนซ์ของคลาส A เป็นพารามิเตอร์ ทำให้สามารถส่งอินสแตนซ์ของทั้งคลาส A และ B เป็นพารามิเตอร์ ที่เมทธอด นี้ได้ แสดงว่า test() มีคุณสมบัติของโพลีมอร์ฟิซึม และในเมทธอด main() มีการสร้างอินสแตนซ์ a ของคลาส A และส่งไปที่ test() ได้ผลการพิมพ์ของ x.print() ออกมาเป็น ‘A’ ต่อมาจึงสร้างอินสแตนซ์ของคลาส B และกำหนดค่าให้แก่ reference a แล้วส่งไปที่ test() ได้ผลการพิมพ์ของ x.print() ออกมาเป็น ‘B’ สังเกตว่า test() จัดการกับวัตถุ x ด้วยโปรแกรมเดียวกันคือ x.print() แต่ print() ถูกผูกชื่อแบบไดนามิก ดังนั้นเมื่อส่งอินสแตนซ์ของคลาส A ไปจะได้ผลของการพิมพ์เป็น ‘A’ และเมื่อส่งอินสแตนซ์ของคลาส B ไปก็ได้ผลของการพิมพ์เป็น ‘B’ แสดงว่า test() ให้ผลการทำงานขึ้นกับว่าอินสแตนซ์ที่ส่งเข้ามาว่าเป็นของคลาสใด

ลักษณะการทำงานของโพลีมอร์ฟิซึมในภาษา Delphi คือเมทธอดที่สามารถจัดการกับออบเจ็กต์หรืออินสแตนซ์ของคลาสที่ต่างกันได้ หากคลาสเหล่านั้นสืบสกุลมาจากคลาสที่กำหนดให้เป็น parameter ของ polymorphism เมทธอดตัวอย่างเช่นการนิยามคลาส TShape เป็นซูเปอร์คลาส และมีการสร้างคลาส TRectangle, TSquare, TEllipse โดยสืบสกุลจาก คลาส TShape และมีการประกาศเมทธอด Draw ของคลาส TRectangle, TSquare, TEllipse เพื่อให้การทำงานของเมทธอด Draw มีการทำงานที่แตกต่างกันในแต่ละคลาส เราสามารถสร้าง Polymorphism เมทธอดชื่อ DrawIt เพื่อใช้งานร่วมกันได้ดังนี้

```
   procedure DrawIt(Shape: TShape);
begin
    Shape.Draw;
end;
```

และคำสั่งในการเรียกใช้โพลีมอร์ฟิซึมเมทธอด DrawIt

```
var
  Rectangle: TRectangle;
  Square: TSquare;
  Ellipse: TEllipse;
begin
  Rectangle := TRectangle.Create;
  Square := TSquare.Create;
  Ellipse := TEllipse.Create;
  DrawIt(Rectangle);        // Draws a rectangle
  DrawIt(Square);           // Draws a square
  DrawIt(Ellipse);          // Draws an ellipse
  Rectangle.Free;
  Square.Free;
  Ellipse.Free;
end;
```

5.3.7 การจัดการเมทธอดแบบเวอร์ชวลและแบบไดนามิก

จากที่ได้กล่าวมาแล้วว่า เมทธอดที่รับค่าอินสแตนซ์ของซูเปอร์คลาสเป็นพารามิเตอร์ จะสามารถรับอินสแตนซ์ของสับคลาสเป็นพารามิเตอร์ได้ด้วย แต่หากในระหว่างซูเปอร์คลาสกับสับคลาสของอินสแตนซ์ที่ส่งไปเมทธอดเดียวกันมีการบังชื่อเกิดขึ้น ภาษาจาวาจะจัดการกับการบังชื่อฟิลด์ที่แตกต่างจากการบังชื่อเมทธอด จึงเป็นเหตุให้เรียกว่า shadowing และ overriding เพื่อให้ต่างกัน การผูกค่า (binding) เป็นการระบุว่าชื่อสมาชิกของอินสแตนซ์ที่ถูกส่งมาเป็นพารามิเตอร์ในเมทธอดนั้นหมายถึงชื่อใดกันแน่ ระหว่างชื่อที่บังกันอยู่ ในภาษาจาวา จะทำการผูกค่าในกรณีของ shadowing แบบ static นั่นคือ ทำในขณะคอมไพล์โปรแกรมแต่ทำการผูกค่าในกรณีของ overriding แบบ dyamic นั่นคือ ทำในขณะคอมไพล์โปรแกรม แสดงได้ตามตัวอย่างต่อไปนี้

```
// Shadowing.java
class A  {  public  int x = 10;  }

//variable x in class B is shadow variable
class B  extends A  { public int x = 20; }

class Shadowing  {
     static void testShadow (A v) {
     System.out.println (v.x);
}

Class Tester
     public static void main (String args[])   {
        testShadow (new A());
        testShadow (new B());
     }
}
```

คลาส A มีการประกาศตัวแปร x ถูกกำหนดค่าเริ่มต้นให้เป็น 10 ส่วนคลาส B มีการสืบทอดมาจากคลาส A ประกาศตัวแปร x และกำหนดค่าเริ่มต้นให้เป็น 20 ในลักษณะที่เป็น Shadow Variable ในคลาส Shadowing มี method testShadow () ที่รับอินสแตนซ์ของคลาส A เป็นพารามิเตอร์ ทำให้สามารถส่งอินสแตนซ์ของทั้งคลาส A และ B เป็นพารามิเตอร์ไปที่เมทธอดนี้ได้ และใน testShadow ( ) มีการพิมพ์ค่า x ของอินสแตนซ์ที่เป็นพารามิเตอร์ออกมา ในเมทธอด main( ) มีการสร้างอินสแตนซ์ของทั้งคลาส A และ คลาส B พร้อมกับส่งไปที่ testShadow ( ) ผลของการพิมพ์ค่า v.x ใน testShadow ( ) สำหรับทั้งสองอินสแตนซ์เป็น 10 ทั้งคู่ ที่เป็นเช่นนี้เพราะขณะที่คอมไพเลอร์ทำการคอมไพล์โปรแกรม และเมทธอด testShadow( ) พบว่าพารามิเตอร์ v เป็นอินสแตนซ์ของคลาส A ดังนั้นเมื่อมีการอ้างถึง v.x ในประโยค System.out.println( ) จึงจัดการสร้างโปรแกรมที่ให้ค่า v.x นี้เป็น x ตัว ที่อยู่ในคลาส A เรียกการผูกค่าระหว่างชื่อตัวแปร กับตัวแปรที่ถูกอ้างถึงในตอนคอมไพล์โปรแกรมอย่างนี้ว่า static binding ดังนั้นขณะที่โปรแกรมทำงาน ไม่ว่าอินสแตนซ์ที่ส่งมาที่ testShadow ( ) จะเป็นของคลาส A หรือคลาส B ก็ตาม โปรแกรมจะทำการพิมพ์ x ที่อยู่ในคลาส A ซึ่งเป็น 10 ออกมาเสมอ

สำหรับอีกตัวอย่างหนึ่งคือลักษณะของการทำ Overriding ซึ่งเป็นตัวอย่างหนึ่งของการทำ Late Binding ในกรณีนี้คอมไพเลอร์อาจจะไม่รู้ล่วงหน้ามาก่อนว่าโปรแกรมจะทำการเรียกใช้งานเมทธอดใดเพื่อให้ทำงาน จากตัวอย่างต่อไปนี้จะเห็นว่ามีการสร้างคลาส A เป็นคลาสแม่ และคลาส B เพื่อสืบทอดจาก A และในทั้งสองคลาสนี้จะมีเมทธอด print ที่มีการเขียนทับ Override แต่มีลักษณะการทำงานที่แตกต่างกัน ขึ้นอยู่กับว่าขณะที่โปรแกรมทำงาน (Run-Time) โปรแกรมจะมีการเรียกใช้งานเมทธอดจากคลาสไหน

ภายในคลาส Tester มีการสร้างเมทธอด test ซึ่งประกาศเป็นแบบสถิต หมายถึงเมทธอดที่ถูกสร้างขึ้นมาอัตโนมัติเมื่อมีการรันโปรแกรม ซึ่งเมทธอด test นี้รับค่าที่เป็นออบเจ็กต์ A เป็นพารามิเตอร์ ซึ่งจะเห็นว่าเมื่อมีการเรียกใช้เมทธอด test ภายใน main เราสามารถส่งค่าพารามิเตอร์เป็น new A ซึ่งจะไปเรียกเมทธอด print ที่อยู่ใน A หรือ new B ก็จะเรียก test ที่อยู่ใน B ได้

```
// Overriding.java
class A  {
    void print ( ) {
     for (int i = 0; i < 10; i++)
             System.out.println ("In A " + i);
    }
}

class B extends A {
    void print ( ) {
     for (int i = 0; i < 20; i++)
             System.out.println ("In B " + i);
    }
}

class Tester  {
    static void test (A v){
        v.print ();
    }

    public static void main (String args []) {
   test (new A ());
   test (new B ());
    }
}
```

ภาษา Delphi มีกลไกในการเชื่อม (Bind) ซึ่งเป็นการเรียกใช้เมทธอดทั้งแบบ Static Binding และ Late Binding ซึ่งอย่างหลังนี้สามารถนำมาใช้สำหรับเมทธอดที่เป็นแบบเวอร์ชวล (Virtual method) ซึ่งจะใช้หน่วยความจำมากกว่าการทำงานเป็นแบบไดนามิก Dynamic method แต่จะสามารถทำงานได้รวดเร็วกว่า เพราะจะมีการโหลด Virtual method เข้าสู่หน่วยความจำ ซึ่งในขั้นตอนการโหลดนี้จะมีการสร้างตาราง VMT (Virtual Method Table) เพื่ออ้างอิงไปยังตำแหน่งของเวอร์ชวลเมทธอดที่ถูกต้อง และข้อมูลในตาราง VMT จะมีการทำอินเด็กซ์เพื่อให้ค้นหาเมทธอดที่ต้องการเรียกใช้ได้อย่างรวดเร็ว ทำส่งผลให้การเรียกใช้ Virtual method จะเร็วกว่าการเรียกใช้

Dynamic method การกำหนดให้เมทธอดมีชนิดเป็น Virtual method หรือ Static method จะใช้ directive Virtual และ Dynamic ตามหลังการประกาศเมทธอดในขั้นตอนของการสร้างคลาส

