การจัดการข้อมูลเชิงเวลาบนฐานข้อมูลเชิงวัตถุ

TEMPORAL DATA MANAGEMENT ON OBJECT-ORIENTED DATABASE

โดย

นายสถาพน พัฒนะคูหา 40010805

นายสรยุทธ กลมกล่อม 40010820

อาจารย์ที่ปรึกษา

อาจารย์ บัณฑิต พัสยา

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์บัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2543

ปริญญานิพนธ์ปีการศึกษา 2543

ภาควิชา วิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การจัดการข้อมูลเชิงเวลาบนฐานข้อมูลเชิงวัตถุ

TEMPORAL DATA MANAGEMENT ON OBJECT-ORIENTED DATABASE

ผู้จัดทำ

1. นายสถาพน พัฒนะคูหา รหัสประจำตัว 40010805
2. นายสรยุทธ กลมกล่อม รหัสประจำตัว 40010820

_____ อาจารย์ที่ปรึกษา

(อาจารย์ บัณฑิต พัสยา)

การจัดการข้อมูลเชิงเวลาบนฐานข้อมูลเชิงวัตถุ

นายสถาพน พัฒนะคูหา 40010805
นายสรยุทธ กลมกล่อม 40010820
อาจารย์ บัณฑิต พัสยา อาจารย์ที่ปรึกษา
ปีการศึกษา 2543

บทคัดย่อ

ตามธรรมชาติของข้อมูลนั้นจะมีการเปลี่ยนแปลงไปตามเวลา แต่ฐานข้อมูลในปัจจุบันสามารถเก็บได้เฉพาะสถานะปัจจุบันของข้อมูลเท่านั้น ทำให้ไม่สามารถเรียกใช้สถานะในอดีตของข้อมูลได้ซึ่งอาจจะทำให้มีปัญหากับการเรียกค้นในภายหลัง หากต้องการให้ข้อมูลที่เก็บมีหลายสถานะ ต้องขยายความสามารถของฐานข้อมูลในปัจจุบันให้รองรับกับข้อมูลเชิงเวลาได้ และด้วยคุณสมบัติของฐานข้อมูลเชิงวัตถุ การจะเพิ่มโครงสร้างข้อมูลและโอเปอร์เรชันที่จัดการกับเวลาทำได้ไม่ยากนัก โดยจะเก็บช่วงเวลาที่แต่ละสถานะของ ข้อมูลเป็นจริง หรือช่วงเวลาที่แต่ละสถานะของข้อมูลถูกเก็บในฐานข้อมูลโดยจะใช้ภาษา OQL (Object Query Language) ของฐานข้อมูลเชิงวัตถุร่วมกับโอเปอร์เรชันที่สร้างไว้ เพื่อจัดการและเรียกค้นข้อมูลเชิงเวลาให้ได้ตามต้องการ

Temporal Data Management on Object-Oriented Database

Sathapon Patanakuha

Sorayut Glomglome

Mr. Bandit Passaya Advisor

ABSTRACT

In natural way, data has changed along the time but current database can store only the current state of data. Hence, we can't call back the former state. It may cause some query problems.

One way to solve this manner is to extend database to support temporal data. With advantages of object-oriented database, adding data structure with temporal semantics is not so hard to implement.

We handle this by keeping the 'valid time' and 'transaction time'. We use OQL (Object Query Language) and some of operations we defined to manipulate and query temporal data.