ลักษณะของ Dynamic method จะคล้ายกับ Virtual method คือ จะมีการเชื่อม (bind) การเรียกใช้เมทธอด และเมทธอดที่จะถูกเรียกใช้ ในขณะที่โปรแกรมทำงาน แต่แตกต่างกันในส่วนของรายละเอียดคือ Dynamic method จะถูกโหลดเข้าสู่หน่วยความจำ เมื่อมีการ เรียกใช้เท่านั้นและจะมี การสร้างตาราง DMT (Dynamic Method Table) ในการอ้างถึงตำแหน่งของ Dynamic method ที่ถูกต้อง ซึ่งข้อมูลในตาราง DMT จะไม่มีการทำอินเด็กซ์ ดังนั้นการค้นหาจะทำแบบ linear จะทำให้ค้นหาช้ากว่าการค้นหาในตาราง VMT

```
type
   TSomeThing = class(TCompoenent)
   Private
          //  Private declarations
          Protected
                 Procedure  DoThis ; Virtual;
                 Procedure  DoThisAlso; Dynamic;
          Public
          //  Public declarations
          Published
          //  Published declarations
   End;
```

นอกจากนี้ในภาษา C# ยังมีคุณสมบัติพิเศษสำหรับการทำ Dynamic Binding นั่นก็คือการทำเมทธอด Reference Binding โดยมีลักษณะการทำงานแบบ Pointer ไปยังเมทธอดเรียกว่า Delegate คือการที่จัดเก็บเมทธอดในรูปของตัวแปรที่สามารถส่งเป็นค่าพารามิเตอร์ไปยังเมทธอดอื่นๆเพื่อเรียกใช้งานเมทธอด Delegate นี้ ตัวอย่างเช่นกรณีที่เรามีเมทธอดอยู่ 2 เมทธอดสำหรับการกำหนดค่า Name และ Surname ซึ่งทั้ง 2 เมทธอดนี้จริงแล้วทำงานในลักษณะเดียวกันคือการกำหนดค่าพารามิเตอร์ให้กับตัวแปรที่เป็นคุณสมบัติของออบเจ็กต์

```
public static string set_name(string name)
{
   return "My Name is : " + name;
}

public static string set_surname(string surname)
{
   return "My surname is : " + surname;
}
```

ซึ่งหากออบเจ็กต์ที่เราสร้างขึ้นนั้นจำเป็นต้องมีการกำหนดค่าคุณสมบัติในทุกตัวและแต่ละเมทธอดก็มีลักษณะการทำงานที่เหมือนหรือคล้ายกันเพียงแต่เปลี่ยนชนิดของตัวแปรไปดังนั้นเราสามารถใช้คุณสมบัติของ Delegate ได้ด้วยกำหนดเป็นเมทธอด Delegate ดังนี้

```
public delegate string SetString(string str);
```

โดยในที่นี่เราจะมองว่า SetString เป็นเมทธอดที่เปรียบเสมือนเป็นตัวแปรหนึ่ง และวิธีการเรียกใช้งาน

```
public static work(SetString x, string str)
{
   string sentence;
   sentence = x(str);
   Console.WriteLine(sentence);
}
```

ซึ่งจากตัวอย่างนี้จะเห็นว่าตัวแปร x จะทำหน้าที่ในการแม็บเมทธอด set_name() และ set_surname() ให้อัตโนมัติขึ้นอยู่กับพารามิเตอร์ (SetString delegate เมทธอด) ที่ส่งเข้ามา และโปรแกรมจะทำ การเลือกให้เองว่าจะเรียกใช้เมทธอดตัวไหน โดยลักษณะการทำงานนี้จะเห็นว่า delegate เหมือนกับการรับและส่งค่า Pointer ไปยังเมทธอด SetString นั่นเอง และตัวอย่างต่อไปนี้เป็นตัวอย่างของโปรแกรม DelegateExample ที่ใช้ตัวอย่างโดยภาษา C#

```
using System;
public class DelegateExample
{
   public delegate string SetString(string str);

   public static string set_name(string name)
   {
          return "My Name is : " + name;
   }

   public static string set_surname(string name)
   {
          return "My Surname is : " + name;
   }

   public static void work(SetString x, string str)
   {
          string sentence;
          sentence = x(str);
          Console.WriteLine(sentence);
   }

   public static void Main()
   {
          SetString x = new SetString(set_name);
          work(x, "John");

          x = new SetString(set_surname);
          work(x, "Doe");
   }
}
```

5.3.8 อินเตอร์เฟส (Interface)

อินเตอร์เฟสจัดได้ว่าเป็นคลาสอีกประเภทหนึ่ง ซึ่งมีลักษณะพิเศษคือ เมทธอดที่ประกาศภายในอินเตอร์เฟสนี้จะเป็น Abstract Method และ Member จะเป็นค่าคงที่เท่านั้น ภาษาเชิงวัตถุในปัจจุบันมีการสนับสนุนการทำงานของอินเตอร์เฟส แต่อาจจะต่างกันที่ลักษณะการนำไปใช้งาน ตัวอย่างเช่น ในภาษาจาวา และภาษา Delphi ที่ไม่สนับสนุนแนวคิดของการสืบทอดหลายชุด (Multiple Inheritance) เหมือนภาษา C++ ทั้งนี้เพราะคลาสของเมทธอดต่างๆ ที่ต้องการประมวลผลต้องปรากฏในช่วงของการคอมไพล์ เพื่อที่คอมไพเลอร์สามารถตรวจสอบ Signature ของเมทธอดได้ อย่างไรก็ตามโดยหลักการทั่วๆไป ของการเทคโนโลยีเชิงวัตถุอนุญาตให้หรือสนับสนุนหลักการของการสืบทอดหลายชุด ภาษาจาวาประยุกต์หลักการของการสืบทอดหลายชุดโดยใช้หลักการของตัวเชื่อมประสานหรืออินเตอร์เฟส (Interface) ที่จริงแล้วจาวาใช้อินเตอร์เฟสเป็นตัวเชื่อมประสานระหว่างคลาสที่ไม่มีความสัมพันธ์กัน แต่ต้องการที่จะติดต่อหรือส่งข้อมูลให้กับอีกออบเจ็กต์ของอีกคลาสหนึ่ง หรือต้องการบริการของอีกออบเจ็กต์ ดังนั้นอินเตอร์เฟสจึงเปรียบเสมือนเป็นช่องทางในการติดต่อสื่อสารกับอีกออบเจ็กต์หนึ่ง เพื่อให้อีกออบเจ็กต์ประมวลผลบางอย่างที่ตัวเองไม่สามารถกระทำได้

การสร้างอินเตอร์เฟสจะคล้ายกับการสร้างคลาส แต่วิธีในการระบุหรือการประกาศจะแตกต่างกันกับการสร้างคลาสและอินเตอร์เฟสจะมีข้อจำกัดมากกว่าของคลาสเพิ่ม ดังนี้

1. อินเตอร์เฟสไม่สามารถมีตัวแปรอินสแตนซ์ แต่สามารถมีตัวแปรค่าคงที่ได้ การประกาศตัวแปรค่าคงที่ไม่จำเป็นต้องประกาศเป็น public static final เพราะคอมไพเลอร์จะเติมให้โดยอัตโนมัติ
2. อินเตอร์เฟสไม่สามารถมีตัวแปรอินสแตนซ์ แต่สามารถมีตัวแปรค่าคงที่ได้ การประกาศตัวแปรค่าคงที่ไม่จำเป็นต้องประกาศเป็น public static final เพราะคอมไพเลอร์จะเติมให้โดยอัตโนมัติ
3. เมทธอดต่างๆ ในอินเตอร์เฟสจะเป็นเมทธอดแบบเชิงนามกล่าวคือ มีแต่ชื่อตัวแปรที่ส่งหรือผ่านค่า ค่าที่ส่งคืนซึ่งจะแตกต่างจากคลาสแบบเชิงนามที่เมทธอดในคลาสแบบเชิงนามไม่จำเป็นต้องเป็น เมทธอดแบบเชิงนามอย่างเดียว
4. เมทธอดต่างๆ ต้องมี access modifier เป็น public อย่างเดียวเท่านั้น

นอกจากนี้อินเตอร์เฟสในภาษาจาวายังสามารถสืบทอดจากอินเตอร์เฟสอื่นได้ เช่น ซูเปอร์อินเตอร์เฟส และสามารถเพิ่มได้หลายๆ อินเตอร์เฟสในคราวเดียวกัน ซึ่งจะแตกต่างจาก คลาสที่สืบทอดได้เพียงคลาสเดียวเท่านั้น การสืบทอดของอินเตอร์เฟสจะใช้หลักการเดียวกับการสืบทอดของคลาส แต่สิ่งที่สืบทอดได้คือตัวแปรค่าคงที่และเมทธอดเชิงนามเท่านั้น ตัวอย่างของการสร้างอินเตอร์เฟสในภาษาจาวา ดังนี้

```
public abstract interface MainInterface
{
       public abstract void interface1();
       public abstract void interface2();
}

class X implements MainInterface
{
       public void interface1()
        {
            System.out.println("use interface  1");
        }
       public void interface2()
        {
            System.out.println("use interface  2");
        }
}

public class TestInterface
{
       public static void main(String[] args)
        {
             System.out.println("Show how to use interface");
             X obj = new X();
             Obj.interface2();
             MainInterface obj1 = new X();
             Obj1.Interface1();
        }
}
```

อินเตอร์เฟสเป็นการกำหนดกลุ่มของ abstract method และ property ในลักษณะของ template คล้ายกับ abstract คลาส เพื่อให้คลาสอื่นสามารถอิมพลีเมนท์ได้อย่างสอดคล้อง สาเหตุหนึ่งที่ Delphi จำเป็นต้องมีอินเตอร์เฟสเนื่องจาก Delphi จะมีลักษณะเป็น Single Inheritance คือสามารถสืบสกุลจากคลาสต้นแบบได้เพียงคลาสเดียวเท่านั้น (เหมือนกับภาษา Java และภาษา C#) ดังนั้นจึงใช้กลไกของอินเตอร์เฟสมาช่วยในการทำ Multiple Inheritance

ใน Delphi สามารถสืบทอดจากซูเปอร์คลาสได้เพียงคลาสเดียว แต่สามารถอิมพลีเมนท์ได้หลายๆ อินเตอร์เฟสการประกาศคลาสอินเตอร์เฟสสามารถทำได้ดังตัวอย่างนี้

```
type
    IVehicle = Interface(IInterface)  //An interface definition
     // Properties and their functions
    function GetAge : Integer;
    function GetDateOfBirth : Date;
    property age : Integer read GetAge;
    property dateOfBirth : Date read GetDateOfBirth;
    // Non-property function
    function GetValue : Currency;
end;
```

หลังจากนั้นก็ทำการสร้างคลาสที่ทำการอิมพลีเมนท์อินเตอร์เฟสนี้ เช่น

```
TCar = Class (TInterfacedObject, IVehicle)
```

ในที่นี้คลาส TCar ทำหน้าที่อิมพลีเมนท์ interface Ivehicle โดยที่ TinterfacedObject เป็นคลาสที่ถูกสร้างไว้แล้วใน Delphi ซึ่งประกอบด้วยเมทธอดที่จำเป็นสำหรับการ implement interface

5.4 การพัฒนาโปรแกรมเชิงวัตถุด้วยภาษาแบบวิชวลโปรแกรมมิ่ง

ปัจจุบันการพัฒนาโปรแกรมเชิงวัตถุโดยภาษาที่สนับสนุนการทำงานทำงานแบบวิชวล เช่นภาษา C++, C#, Java และ Delphi ต่างล้วนมีซอฟต์แวร์ที่สนับสนุนการพัฒนาที่เป็น Intergrated Development Environment: IDE และต่างก็สนับสนุนการทำงานที่เป็นแบบ Visual Programming โดยที่เครื่องมือเหล่านี้มีการเชื่อมโยงกับไลบรารี ซึ่งในปัจจุบัน ภาษาต่างๆเหล่านี้ล้วนแล้วแต่มีคลังไลบรารีขนาดใหญ่ไม่ว่าจะเป็น Microsoft Foundary Class (MFC), .NET Framework Class Library ของบริษัทไมโครซอฟท์ หรือไลบรารี่ Visual Component Library (VCL) ของบริษัทบอร์แลนด์ หรือแม้กระทั่งของภาษาจาวาเองก็มีคลาสไลบรารี่ Java Class Library ขนาดใหญ่ ซึ่งเป็นคลาสที่สร้างมาพร้อมกับภาษาโดยค่ายต่างๆ ซึ่งการเรียกใช้คลาสต่างๆเหล่านี้จะกระทำผ่าน Application Programming Interface (API) ที่มีมาให้โดยภาษาหรือเจ้าของผลิตภัณฑ์นั้นๆ

ภาษา	เครื่องมือสำหรับพัฒนา (IDE)	คลาสไลบรารี
Visual Basic	Microsoft Visual Studio	.NET Framework Class Library
Visual C++	Microsoft Visual Studio	Microsoft Foundation Class (MFC), .NET Framework Class Library
Visual C#	Microsoft Visual Studio	.NET Framework Class Library
Visual Delphi	Borland Delphi	Visual Component Library (VCL)
Borland C++	Borland C++	Visual Component Library (VCL)
Java	Netbean, Eclipse	Java Package

ซึ่งประโยชน์ที่นักพัฒนาโปรแกรมจะได้รับก็คือคลังไลบรารี่ขนาดใหญ่ที่ถูกพัฒนาขึ้นมาให้เรียกใช้ได้โดยเป็นลักษณะของ Build-in Library หรือในบางครั้งเรียกว่า Package หรือ Component ซึ่งไลบรารี่เหล่านี้ถูกพัฒนาขึ้นมาโดยอาศัยแนวคิดของของการพัฒนาซอฟต์แวร์เชิงวัตถุที่ได้กล่างมาในข้างต้น และคลาสต่างๆเหล่านี้ก็คือออบเจ็กต์ต่างๆที่สามารถนำมาเชื่อมโยงและใช้งานร่วมกับซอฟต์แวร์ที่เราพัฒนา

```
using System;
using System.Net.Sockets;

public class AsynchIOServer
{
    public static void Main()
    {
       TCPListener tcpListener = new TCPListener(10);
       tcpListener.Start();
       Socket socketForClient = tcpListener.Accept();
```

```
            if (socketForClient.Connected())
            {
                  Console.WriteLine("Client connected");
            }

            socketForClient.Close();
            Console.WriteLine("Exiting...");
        }
    }
```

จากตัวอย่างต่อไปนี้ จะเห็นว่าการพัฒนาโปรแกรมที่เกี่ยวข้องกับการเชื่อมต่อกับระบบเครือข่ายโดยอาศัย Socket Programming สามารถทำได้โดยเรียกใช้ไลบรารีที่เกี่ยวข้องกับซอกเก็จ using System.Net.Sockets ซึ่งทำให้เราสามารถสร้างออบเจ็กต์ TCPListerner และเรียกใช้งานเมทธอด Start, Accept ได้ โดยในหลักการมองว่าออบเจ็กต์ TCPListener เป็นกล่องดำที่จะรับและส่งค่าอะไรออกมาจากกล่องดำนี้บ้าง ซึ่งการเรียกใช้งานผู้พัฒนาไม่จำเป็นต้องเข้าใจถึงรายละเอียดการทำงานภายใน แต่ในภาพรวมซึ่งจะต้องรู้ว่าเมทธอดที่มีบริการให้เรียกใช้ภายในออบเจ็กต์ TCPListener มีอะไรบ้าง และทำงานอะไร โดยผู้พัฒนาจำเป็นต้องศึกษาถึงเอกสาร API ที่ประกอบกับไลบรารี่ โดยที่เพิ่มความรวดเร็วในการพัฒนาโปรแกรมในปัจจุบันเป็นอย่างมาก ดังตัวอย่าง API ต่อไปนี้

System.Net.Sockets Namespace[4]

.NET Framework 1.1

The **System.Net.Sockets** namespace provides a managed implementation of the Windows Sockets (Winsock) interface for developers who need to tightly control access to the network.

The TCPClient, TCPListener, and UDPClient classes encapsulate the details of creating TCP and UDP connections to the Internet.

Class	Description
IrDAClient	Provides connection information, and creates client connection objects for opening and closing connections to a server.
IrDADeviceInfo	Provides information about available servers and ports obtained by the client during a discovery query.
IrDAListener	Places a socket in a listening state to monitor connections from a specified service or network address.
LingerOption	Specifies whether a Socket will remain connected after a call to Close and the length of time it will remain connected, if data remains to be sent.
MulticastOption	Contains IPAddress values used for joining and dropping multicast groups.
NetworkStream	Provides the underlying stream of data for network access.

4 Online Source: http://msdn.microsoft.com/en-us/library/system.net.sockets(v=vs.71).aspx

Socket	Implements the Berkeley sockets interface.
SocketException	The exception that is thrown when a socket error occurs.
TcpClient	Provides client connections for TCP network services.
TcpListener	Listens for connections from TCP network clients.
UdpClient	Provides User Datagram Protocol (UDP) network services.

การพัฒนาซอฟต์แวร์โดยอาศัยเครื่องมือ Visual Programming ทำให้เราสามารถพัฒนาโปรแกรมได้อย่างรวดเร็วโดยอาศัยออบเจ็กต์ต่างๆที่มีมาในคลาสไลบรารีของภาษา และเครื่องมือ IDE และสิ่งที่สำคัญต่อการพัฒนาซอฟต์แวร์ในปัจจุบันก็คือ การพัฒนาส่วนการเชื่อมต่อกับผู้ใช้งาน (User Interface: UI) ซึ่งภาษาที่เป็นวิชวลโปรแกรมในปัจจุบันจะมีไลบรารีที่ให้นักพัฒนาสามารถสร้างหน้าจอในลักษณะที่เป็น Windows Form และกลุ่มของออบเจ็กต์ที่ใช้ในการสร้าง UI เรียกว่าคอนโทรล (Control) ที่สามารถนำมาพัฒนาเป็นโปรแกรมต้นแบบ Prototype ให้แก่ผู้ใช้งานเพื่อให้ได้เห็นผลลัพธ์สุดท้าย (Program Output) ของโปรแกรมได้อย่างรวดเร็ว ตัวอย่างเช่น ในภาษา C# จะมีไลบรารี System.Windows.Forms และภาษาจาวาจะมีไลบรารี javax.swing ทั้งฟอร์มและคอนโทรล ต่างก็ถูกพัฒนาให้เป็นออบเจ็กต์ต่างๆ บางออบเจ็กต์ก็จะมีคุณสมบัติเป็นคอนเทนเนอร์ (Container) คือสามารถที่จะบรรจุคอนโทรลอื่นๆเช่น ออบเจ็กต์ Forms, GroupBox, FlowLayOut, ToolStrip, StatusStrip, ToolStripContainer เป็นต้น

การสร้างออบเจ็กต์ที่เป็น UI ขึ้นมาใช้งานโดยส่วนใหญ่จะเริ่มต้นจากการสร้างฟอร์มเพื่อสำหรับเป็นหน้าแรกเรียกว่า Startup Form โดยที่ฟอร์มนี้มักจะเป็นฟอร์มเปล่า โดยที่ยังไม่มีคอนโทรลใดๆทำงานซึ่งเมื่อมีการสร้างฟอร์มแรกขึ้นมา โปรแกรม IDE จะทำการสร้างโปรแกรมโค๊ดที่จำเป็น โดยที่ผู้ใช้งานไม่จำเป็นต้องเขียนโค๊ดเหล่านี้เอง ดังตัวอย่างต่อนี้ซึ่งโปรแกรม IDE เขียนขึ้นให้เองอัตโนมัติ

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
```

```
            {

            }
        }
    }
```

5.4.1 การสร้างคอนโทรลออบเจ็กต์

การสร้างคอนโทรลออบเจ็กต์ในภาษาที่สนับสนุนการทำงานแบบวิชวลนั้น สามารถกระทำได้โดยอาศัยเครื่องมือการออกแบบ Design Tool ใน IDE ซึ่งส่วนใหญ่จะเป็นลักษณะของการลากและวาง ซึ่งในเครื่องมือเหล่านี้จะมีคอนโทรลต่างๆให้เลือกใช้เพื่อสำหรับสร้างส่วนที่เชื่อมต่อกับผู้ใช้ UI พื้นฐาน เช่น Pointer, Button, CheckBox, CheckListBox, ComboBox, DateTimePicker, Label, LinkLabel, ListBox, ListView, TextBox เป็นต้น เมื่อทำการลากและวางคอนโทรลเหล่านี้ลงบนฟอร์ม โปรแกรมก็จะทำการสร้างโค๊ดขึ้นมาในส่วนต่างๆของโปรแกรมให้อัตโนมัติเช่นเดียวกัน ดังตัวอย่างต่อไปนี้ซึ่งใช้ภาษา C# เป็นการแสดงให้เห็นถึงโปรแกรมโค๊ดเมื่อมีการลากคอนโทรล Button มาใช้งาน โดยที่จะปรากฏโปรแกรมโค๊ด ดังนี้