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้คงไม่อาจเสร็จได้ด้วยดี หากไม่ได้รับความช่วยเหลือ และร่วมมือจากหลาย ๆ ฝ่ายด้วยกัน บุคคลแรกที่ต้องกล่าวถึงเพราะเป็นส่วนสำคัญที่ทำให้วิทยานิพนธ์นี้เสร็จลงได้ก็คือ อาจารย์บัณฑิต พัสยา อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่ให้ความเอาใจใส่ แนะนำ และช่วยเหลือเสมอมา ซึ่งต้องขอขอบพระคุณเป็นอย่างมาก

ขอบคุณ Computer Associates สนับสนุนโปรแกรมจัดการฐานข้อมูลเชิงวัตถุ Jasmine v.1.2 ในงานวิจัยครั้งนี้

สุดท้ายนี้ขอขอบคุณบิดา มารดา อันเป็นที่เคารพรักยิ่ง รวมทั้งเพื่อนๆ พี่ๆ น้องๆ ชุมนุมอิเล็กทรอนิกส์ สำหรับคำแนะนำและความช่วยเหลือตลอดมา

นายสถาพน พัฒนะคูหา
นายสรยุทธ กลมกล่อม

สารบัญ

สารบัญ
(ต่อ)

สารบัญรูปภาพ

สารบัญรูปภาพ
(ต่อ)

สารบัญตาราง

บทที่ 1
บทนำ

วัตถุประสงค์

1. เพื่อศึกษาการจัดการของระบบฐานข้อมูลเชิงวัตถุ
2. เพื่อศึกษา Temporal Database
3. ศึกษาและประยุกต์ใช้ TOM (Temporal Object Model)
4. นำ Temporal Object-Oriented Database ไปใช้งานจริงได้

ในฐานข้อมูลเชิงสัมพันธ์นั้นใช้ External Identifier เป็น key ถ้าไม่ได้รับการออกแบบที่ดีอาจจะทำให้ค่าเปลี่ยนได้ ซึ่งค่า key ไม่ควรจะเปลี่ยนเพราะจะมีปัญหาในเรื่องการจัดการกับฐานข้อมูล แต่ในฐานข้อมูลเชิงวัตถุใช้ OID (Object Identifier) ที่ระบบสร้างให้และจะไม่มีการเปลี่ยนแปลง และถ้าหากนำฐานข้อมูลเชิงวัตถุไปจัดการกับข้อมูลเชิงเวลาผู้พัฒนาโปรแกรมจะต้องจัดการในเรื่องเวลาเอง แต่ในฐานข้อมูลเชิงวัตถุสามารถฝังการจัดการกับเวลาไว้ที่ฐานข้อมูลได้ ทำให้ไม่ต้องจัดการเรื่องเหล่านี้ในโปรแกรมอีก

งานวิจัยชิ้นนี้เป็นการศึกษาเกี่ยวกับทฤษฎีของ Temporal และแนวความคิดของฐานข้อมูลเชิงวัตถุเพื่อพัฒนาให้ฐานข้อมูลเชิงวัตถุนั้นสามารถจัดการกับข้อมูลเชิงเวลาได้ โดยจะศึกษาเกี่ยวกับโมเดลต่างๆ ถึงข้อดีข้อเสียของแต่ละโมเดล แล้วทำการเพิ่มโครงสร้างข้อมูลและโอเปอร์เรชันที่จัดการกับเวลาลงไปในฐานข้อมูลเชิงวัตถุ โดยจะสร้างต้นแบบโปรแกรมสำหรับระบบทะเบียนนักศึกษาขึ้นมาเพื่อทดลองนำฐานข้อมูลที่พัฒนาแล้วทดลองใช้จริง

บทที่ 2
ฐานข้อมูลเชิงเวลา

ฐานข้อมูลเชิงเวลา (Temporal Database) คือ ฐานข้อมูลที่สามารถเก็บและจัดการข้อมูลที่แปรผันตามเวลาได้ ซึ่งโปรแกรมส่วนใหญ่ที่ทำงานกับฐานข้อมูลก็จะต้องเกี่ยวข้องกับเวลาอยู่แล้ว ไม่ว่าจะเป็นงานด้านการเงิน เช่น การบัญชี, การธนาคาร หรืองานด้านการเก็บข้อมูลต่างๆ เช่น การเก็บข้อมูลส่วนตัว, เวชระเบียน, การจัดการสต๊อกสินค้า หรืองานด้านจัดการตารางเวลา เช่น สายการบิน, รถไฟ, ระบบจองห้องพักของโรงแรม หรืองานด้านวิทยาศาสตร์อื่นๆ เช่น การพยากรณ์อากาศ เป็นต้น