```
private System.Windows.Forms.Button button1;

private void button1_Click(object sender, EventArgs e)
{

}
```

ซึ่งโปรแกรมโค๊ดที่สร้างขึ้นมาให้นี้ เรียกว่าหลักการของ Code Behind เนื่องจากคอนโทรลต่างๆที่สร้างขึ้นมาจะมีโค๊ดซ่อนอยู่เบื้องหลัง ซึ่งจะผูกติดอยู่กับเหตุการณ์ที่จะเกิดขึ้น เรียกว่า Event ซึ่งการพัฒนาโปรแกรมที่เป็น Windows Base Application นั้นในบางครั้งเราจะเรียกว่าการพัฒนาโปรแกรมแบบ Event-Driven Programming เนื่องจากภายในโปรแกรมหนึ่งๆ อาจจะมีคอนโทรลอยู่หลายๆตัวด้วยกัน โดยที่ผู้ใช้งานอาจจะใช้เมาส์คลิกเพื่อที่จะเลือกทำงานกับคอนโทรลใดคอนโทรลหนึ่ง ซึ่งก็จะเกิดเหตุการณ์ (Event) หรือ Action สำหรับคอนโทรลนั้น ตัวอย่างเช่น เมื่อผู้ใช้เลือกปุ่ม “Open File” โปรแกรมก็จะให้ผู้ใช้เลือกชื่อไฟล์ที่ต้องการเปิดอ่าน หรือเมื่อผู้ใช้คลิกเลือกรายงานเพลงใน ListBox ก็จะทำการเปิดเพลงที่ผู้ใช้เลือก เป็นต้น ซึ่งการพัฒนาโปรแกรมที่เป็นลักษณะ Event-Driven นั้นจะต่างจากการพัฒนาโปรแกรมสมัยก่อนที่เป็นลักษณะ Top-Down คือโปรแกรมจะมีการทำงานที่เป็นลำดับขั้นตอน แต่ Event-Driven นั้น โปรแกรมทำงานแบบไม่เป็นลำดับ และเหตุการณ์ที่เกิดขึ้นก็จะขึ้นอยู่กับว่าผู้ใช้จะเลือกคอนโทรลตัวไหนเพื่อทำงาน

การทำงานของโปรแกรมแบบ Event-Driven ดังกล่าว จะเห็นว่าในตัวอย่างที่แสดง เมื่อทำการสร้างคอนโทรลใดๆก็ตามขึ้นมาในฟอร์ม โปรแกรม IDE จะทำการสร้างโค๊ดขึ้นมา 2 ส่วนได้แก่ 1)เพื่อการเฝ้าฟังเหตุการณ์ใดๆที่จะเกิดขึ้นกับคอนโทรลนั้นๆ เราเรียกว่า “Event” ซึ่งนั่นได้แก่เหตุการณ์ที่ผู้ใช้ นำเมาส์ไปคลิก เรียกว่า OnClick หรือเหตุการณ์อื่นๆเช่น OnClick, OnDoubleClick, OnMouseOver ... สำหรับในภาษา C# และ Visual Basic โค๊ดส่วนนี้ได้ถูกซ่อนเอาไว้ แต่กรณีที่เป็น C++ หรือ Java จะต้องสร้างเป็น

อินเตอร์เฟส Listener ขึ้นมาเพื่อทำการเฝ้าฟังเหตุการณ์เอง และ 2) เพื่อทำการจัดการเมื่อมีเหตุการณ์นั้นๆ เกิดขึ้น เรียกว่าการ “Handle Event” ซึ่งจากตัวอย่างการเขียน Event-Handle ในภาษาจาวาดังต่อไปนี้

```
JButton myButton = new JButton("Click Me");
myButton.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            doAction();  // code perform when button is pressed
        }
    }
);
```

จากตัวอย่างนี้จะเห็นว่าการเฝ้าฟังเหตุการณ์ที่จะเกิดขึ้นกับปุ่ม นั้นจะต้องทำการลงทะเบียน Register คอนโทรล Button นี้ก่อนด้วยคำสั่ง addActionListener และกรณีที่มีเหตุการณ์การกดปุ่มเกิดขึ้นก็จะ Handle โดยคำสั่ง actionPerformed เพื่อทำงานในส่วนที่ผู้พัฒนาต้องการทำอะไรเมื่อผู้ใช้กดปุ่มนี้

5.4.2 คุณสมบัติของคอนโทรลออบเจ็กต์

เมื่อทำการสร้างคอนโทรลโดยวิธีการลากและวางดังตัวอย่างข้างต้น โปรแกรม IDE จะทำการสร้างโค๊ดหลายอย่างเพื่อเป็นโค๊ดสำหรับการตั้งต้นในแก่นักพัฒนา เช่น การเรียกใช้ไลบรารี่ที่จำเป็น การประกาศตัวแปร การสร้างอินสแตนซ์ การสร้างการเฝ้าฟังเหตุการณ์ และการจัดการเมื่อมีเหตุการณ์เกิดขึ้น โดยที่ผู้พัฒนาเพียงจะต้องคำนึงถึงโปรแกรมโค๊ดที่เกี่ยวข้องกับเนื้องานเท่านั้น จากตัวอย่าง

```
private void button1_Click(object sender, EventArgs e)
{
       MessageBox.Show(“You click Button 1”);
}
```

จะเป็นการ Handle เมื่อมีการกดปุ่ม Button ก็จะทำการแสดงคำว่า “You click Button 1” ซึ่งในส่วนนี้เป็นส่วนที่ผู้ใช้งานจะต้องทำการเขียนโค๊ดที่เกี่ยวข้องขึ้นมาเอง แต่นั่นก็หมายความว่าโปรแกรม IDE ได้เขียนโปรแกรมโค๊ดส่วนใหญ่ให้กับทางผู้พัฒนาแล้ว

การประกาศตัวแปร button1 จากออบเจ็กต์ Button ซึ่งเป็นหนึ่งออบเจ็กต์ที่อยู่ภายในโครงสร้างของ Class Hierarchy ซึ่งเราจะเห็นจากลำดับชั้นของการสืบทอดสืบสกุล ดังนั้นการอ้างอิงสามารถอ้างจาก Full Path ได้เป็น System.Windows.Forms.Button ซึ่งมีการประกาศใช้ Access Modifier เป็นแบบส่วนตัว Private ดังนี้ `private System.Windows.Forms.Button button1;` โดยที่ Button นั้นเป็นออบเจ็กต์ที่มีคุณสมบัติสองส่วน Data Member เช่นสี ชื่อ ตำแหน่ง ขนาด ฯลฯ และ Behavior (Event) เช่น Click, DragDrop, KeyPress ... ดังที่ได้กล่าวไปในบทก่อนหน้า ซึ่งสามารถปรับแก้ไขได้โดยผ่านทาง User Interface ที่ผู้ช่วยเหลือผู้พัฒนาเช่นกัน

บทที่ 6 – การตรวจจับข้อผิดพลาด

ซอฟต์แวร์ที่ถูกพัฒนาขึ้นมานั้นต่างมีวัตถุประสงค์การทำงานอย่างชัดเจน ซึ่งการทำงานจะต้องมีฟังก์ชันการทำงานที่ถูกต้องตามขอบเขตที่กำหนดให้ทำงานแล้ว ซอฟต์แวร์ควรจะต้องเป็นซอฟต์แวร์ที่มีคุณภาพด้วย ซึ่งย่อมหมายถึงการที่ซอฟต์แวร์มีคุณสมบัติต่างๆ ดังต่อไปนี้

- มีฟังก์ชันการทำงานที่ครบถ้วนตามวัตถุประสงค์และ ตามความต้องการของผู้ใช้งาน (User Requirement)
- มีหน้าจอและการเรียกใช้งานเพื่อให้ง่ายต่อการใช้งาน (User Friendly) โดยมีการเอื้ออำนวยความสะดวกในการเข้าถึงข้อมูลหรือฟังก์ชันงานโดยผ่านเมนูและคำสั่งต่างๆ หรือแม้กระทั่งคำสั่งลัด (Short Cut Key) ต่างๆ
- มีการประมวลผลข้อมูลที่ถูกต้องแม่นยำ (Accuracy)
- มีระยะเวลาในการตอบสนองการทำงาน (User Response Time) ที่เหมาะสม

ทั้งนี้โดยส่วนใหญ่แล้วซอฟต์แวร์จะต้องมีการทำงานร่วมกับผู้ใช้งาน ซึ่งในความหมายคือการตอบสนองในลักษณะที่เป็นการโต้ตอบ (Interactive) และการแสดงข้อความโต้ตอบถึงสิ่งที่ซอฟต์แวร์กำลังปฏิบัติ เช่น การรับอินพุท การแสดงสถานะของการประมวลผล และเมื่อเสร็จสิ้นการประมวลผล หรือแม้กระทั่งกรณีที่มีข้อผิดพลาดใดๆก็ตามเกิดขึ้นกับซอฟต์แวร์ ซึ่งจะทำให้ผู้ใช้งานได้เข้าใจถึงกระบวนการและสถานะการทำงานของซอฟต์แวร์ด้วย

สำหรับการพัฒนาโปรแกรมเชิงวัตถุ การดักจับข้อผิดพลาดของโปรแกรมถือว่าเป็นส่วนที่สำคัญ เนื่องจากข้อผิดพลาดที่เกิดจากการพัฒนาโปรแกรมมีโอกาสเกิดขึ้นได้เสมอ ดังนั้นเมื่อมีข้อผิดพลาดเกิดขึ้น ระบบจะทำการโยนข้อผิดพลาดออกมา (Raise Exception) ทำให้โปรแกรมหยุดทำงาน หรือส่งผลกระทบต่อระบบปฏิบัติการ และเครื่องคอมพิวเตอร์โดยรวม ซึ่งโดยทั่วไป การพัฒนาโปรแกรมต่างๆล้วนแต่มีโอกาสที่จะเกิดข้อผิดพลาดได้ ซึ่งข้อผิดพลาดเหล่านี้อาจจะเกิดขึ้นจากตัวโปรแกรมเองที่ทำงานได้ไม่ถูกต้อง หรือข้อผิดพลาดที่เกิดจากตัวผู้ใช้งานเอง ซึ่งปัญหาต่างๆเหล่านี้ควรจะต้องมีการป้องกัน หรือมีการดักจับข้อผิดพลาดก่อนที่จะมีข้อผิดพลาดเกิดขึ้น หรืออีกลักษณะหนึ่งคือการแก้ไขปัญหาในกรณีที่มีข้อผิดพลาดเกิดขึ้นแล้ว สำหรับในการพัฒนาโปรแกรม ข้อผิดพลาดนั้นสามารถเกิดขึ้นได้อยู่ 3 ลักษณะด้วยกันคือ

1. ข้อผิดพลาดที่เกิดขณะคอมไพล์โปรแกรม (Compile Error) ซึ่งเป็นข้อผิดพลาดที่เกิดขึ้นขณะที่ทำการคอมไพล์โปรแกรมจากโปรแกรมให้เป็นภาษาเครื่อง โดยโปรแกรมที่ทำหน้าที่เป็นคอมไพล์เลอร์จะทำหน้าที่ในส่วนนี้ ซึ่งโปรแกรมจะทำการตรวจสอบโค๊ดของเราว่าเป็นไปตามไวยากรณ์ภาษา (Syntax) ที่เขียนหรือไม่ ซึ่งในแต่ละภาษาที่เขียนอาจจะมีไวยากรณ์ที่แตกต่างกัน เช่น การเขียนเมทธอด Calculate ในภาษา C#

```
public int Calculate (int a, int b)
{
   return (a + b);
}
```

จะเห็นได้ว่าการประกาศชื่อหรือโครงสร้างของเมทธอดนั้นจะต้องประกอบด้วย modifier ที่เป็น public หรือ private และการรีเทิร์นค่ากลับ ตามด้วยชื่อของเมทธอด และการใช้เครื่องหมายปีกกาสำหรับเปิดปิดบล็อกของเมทธอด หรือการใช้เครื่องหมาย ‘;’ เพื่อบ่งบอกถึงจุดสิ้นสุดของชุดคำสั่ง (Statement) ซึ่งหากเราเขียนส่วนใดส่วนหนึ่งผิดพลาดไปโปรแกรมคอมไพล์เลอร์ก็จะแจ้งให้กับเราทราบว่าโปรแกรมมีข้อผิดพลาดในส่วนใด หรือผิดพลาดในบรรทัดใดของโปรแกรม ซึ่งในบางครั้งเราเรียกข้อผิดพลาดในลักษณะนี้ว่าเป็น Syntax Error เนื่องจากเป็นข้อผิดพลาดจากไวยากรณ์ของโปรแกรมภาษา

2. ข้อผิดพลาดที่เกิดขึ้นขณะที่โปรแกรมทำงาน (Runtime Error) เป็นข้อผิดพลาดที่เกิดขึ้นในระหว่างที่โปรแกรมมีการทำงานอยู่ ซึ่งในลักษณะนี้โปรแกรมนั้นผ่านการคอมไพล์มาอย่างถูกต้อง คือเขียนโปรแกรมได้อย่างถูกต้องตามหลักไวยากรณ์ แต่ขณะที่โปรแกรมทำงานอาจจะยังมีข้อผิดพลาดเกิดขึ้นได้ ซึ่งข้อผิดพลาดนี้มักจะเกิดจากการที่โปรแกรมทำงานได้ไม่ครอบคลุมลักษณะต่างๆที่กำหนดไว้ตั้งแต่แรก ตัวอย่างเช่น การรับค่าอินพุทจากผู้ใช้งานที่อาจจะรับเป็นตัวเลข ตัวอักษร หรือข้อความเป็นต้น โดยที่โปรแกรมอาจจะไม่ได้มีการคาดการณ์ไว้ล่วงหน้า ซึ่งกรณีนี้โปรแกรมจะต้องมีการแปลงค่าชนิดของข้อมูลไปเป็นประเภทต่างๆก่อนที่จะนำไปคำนวณหรือประมวลผล สำหรับข้อผิดพลาดที่เกิดขึ้นขณะที่โปรแกรมทำงานนั้นอาจจะก่อให้เกิดผลข้อผิดพลาดที่เล็กน้อย ที่แสดงข้อความของข้อผิดพลาดและโปรแกรมยังสามารถทำงานต่อไปนี้ จนกระทั่งถึงในระดับที่โปรแกรมเกิดข้อผิดพลาดที่ร้ายแรงจนทำให้โปรแกรมหยุดจนไม่สามารถทำงานต่อไปได้ (Program Crash) เช่นการใช้หน่วยความจำมากเกินไป (Out-of-Memory) หรือไม่ได้จองหน่วยความจำ (Memory Allocation Error)

3. ข้อผิดพลาดที่เกิดขึ้นจากตรรกะของโปรแกรม (Logic Error) ซึ่งข้อผิดพลาดประเภทนี้อาจจะดูเหมือนว่าโปรแกรมสามารถทำงานได้ตามปกติ แต่เมื่อโปรแกรมทำงานแล้ว โปรแกรมไม่สามารถทำงานหรือประมวลผลได้อย่างถูกต้อง ตัวอย่างจากข้อผิดพลาดในลักษณะนี้เช่นการใช้เครื่องหมายตรรกะ AND, OR, NOT และการใช้เครื่องหมายเปรียบเทียบสำหรับเงื่อนไขต่างๆ เช่น

```
if ((a < 10) && (a > 20))
{
      //statement
}
```

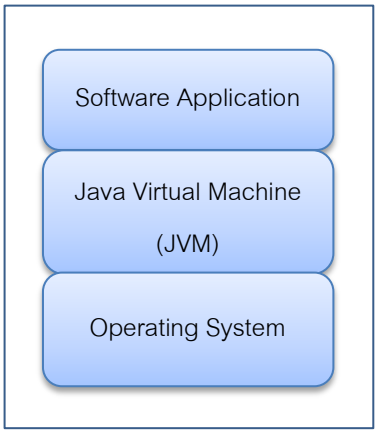
ซึ่งในกรณีนี้ โอกาสที่ a จะมีค่าน้อยกว่า 10 และมีค่ามากกว่า 20 ในขณะเดียวกันแทบจะเป็นไปไม่ได้เลย ซึ่งก็จะทำให้ชุดคำสั่งไม่เกิดการทำงาน

6.1 หลักการทำงานเมื่อโปรแกรมมีข้อผิดพลาด

ในการตรวจสอบข้อผิดพลาดนั้นสำหรับภาษาในเชิงวัตถุจะขึ้นอยู่กับคุณสมบัติของภาษา แต่การตรวจจับข้อผิดพลาดจะมีอยู่ 2 ลักษณะคือการตรวจสอบ (Checked) กับการไม่ตรวจสอบ (Unchecked Exception) ข้อผิดพลาด ในภาษาจาวานั้นจะสนับสนุนการทำงานทั้ง 2 แบบ แต่ตัวอย่างบางภาษาเช่น ภาษา C# จะสนับสนุนเพียงแค่แบบไม่ตรวจสอบ (Unchecked) เพียงอย่างเดียว สำหรับข้อแตกต่างระหว่างการตรวจสอบข้อผิดพลาดทั้ง 2 ประเภทนี้คือ

1) วิธีการตรวจเช็ค (Checked exception) จะต้องใช้วิธีการดักจับอย่างชัดเจน หรือใช้วิธีการเผยแพร่ข้อผิดพลาดโดยใช้ try-catch-finally
2) วิธีการแบบไม่ตรวจเช็ค (Unchecked exception) ซึ่งเป็นลักษณะของการเผยแพร่ข้อผิดพลาด (Propagate) เป็นการส่งต่อข้อผิดพลาดนั้นๆออกไปยังเมทธอดที่เรียกใช้

โดยหลักการทำงานของโปรแกรมนั้นจะต้องทำงานอยู่บนระบบปฏิบัติการ ดังเช่นโปรแกรมจาวาจะต้องทำงานอยู่บน JVM (Java Virtual Machine) อีกทีหนึ่งซึ่งสามารถมองเป็นเลเยอร์ได้ดังนี้



รูป **6.1** ลำดับชั้นการทำงานของซอฟต์แวร์บน **Java Virtual Machine**

ปัจจุบันระบบปฏิบัติการจะมีความสามารถรองรับการทำงานแบบ Multi-Tasking โดยแบ่งโปรแกรมที่ทำงานเป็นแบบโพรเซส (Process) ซึ่งในแต่ละโพรเซสอาจจะมีโพรเซสย่อยภายในได้อีกด้วย ที่เรียกว่าเทรด (Thread) โดยหลักการของการแบ่งโปรแกรมเป็นแบบโพรเซสนี้ จะทำให้หลายๆโปรแกรมสามารถทำงานและประมวลผลได้พร้อมกัน (โดยการจัดการของระบบปฏิบัติการที่อาศัยหน่วยประมวลผล 1 หน่วยหรือมากกว่าก็ได้) สำหรับโปรแกรมที่พัฒนาขึ้นมาก็เช่นเดียวกัน จะถูกระบบปฏิบัติการมองว่าเป็นโพรเซสหนึ่งๆ จากตัวอย่างจะเห็นได้ว่าโปรแกรมทำงานในอยู่บน JVM และ JVM ทำงานอยู่บนระบบปฏิบัติการอีกชั้นหนึ่ง ลักษณะของการดักจับข้อผิดพลาดจะมีการทำงานคือการดักจับในตัวโปรแกรมเป็นอันดับแรกก่อน แต่หากโปรแกรมไม่มีการดักจับก็จะส่งต่อมายัง JVM และหากไม่เช่นนั้นแล้วก็จะเป็นการดักจับในระดับของระบบปฏิบัติการในลักษณะที่เป็นทอดๆไป

วิธีการดักจับข้อผิดพลาดของโปรแกรมที่มักจะนิยมทำกันก็คือการป้องกันข้อผิดพลาดที่อาจจะเกิดขึ้นก่อนการทำงานอย่างใดอย่างหนึ่งโดยอาศัยเงื่อนไขที่ทำการตรวจสอบเช่น คำสั่ง if ... then หรือ switch case ... ซึ่งเป็นวิธีที่ได้รับความนิยม แต่อย่างไรก็ตามวิธีนี้เหมาะสำหรับกรณีที่เรารู้ล่วงหน้าว่าจะมีข้อผิดพลาดอะไรเกิดขึ้นบ้าง เช่นกรณีที่ค่าอินพุทจะเกินค่าที่กำหนดหรือจะมีตัวหารด้วยตัวเลขศูนย์เป็นต้น แต่สำหรับกรณีที่มีเหตุการณ์ที่ไม่สามารถคาดการณ์ได้ล่วงหน้า ซึ่งมักจะเกิดในขณะที่โปรแกรมทำงาน (Runtime) โปรแกรมควรจะต้องมีการดักจับข้อผิดพลาดเหล่านี้ด้วย

โดยหลักการทำงานของโปรแกรมที่เป็นออบเจ็กต์ โปรแกรมจะทำงานอยู่บนเฟรมเวิร์กของภาษานั้นๆ เช่นภาษาจาวาจะทำงานอยู่บน Java Virtual Machine และโปรแกรมที่อยู่ในตระกูลดอทเน็ทของไมโครซอฟท์ก็จะทำงานอยู่บน .NET Framework ซึ่งจะมีคุณสมบัติในการดักฟังข้อผิดพลาดต่างๆที่อาจจะเกิดขึ้น โดยเฟรมเวิร์กเหล่านี้มีหลักการทำงานของออบเจ็กต์ต่างๆจะถูกสร้างขึ้นมาเพื่อสืบทอดจากคลาส Throwable ดังตัวอย่างโครงสร้างของคลาส และจากที่ได้อธิบายไปก่อนหน้าที่ว่าทุกออบเจ็กตที่สร้างขึ้น จะสืบทอดจากคลาสออบเจ็กต์ ซึ่งคลาสเหล่านี้ในเชิงของเทคโนโลยีการพัฒนาโปรแกรมเชิงวัตถุจะมีความสามารถในการสร้างตัวดักข้อผิดพลาด (Raise Error Exception) ขึ้นมาทั้งสิ้น ซึ่งชนิดข้อผิดพลาดเหล่านี้จะขึ้นอยู่ชนิดของข้อผิดพลาด ซึ่งจะเป็นลักษณะของการโยน (Throw) ข้อผิดพลาดออกมาจากการทำงานผ่านขั้นตอนการเรียก ใช้งานเมทธอด (Method Caller) ซึ่งจากตัวอย่างของโครงสร้างเมทธอดที่ประกาศไว้ล่วงหน้า (Forward Declaration) ดังต่อไปนี้