2.1 หลักการพื้นฐานของฐานข้อมูลเชิงเวลา

ฐานข้อมูลจะเก็บข้อมูลที่เป็นจริงบางส่วน(เฉพาะที่จำเป็น)เอาไว้ ซึ่งจะเรียกข้อมูลที่เก็บนี้ว่า *mini-world* โดยอาจเก็บไว้ในรูปแบบต่างๆ กัน เรียก *database entities* และ คำว่า *fact* จะหมายความถึง ข้อมูลใดๆ ที่เราสามารถหาค่าได้ว่าเป็นจริงหรือไม่ โดยทั่วไปเวลา (Time) จะสัมพันธ์กับ database entities

เวลาที่เราสนใจในฐานข้อมูลเชิงเวลาอาจแบ่งได้เป็น 3 ประเภทคือ

- **Valid Time** : คือ เวลาที่ fact นั้นเป็นจริง เช่น ในระบบทะเบียนนักศึกษา นักศึกษาชื่อ "สถาพน" ได้เข้าเป็นนักศึกษาตั้งแต่เดือนพฤษภาคม ปี 2540 และได้เปลี่ยนชื่อเป็น "สรยุทธ" ในเดือนพฤษภาคม ปี 2541 ดังนั้นในช่วงเดือนพฤษภาคม ปี 2540 จนถึงเดือนพฤษภาคม ปี 2541 (valid time = [พฤษภาคม 2540-พฤษภาคม 2541)) fact ของชื่อนักศึกษาคนนี้จะเป็น "สถาพน" และตั้งแต่พฤษภาคม 2541 เป็นต้นไป (valid time = [พฤษภาคม 2541 - พฤษภาคม 9999) ชื่อนักศึกษาก็จะเป็น "สรยุทธ"
- **Transaction Time** : คือ เวลาที่ fact นั้นปรากฏในฐานข้อมูล เช่น จากตัวอย่างระบบทะเบียนข้างต้น นักศึกษาเปลี่ยนชื่อเมื่อเดือนพฤษภาคม 2541 แต่บางครั้งด้วยขั้นตอนการทำงาน fact นั้นอาจไม่ได้ใส่ลงฐานข้อมูล ณ เวลานั้น เช่น ผู้ทำหน้าที่ update ข้อมูลอาจนำ fact นั้นไปใส่ในเมื่อเดือนมิถุนายน 2541
- **User-define Time** : คือ เวลาที่ผู้ใช้กำหนดขึ้นมาเอง เช่น วันเกิด

เวลาที่สำคัญที่สุดคือ *valid time* ของ *fact* (อาจมีความหมายได้ทั้งอดีต ปัจจุบัน และอนาคต) ซึ่งจะเก็บช่วงเวลาที่ fact นั้นเป็นจริงใน mini-world และ valid time จะเก็บสถานะที่เปลี่ยนไปตามเวลา (Time-varying States) ในทางทฤษฎีทุกๆ fact จะมี valid time แต่ในบางครั้ง valid time ก็ไม่จำเป็นจะต้องเก็บลงบนฐานข้อมูล อาจเป็นเพราะ ไม่ทราบ valid time หรือไม่มีความจำเป็นต้องใช้ในงานนั้นๆ

transaction time ของ *database fact* คือ เวลาที่ fact นั้นอยู่ในฐานข้อมูลต่างกับ valid time เพราะ transaction time อาจสัมพันธ์กันกับ database entities ใดๆ ก็ได้ไม่เฉพาะ fact เช่น transaction time อาจสัมพันธ์กับออบเจ็กต์ (Object) หรือ value ใดๆ ที่ไม่เป็น fact (เพราะไม่สามารถเป็นจริงหรือเท็จได้โดยอิสระ