```
public static Class forName(String className) throws ClassNotFoundException
public Object newInstance() throws InstantiationException, IllegalAccessException
```

จากตัวอย่าง การเรียกเมทธอด forName ในภาษาจาวาเพื่อที่จะทำการสร้างคลาสสำหรับ className ใหม่ขึ้นมา จะเห็นได้ว่าเมทธอดนี้จะทำการสร้างข้อผิดพลาดชื่อว่า ClassNotFoundException ซึ่งเป็นข้อผิดพลาดที่ใช้สำหรับการดักจับกรณีที่หาคลาสนั้นๆไม่พบ หรือเช่นเดียวกันกับเมทธอดสำหรับสร้างอินสแตนซ์ใหม่ซึ่งก็จะมีการดักจับข้อผิดพลาดชื่อว่า InstantiationException และ IllegalAccessException ออกมาจากเมทธอดกรณีที่มีข้อผิดพลาดใดๆก็ตามเกิดขึ้นขณะที่มีการเรียกใช้งานเมทธอด[5]

6.2 ข้อผิดพลาด Error Exception เกิดขึ้นได้อย่างไร

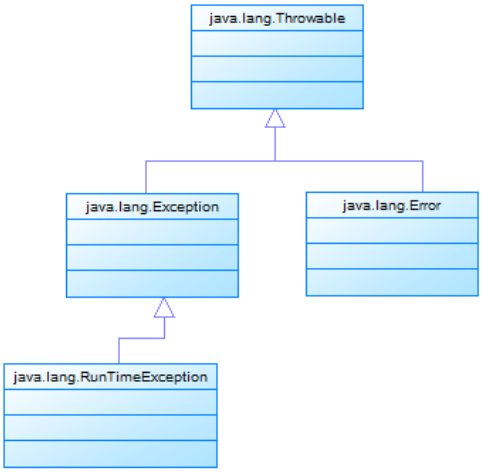
สำหรับการพัฒนาโปรแกรมเชิงวัตถุนั้น โดยส่วนใหญ่แล้วเราจะเป็นผู้ที่เรียกใช้คลาสและเมทธอดที่มีอยู่ในคลาสไลบรารี่ที่มีมาให้ใน Packages ในภาษาจาวา หรือ .NET Framework สำหรับภาษาในตระกูลไมโครซอฟท์ และ Visual Control Language (VCL) สำหรับภาษา Delphi ซึ่งเป็นไลบรารีมาตราฐานที่มาพร้อมกับคอมไพเลอร์ของแต่ละภาษา โดยที่โครงสร้างการออกแบบของคลาสและวิธีการพัฒนาคลาสเหล่านี้

5 สำหรับเอกสารที่ใช้เป็นคู่มือสำหรับการอ้างอิงว่าโครงสร้างการทำงานของเมทธอดและข้อผิดพลาดที่จะมีการดักจับนั้นสามารถอ้างอิงได้จาก Java Class Library Reference

จะมีการตรวจสอบข้อผิดพลาดที่อาจจะเกิดขึ้นในระหว่างที่มีการเรียกใช้คลาสหรือเมทธอดนั้นๆ โดยที่โปรแกรมโค๊ดจะทำการดักจับข้อผิดพลาดและทำการสร้าง Error Exception ขึ้นมา โดยมีวัตถุประสงค์หลักคือเพื่อป้องกันไม่ให้เกิดข้อผิดพลาดที่อาจจะเกิดขึ้นระหว่างการเรียกใช้เมทธอดก่อให้เกิดความเสียหายหรือส่งผลกระทบต่อโปรแกรมโดยรวม ซึ่งความผิดพลาดในบางประเภทอาจมีผลเฉพาะโปรแกรมที่กำลังทำงานอยู่เท่านั้น หรืออาจจะมีความรุนแรงต่อระบบปฏิบัติการ ทำให้เครื่องคอมพิวเตอร์หยุดทำงาน ทำให้ต้องมีการปิดและเปิดเครื่องคอมพิวเตอร์ใหม่

```
public class Point
{
        …
        public void setLatitude(double lat)
        {
                if ((lat < -90) || (lat > 90))
                // is it within the range
                {
                        throw new IllegalArgumentException();
                }
                else
                {
                        latitude = lat;
                }
        }
}
```

จากตัวอย่างนี้จะเห็นว่าโปรแกรมจะทำการตรวจสอบว่าค่าพารามิเตอร์ที่รับเข้ามานั้นอยู่นอกเหนือช่วงข้อมูลน้อยกว่า -90 หรือมากกว่า 90 หากไม่อยู่ในช่วงดังกล่าวก็ให้หยุดการทำงาน โดยจะ Raise Error ข้อผิดพลาดชื่อว่า IllegalArgumentException และโยน (Throw) กลับไปยังเมทธอดที่เรียก (Caller Method) setLatitude นี้ สำหรับ IllegalArgumentException จริงแล้วเป็นคลาสที่ทำหน้าที่ในการแสดงข้อผิดพลาด ซึ่งจัดได้ว่าเป็นคลาสประเภทหนึ่งที่มีคุณสมบัติพิเศษ คือคลาสที่อยู่ในกลุ่มนี้จะต้องมีการสืบทอดมาจากคลาสแม่คือคลาส java.lang.Exception ซึ่งสืบทอดมาจากคลาส java.lang.Throwable อีกชั้นหนึ่ง



รูป **6.2** ผังโครงสร้างของคลาส **java.lang.Throwable**

การที่เมทธอด Caller ทำการเรียกใช้เมทธอดใดๆที่สามารถ Raise Error หรือโยน Error ออกมา ซึ่งภาษาที่สนับสนุนการทำงานเชิงวัตถุจะบังคับให้เมทธอดที่นำเอาคลาสเหล่านี้ไปใช้จะต้องทำการดักจับข้อผิดพลาดเหล่านี้เมื่อมีการเรียกใช้งานเมทธอดเสียก่อน ไม่เช่นนั้นแล้วโปรแกรมจะไม่สามารถคอมไพล์ได้

6.3 วิธีการดักจับข้อผิดพลาดของโปรแกรม

สำหรับการดักจับข้อผิดพลาดของโปรแกรมที่สามารถโยนข้อผิดพลาด (Throw Error Exception) จะต้องอาศัยวิธีการ try .. catch เป็นการบอกให้แก่โปรแกรมว่าให้ทำการลอง (try) ทำคำสั่งต่อไปนี้ (Action Statement) ดู และหากว่ามีข้อผิดพลาดเกิดขึ้น ให้ทำการดักจับ (catch) และกระโดดไปทำงานในส่วนของ catch statement เพื่อทำการจัดการ (handle) กับข้อผิดพลาดที่โยนออกมาจากเมทธอดที่เรียกใช้งาน เพื่อไม่ให้เกิดผลกระทบต่อโปรแกรมโดยรวม

```
try {
    action statement …
    action statement …

}catch(Throwable exception) {
    Error handling statement …
}
```

จากตัวอย่างนี้จะเห็นว่าในบล็อก try .. catch สามารถมีคำสั่งที่เรียกใช้มากกว่า 1 คำสั่ง ซึ่งการตรวจจับข้อผิดพลาดนั้นสามารถที่จะทำได้ภายใน catch statement เดียว โดยในเบื้องต้นเราจะต้องรู้ว่าเมทธอดที่เรียกใช้จะทำการโยนข้อผิดพลาดใดกลับออกมา เพื่อให้สามารถดักจับได้อย่างถูกต้อง แต่สำหรับกรณีที่มีการเรียกเมทธอดหลายเมทธอด หรือเมทธอดหนึ่งๆสามารถโยนข้อผิดพลาดได้มากกว่าหนึ่ง วิธีการ

ตรวจจับข้อผิดพลาดสามารถทำได้ในหลาย catch statement หรือ multiple catch statement ดังต่อไปนี้

```
try {
    action statement …
    action statement …

} catch (IOException e) {
    Error handling statement …
} catch(FileNotFoundException e) {
    Error handling statement …
}
```

จะเห็นว่าในการตรวจจับข้อผิดพลาด หากกรณีที่เป็นข้อผิดพลาดที่เกิดจาก Input/Output ก็จะถูกตรวจจับโดย IOException หรือกรณีที่เป็นข้อผิดพลาดจากการหาไฟล์ไม่เจอ ก็จะกระโดดไปในส่วนของการตรวจจับใน FileNotFoundException ซึ่งจะทำให้สามารถจัดการกับข้อผิดพลาด (Handle) ได้อย่างเหมาะสม

การดักจับข้อผิดพลาดในลักษณะนี้จำเป็นที่จะต้องรู้ล่วงหน้าว่าเมทธอดที่กำลังเรียกใช้อยู่นั้นจะโยนข้อผิดพลาดใดๆกลับออกมา ซึ่งในเบื้องต้นจะต้องศึกษาโครงสร้างของเมทธอดและคลาสที่เรียกใช้ แต่ในบางครั้งการตรวจจับข้อผิดพลาดเพื่อให้เกิดความยืดหยุ่นเพียงพอในการที่จะตรวจจับข้อผิดพลาดอื่นๆที่นอกเหนือจากเมทธอดที่เรียกใช้กำหนดไว้ เราสามารถกำหนดให้สามารถดักจับข้อผิดพลาดที่นอกเหนือได้โดยใช้ Exception ดังนี้

```
try {
    action statement …
    action statement …
     action statement …
} catch (IOException e) {
    Error handling statement …
} catch(FileNotFoundException e) {
    Error handling statement …
} catch(Exception e) {
    Error handling statement …
}
```

จากโครงสร้างของคลาสในรูป 6.2 จะเห็นได้ว่าโครงสร้างของคลาส java.lang.Throwable นั้นมีคลาส Exception ซึ่งจะทำหน้าที่เป็นคลาสแม่ของทุกคลาสที่ทำหน้าที่ในการดักจับและจัดการข้อผิดพลาด (Exception Classes) ดังนั้นในการดักจับข้อผิดพลาดโดยทั่วไปแล้วเราสามารถดักจับโดยใช้คลาส Exception โดยตรงได้ เนื่องจากอาศัยหลักการของ Polymorphism ที่สามารถ Cascade จากคลาสลูกไปยังคลาสแม่ได้การดักจับข้อผิดพลาดโดยใช้คำสั่ง try..catch นั้นเป็นวิธีการที่ดักจับข้อผิดพลาดในลักษณะที่ให้โปรแกรมลองทำงานบางอย่าง และหากกรณีที่มีข้อผิดพลาดใดๆก็ตาม โปรแกรมก็จะกระโดดไปทำงานในบล็อกของคำสั่ง catch ซึ่งเป็นการ Handle ข้อผิดพลาด โดยปกติจะแสดงข้อความที่อธิบายให้ผู้ใช้ทราบถึงข้อผิดพลาดว่าเกิดอะไรขึ้น เพื่อให้ผู้ใช้สามารถกลับไปแก้ไข ดังตัวอย่างพบข้อผิดพลาดกรณีที่ผู้ใช้ส่งค่า

ตัวอักษรเพื่อที่จะทำการแปลงเป็นตัวเลข ก็จะเป็นการบอกผู้ใช้งานว่าตัวอักษรที่ส่งมานั้นไม่สามารถแปลงเป็นตัวเลขได้ เป็นต้น

```
public int convert(String s)
{
    int result = 0;
    try
    {
        result=Integer.parseInt(s);
    }
        catch (NumberFormatException e)
    {
        System.out.println(e + "Unable to convert to number");
    }
}
```

การทำงานของโปรแกรมที่อยู่ในบล็อก try..catch ส่วนใหญ่มักจะเป็นคำสั่งที่มีมากกว่า 1 คำสั่งทำงานอย่างต่อเนื่องกัน เพื่อทำงานอย่างเป็นลำดับขั้นตอน เช่นเราอาจจะกำหนดให้คำสั่งทำงาน

1. ทำการเปิดไฟล์และอ่านข้อมูลที่อยู่ในไฟล์
2. ทำการเชื่อมต่อกับระบบเครือข่าย
3. ทำการส่งข้อมูลผ่านทางเครือข่าย

...

ซึ่งหากคำสั่งทั้ง 3 คำสั่งที่ทำการเสร็จสิ้นสมบูรณ์แล้วก็จะต้องทำการปิดไฟล์ และปิดการเชื่อมต่อกับระบบเครือข่าย แต่หากกรณีที่มีข้อผิดพลาดเกิดขึ้นจากคำสั่งใดคำสั่งหนึ่ง โปรแกรมก็จะกระโดดไปทำงานที่บล็อกของคำสั่ง catch นั้นๆทันที ซึ่งกรณีเช่นนี้จะทำให้โปรแกรมทำงานไม่ต่อเนื่อง และภายในบล็อก catch ก็จะจัดการเฉพาะคำสั่งที่พบปัญหาเท่านั้น ไม่ได้ทำการจัดการกับคำสั่งหรือการจบการทำงาน (Process Clean-up) อย่างสมบูรณ์ ดังนั้นเราจะจัดการปัญหาในลักษณะนี้โดยอาศัย finally บล็อก

```
try {
      …
 } catch {
      …
 } catch {
      …
 } finally {
      …
 }
```

โดย finally บล็อกมีลักษณะการทำงานคือ finally จะถูกทำงานทุกครั้งที่มีการทำคำสั่งภายใน try..catch โดยไม่ว่าจะเป็นคำสั่ง try หรือ catch ที่ทำงานเสร็จสิ้นแล้ว โปรแกรมก็จะทำงานในส่วนของ finally เป็นบล็อกสุดท้ายเสมอ ซึ่งจะเห็นได้ว่าประโยชน์ของ finally คือจะถูกเรียกใช้งานเสมอไม่ว่าจะเป็นในกรณีที่โปรแกรมทำงานปกติ หรือมีข้อผิดพลาดเกิดขึ้น ดังนั้นจากตัวอย่างปัญหาดังกล่าวข้างต้นเราจึง

สามารถใช้ finally เพื่อจัดการกับคำสั่งที่จำเป็นต้องทำงานในภาวะปกติหรือเมื่อมีข้อผิดพลาดเกิดขึ้น ดังตัวอย่างต่อไปนี้ที่ใช้ finally บล็อกเพื่อทำการปิดไฟล์ ปิดการเชื่อมต่อกับระบบเครือข่าย

```
try {
    Open file to read
            Open network connection
            Send information over the network
} catch {
    Handle error from I/O or reading from file
} catch {
    Handle error from opening network connection
} finally {
    Close file
    Close network connection
}
```

6.4 การจัดการ Exception ในเมทธอดแบบซ้อนทับ (Overriding Method)

เราได้อธิบายหลักการทำงานของเมทธอดแบบซ้อนทับในบทที่ 4 ในกรณีที่มีการสืบทอดจากคลาสแม่ โดยที่ชื่อของเมดธอดในคลาสลูกมี Signature เหมือนกันกับคลาสแม่ เราจะเรียกเมทธอดนี้ว่าเป็น Override Method สำหรับในกรณีเมทธอดที่อยู่ในคลาสแม่มีการโยน Exception ข้อผิดพลาดขึ้นมา เมทธอดที่ถูกสร้างแบบซ้อนทับจะต้องถูกออกแบบให้มีการโยน Exception ข้อผิดพลาดนี้ด้วย ดังตัวอย่างเช่น

```
Superclass
       methodA() throws ExceptionX, ExceptionY
       methodB() throws ExceptionZ
SubClass
       methodA() throws ExceptionX, ExceptionY
       methodB() throws ExceptionZ
```

คลาส Superclass มีเมทธอด A ซึ่งโยน Exception X และ Y ออกมา ดังนั้น เมื่อมีเมทธอด Override method ในคลาสลูกก็จำเป็นต้องมีการโยน Exception X และ Y ด้วยเช่นกัน โดยเงื่อนไขของการทำ Exception สำหรับเมทธอดแบบซ้อนทับมีดังต่อไปนี้

1. เมทธอดแบบซ้อนทับจะต้องโยนข้อผิดพลาดเหมือนกันกับคลาสที่ทำการ Override เช่น ExceptionX, ExceptionY
2. เมทธอดแบบซ้อนทับจะต้องโยนข้อผิดพลาดตัวใดตัวหนึ่ง เหมือนกันกับคลาสที่ทำการ Override เช่น ExceptionX หรือ ExceptionY

6.5 วิธีการ Handle Exception โดยวิธีการเผยแพร่ข้อผิดพลาด

การจัดการกับข้อผิดพลาดที่สร้างขึ้นโดยเมทธอดที่เรียกใช้งานโดยวิธีการ try..catch นั้นเป็นวิธีหนึ่งที่สามารถทำได้ และทำให้ผู้ใช้งานสามารถรู้ถึงข้อผิดพลาดที่เกิดขึ้นในทันทีเมื่อมีการเรียกคำสั่งในเมทธอด แต่

อีกวิธีหนึ่งที่สามารถใช้จัดการกับข้อผิดพลาด Exception โดยวิธีการเผยแพร่ต่อ จากตัวอย่างต่อไปนี้เมื่อมีการอ่านค่าอินพุทจากผู้ใช้งานผ่านทาง Console โดยปกติแล้วจะต้องเรียกภายในบล็อก try..catch

```
try
{
     BufferedReader stdin = new BufferedReader (new InputStreamReader
     (System.in));
} catch (IOException e)
{
     //catch error from BufferedReader
}
```

แต่ในกรณีนี้โปรแกรมจะไม่แสดงข้อผิดพลาดใดๆเกิดขึ้นจากคอมไพเลอร์เนื่องจากเมทธอด getUserInput ทำการเผยแพร่ข้อผิดพลาด IOException ต่อออกไปยังเมดธอดที่เรียก getUserInput อีกต่อหนึ่ง ซึ่งกรณีนี้คือเมทธอด Main ก็จะต้องทำการจัดการกับข้อผิดพลาดโดยใช้ try..catch เมื่อเรียกใช้ getUserInput