(in isolation) ดังนั้นทุกๆ database entity จะมี transaction time แต่อาจเลือกที่จะเก็บ transaction time ลงในฐานข้อมูลหรือไม่ก็ได้

ทั้ง valid time และ transaction time เป็นแนวคิดหลักที่สำคัญของฐานข้อมูลทุกชนิด และมีความจำเป็นที่จะต้องใช้เวลาทั้งสองชนิดนี้ในหลายๆ application และยิ่งไปกว่านั้น transaction time นั้นมีลักษณะพิเศษที่อาจให้ระบบจัดการฐานข้อมูล (Database Management System ; DBMS) จัดการอย่างอัตโนมัติ โดยเฉพาะ transaction time ของ fact ที่เก็บในฐานข้อมูลที่มีทิศทางไปข้างหน้าอย่างเดียว (marches monotonically forward) และผูกพันอยู่กับเวลาเพียง 2 ชนิด คือ เวลาที่สร้างฐานข้อมูลขึ้นมาและเวลาปัจจุบัน (current time) ซึ่งนั่นก็เป็นเหตุผลสำคัญข้อหนึ่งที่ทำให้หัวข้อของงานวิจัยในเรื่องของฐานข้อมูลเชิงเวลาจะเน้นไปที่การปรับปรุงให้สนับสนุน valid time และ transaction time ในลักษณะที่แยกเป็นอิสระออกจากกัน

นอกจาก valid time และ transaction time แล้ว ยังมีเวลาอื่นๆ อีก เช่น user-define time, decision time แต่ความต้องการที่จะใช้เวลาเหล่านั้นบนฐานข้อมูลเชิงเวลามีจำกัด เพราะว่าจำนวนและความหมายของเวลาเหล่านั้นยังต่างกันตามแต่ application และเวลาเหล่านั้นไม่มีคุณสมบัติพิเศษเช่นที่ transaction time มี

งานวิจัยส่วนมากจะเน้นไปที่ semantic และ representation ของเวลา ตั้งแต่ทฤษฎีอย่างเช่น temporal logic หรือ infinite periodic time sequence ไปจนถึงคำถามในเชิงปฏิบัติ เช่น จะ represent เวลาอย่างไรให้ใช้ space น้อยที่สุด หรือจะใช้ปฏิทินให้เป็นประโยชน์ได้อย่างไร และงานวิจัยอีกส่วนใหญ่ๆ อีกส่วนหนึ่งก็จะพูดถึง time data type เช่น time point, ช่วงเวลา (time interval, period), temporal element (set ของ intervals)

2.2 ทำไมต้องใช้ฐานข้อมูลเชิงเวลา

ในงานบางงานข้อมูลมีการเปลี่ยนแปลงไปตามเวลา และเรามีความจำเป็นที่จะต้องเก็บสถานะ (state) ของข้อมูลที่เปลี่ยนไปตามเวลา ทั้งเพื่อใช้ในการประมวลผลและในการเรียกค้น ซึ่งการจัดการกับเวลาที่เพิ่มเข้ามานี้เป็นเรื่องที่ยุ่งยากมาก เริ่มแรกผู้ใช้จะต้องเพิ่ม valid time, transaction time หรือเวลาแบบอื่นๆ เข้าไปในฐานข้อมูลเอง จัดการกับการเปลี่ยนแปลงสถานะของข้อมูลเอง รวมทั้งต้องเขียน query language ที่ซับซ้อนด้วยเงื่อนไขเพื่อเรียกค้นข้อมูลที่ต้องการด้วยตัวเอง