```
public String getUserInput () throws IOException
{
     …
     BufferedReader stdin = new BufferedReader (new InputStreamReader
     (System.in));
     …
     String s = stdin.readLine( );
     return s;
}


public static void main(String [] argv)
{
    try {
         getUserInput();
    catch (IOException e) {
         System.out.println("Unable to read an input");
    }
}
```

ประโยชน์ของการทำเผยแพร่ Error Propagation คือการที่เราไม่จำเป็นต้องทำการ Handle ข้อผิดพลาดโดยวิธี try..catch กับทุกคำสั่งที่เราต้องการเรียกใช้ แต่การเผยแพร่ Exception นั้นเป็นการปล่อยให้การดักจับข้อผิดพลาดนั้นเป็นหน้าที่ของเมทธอดต่อๆไปที่จะมาเรียกเมทธอดของเราอีกทีหนึ่ง ซึ่งก็เป็นวิธีที่มีข้อดีคือลดความยุ่งยากในการที่จะต้องมาคอยดักจับข้อผิดพลาด และความซ้ำซ้อนของโปรแกรม แต่ข้อเสียก็คือการดักจับข้อผิดพลาดและการแสดงข้อความให้แก่ผู้ใช้งานอาจจะไม่ชัดเจน และขาดรายละเอียดของข้อผิดพลาดไป เนื่องจากเป็นการมองโปรแกรมที่รวมเอาคำสั่ง (ซึ่งแต่ละคำสั่งอาจจะก่อให้เกิดข้อผิดพลาดได้) ในลักษณะที่เป็นกล่องดำ (Blackbox) ดังตัวอย่างในคลาส Network ต่อไปนี้ ซึ่งเป็นการสร้างการเชื่อมต่อกับระบบเครือข่าย หากมีข้อผิดพลาดใดๆเกิดขึ้นก็จะทำการโยนข้อผิดพลาด Exception กลับไปยัง Caller Methor ซึ่งเป็นเมทธอดที่เรียกใช้ OpenNetworkConnnection ก็จะรู้เพียงแค่ว่ามีข้อผิดพลาดเกิดขึ้นจากการเชื่อมต่อกับเครือข่ายแต่อาจจะไม่รู้ว่าเกิดขึ้นจากสาเหตุใด

```
public class Network {
      public void OpenNetworkConnection() throws Exception
      {
            Socket s = new Socket("SERVER", 50000);

            ObjectOutputStream oos;
            oos = new ObjectOutputStream(s.getOutputStream());
            oos.writeObject(Locale.getDefault());

            InputStreamReader isr;
            isr = new InputStreamReader(s.getInputStream());
            BufferedReader br = new BufferedReader(isr);
            System.out.println(br.readLine());
       }

      public void Connect()
      {
            try
            {
                OpenNetworkConection();
            } catch (Exception e) {
                System.out.println("Unable to connect to SERVER");
            }
      }
}
```

6.6 การสร้าง Exception สำหรับดักจับข้อผิดพลาดของโปรแกรมขึ้นมาใช้งานเอง

จากที่ผ่านมาเราจะพบว่าการดักจับข้อผิดพลาดนั้นจะใช้ข้อผิดพลาดที่อยู่ในกลุ่มของ Exception Class ที่มีลักษณะเป็น Build-in Class และเมทธอดที่สร้างมาให้กับคอมไพล์เลอร์ที่อยู่ในรูปของแพคเกจต่างๆนั้นจะมีการดักจับข้อผิดพลาดและโยน Exception ต่างๆออกมาอยู่แล้ว โดยที่เมื่อเราเรียกใช้งานเมทธอดเหล่านั้น เราอาจจะต้องศึกษาคู่มือจากเอกสารอ้างอิงออนไลน์ของคอมไพล์เลอร์ก่อน เพื่อดูว่าเมทธอดที่เรากำลังเรียกใช้จะโยน Exception อะไรออกมา เพื่อจะได้ทำการ try...catch ได้อย่างถูกต้อง

สำหรับกรณีที่ต้องการสร้างคลาสที่ทำการโยน Exception เองนั้นสามารถทำได้โดยคำสั่ง throw คลาสที่ทำหน้าที่ Exception ที่มีมากับคอมไพล์เลอร์ (Build-in Exception) โดยวิธีการสร้างเป็นอินสแตนซ์ของคลาสขึ้นมาใหม่ ดังตัวอย่างต่อไปนี้

```
if (ErrorOccur)
{
      throw new Exception("message description" +
      localVariable.toString() );
}
```

ซึ่งกรณีนี้เราสามารถเลือกว่าคลาส Exception ไหนมีความเหมาะสมกับข้อผิดพลาดของโปรแกรมที่เราพัฒนาอย่างไร ดังกรณีเช่น

คลาสที่จัดการข้อผิดพลาด	คำอธิบาย
IOException	ไม่สามารถอ่านค่าอินพุทจากไฟล์ได้
FileNotFoundException	ไม่สามารถหาไฟล์ที่ต้องการได้
DataFormatException	ไม่สามารถแปลงเป็นชนิดของข้อมูลที่ต้องการได้
ClassNotFound	ไม่สามารถพบคลาสที่ต้องการได้
SQLException	ไม่สามารถทำงานกับคำสั่ง SQL ที่กำหนดได้
FontFormatException	ไม่สามารถทำงานกับรูปแบบฟอนต์ที่ต้องการได้

หรือ Exception อื่นๆที่เกี่ยวข้องกับข้อผิดพลาดในเรื่องต่างๆ ซึ่งในภาษาจาวา หรือดอทเน็ตของไมโครซอฟท์เองจะมีคลาสที่เกี่ยวข้องกับ Exception ให้เลือกอยู่มากมาย แต่อย่างไรก็ตามในบางครั้งคลาส Exception ที่มีมาให้นั้นอาจจะยังไม่สื่อความหมาย หรือไม่สามารถแสดงข้อความตามที่เราต้องการใช้ได้จริง ซึ่งกรณีแบบนี้เราก็อาจจะจำเป็นต้องมีการสร้างคลาสที่ทำหน้าที่เป็น Error Exception ของเราขึ้นมาเอง ซึ่งวิธีการก็คือ

1. สร้างคลาสของเราเองขึ้นมาใหม่ โดยตั้งชื่อคลาสด้วยชื่อที่สื่อความหมาย และควรลงท้ายด้วยคำว่า ...Exception เช่น MyOwnErrorException และจะต้องให้คลาสนี้ทำการสืบทอดจากคลาสแม่ Exception (ดังที่ได้กล่าวมาข้างต้นแล้วว่าทุกคลาสที่ทำหน้าที่ในการดักจับข้อผิดพลาดจะต้องสืบทอดจากคลาสแม่ Exception)
2. ภายในคลาสใหม่นี้ ให้ทำการ Override คอนสตรักเตอร์ของคลาส Exception โดยการเรียกผ่านคำสั่ง super(...) ซึ่งจะต้องมีอย่างน้อย 2 คอนสตรักเตอร์ ได้แก่

```
super() {  …  }            //default constructor
super(String str) { … }    //constructor with variable
```

ในลักษณะที่เป็นการโอเวอร์โหลดคอนสตรักเตอร์ เพื่อเป็นการกำหนดค่าเริ่มต้นให้แก่คลาส ดังตัวอย่างที่เป็น Error Exception สำหรับการตรวจสอบการทำงานของสแต็ก (Stack)

```
public class FullStackException extends Exception
{
   public FullStackException()
   {
       super();
   }

   public FullStackException (String message)
   {
       super(messsage);
   }
}


public class EmptyStackException extends Exception
{
   public EmptyStackException()
   {
```

```
        super();
    }

    public EmptyStackException (String message)
    {
        super(messsage);
    }
}
```

และเมื่อเราทำการเขียนโปรแกรมโค๊ดเพื่อทำการนำข้อมูลเข้าสู่สแต็ก (push) หรือการนำออกจากสแต็ก (pop) จะเห็นว่ากรณีที่พบข้อผิดพลาดที่อาจจะเกิดขึ้นในกรณีที่สแต็กเต็มหรือว่างเปล่า โปรแกรมจะทำการโยนข้อผิดพลาด (Throw Error) โดยเป็นคลาส Error ที่เราทำการสร้างขึ้นมาเอง

```
public class Stack
{
       public void push(Object o) throw FullStackException
       {
          if (isFull()) {
                throw new FullStackException();
          }
          else {
                // else push the value
         }
       }

       public Object pop() throw EmptyStackException
       {
          if (isEmpty()) {
                throw new EmptyStackException();
          }
          else {
                // else pop the value from stack
         }
       }
}
```

ดังนั้นเมื่อเวลาที่เราเรียกใช้เมทธอด push หรือเมทธอด pop เราจำเป็นต้องเรียกใช้ภายในบล็อก try..catch และทำการดักจับข้อผิดพลาด FullStackException หรือ EmptyStackException ตามที่เมทธอด push หรือ pop ได้มีการโยนออกมา

```
public static void main(String [] argv)
{
       Stack s = new Stack();

       try {
           s.push( obj );
       } catch (FulllStackException e) {};

       try {
           Object obj = s.pop();
       } catch (EmptyStackException e) {};
}
```

บรรณานุกรม

Abraham Silberschatz , Henry F. Korth , S. Sudarshan. *Database System Concepts*, Third Edition, Mc Graw Hill, 1997.

Bertrand Meyer, *Object Success*, Prentice Hall 1995.

Bertrand Meyer, *Object-Oriented Software Construction*, 2nd Edition, Prentice-Hall International Series.

Cay Horstmann, *Computing Concepts with Java Essentials*. 3rd Edition. John Wiley & Sons, Inc., 2003

Dan Osier, Steve Grobman and Steve Baston, *Teach Yourself Delphi 2 in 21 days*, SAMS Publishing, 1996

Dennis Kafura. *Object-Oriented Software Design & Construction with JAVA Web Enhanced.* Prentice Hall, 2000.

Eric Herrmann. *Teach Yourself CGI Programming with Perl in a week*. Sams.Net Publishing, 1996.

Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd Edition, Addison Wiley.

Jamie Jaworski, *Java 2 Certification*, New Riders Publishing, 1999.

Jesse Liberty. *Teach Yourself C++ in 21 Days*. 3rd Edition. Sams Publishing, 1999.

Matt Weisfeld, *The Object-Oriented Thought Process*, 3rd Edition, Pearson Education, 2008

P. J. Plauger, *The Standard C Library*, Prentice Hall, 1992.

S. Monk., J.A. Mariani., B. Elgalal. and H. Campbill. 1996. *Migration from relational to object-oriented database*. Information and software Technology. Vol.38, pp. 467-475.

Syngress Media, Inc. *Sun Certified Programmer for Java 2 Study Guide*, 2nd Edition. Osborne McGraw-Hill, 2001.

W.Meng., A.Kamada. and Y.Chang. 1995. *Transformation of Relational Schemas to Object-Oriented Schemas*: IEEE.

Xavier Pacheco, Steve Teixeira, Delphi 2 Developer's Guide, SAMS Publishing, 1996

ดัชนีศัพท์