แนวคิดเรื่องฐานข้อมูลเชิงเวลาจึงได้รับการพัฒนาขึ้นมาเพื่อแก้ไขปัญหานี้ โดยให้ DBMS เป็นตัวจัดการความยุ่งยากเหล่านี้ทั้งหมด เพื่อความสะดวกและรวดเร็วในการนำเอาเวลาและข้อมูลที่ได้ไปใช้ประโยชน์ในขั้นสูงกว่าต่อไป

DBMS ที่เป็นฐานข้อมูลเชิงเวลานั้น จะจัดการเรื่องเวลาและเปลี่ยนแปลงข้อมูลให้ผู้ใช้โดยอัตโนมัติ เช่น จากตัวอย่างในหัวข้อ 2.1 เรื่อง valid time เมื่อนักศึกษาเปลี่ยนชื่อ (update) ก็จะ set Valid Time End (VTE) ของชื่อเก่าให้เป็นวันที่เปลี่ยนชื่อ และ set Valid Time Start (VTS) ของชื่อใหม่ให้เป็นวันที่เปลี่ยนชื่อให้โดยอัตโนมัติ หรือเมื่อผู้ใช้ลบ (delete) ข้อมูลออกจากฐานข้อมูล ระบบก็จะไม่ลบข้อมูลนั้นออกไปจริงๆ แต่จะ set VTE ของข้อมูลตัวนั้นให้เป็นเวลาที่ข้อมูลตัวนั้นถูกลบไป เป็นต้น

ฐานข้อมูลเชิงเวลาจะช่วยให้การเรียกค้นข้อมูลที่ต้องการทำได้ง่ายขึ้นด้วยการเพิ่ม keyword เกี่ยวกับ temporal เข้าไป เช่น VALID เป็นต้น

2.3 โมเดลของเวลาในระบบจัดการฐานข้อมูลเชิงเวลา (T-DBMS)

1. **Snapshot DBMS** เช่น DBMS แบบเดิมๆ ที่ไม่ได้สนับสนุน temporal เก็บข้อมูลเพียงสถานะเดียว
2. **Valid Time (Historical) DBMS** ได้แก่ DBMS ที่สนับสนุนเฉพาะ valid time เท่านั้น
3. **Transaction Time (Rollback) DBMS** ได้แก่ DBMS ที่สนับสนุนเฉพาะ transaction time
4. **Bitemporal DBMS** ได้แก่ DBMS ที่สนับสนุนทั้ง valid time และ transaction time

บทที่ 3
ฐานข้อมูลเชิงวัตถุ

3.1 ฐานข้อมูลเชิงวัตถุ

3.1.1 แนวความคิดเชิงวัตถุ

3.1.1.1 ออบเจ็กต์ (Object)

ออบเจ็กต์คือ Abstract Representation ของ Real World Entity ที่มี

- Unique Identity : ออบเจ็กต์ต้องมีสิ่งที่ระบุถึงความแตกต่างจากออบเจ็กต์อื่น
- Embedded Properties
- ความสามารถในการติดต่อสื่อสารกับออบเจ็กต์อื่นและตัวเอง

3.1.1.2 OID

Object Identity หรือ Object Identifier (OID) เป็นสิ่งที่ใช้ระบุหรืออ้างอิงออบเจ็กต์ ซึ่งแต่ละออบเจ็กต์จะมี OID ไม่ซ้ำกัน OID จะถูกกำหนดให้โดยระบบตั้งแต่เมื่อออบเจ็กต์ถูกสร้างขึ้นและจะไม่สามารถเปลี่ยนได้

OID จะแตกต่างจาก Primary key (PK) ของระบบฐานข้อมูลเชิงสัมพันธ์ เพราะ PK จะขึ้นอยู่กับค่าของแอททริบิวท์ (Attribute) ที่ผู้ใช้กำหนด (user-given value) ซึ่งสามารถเปลี่ยนแปลงในภายหลังได้ แต่ OID ถูกกำหนดโดยระบบซึ่งจะไม่ขึ้นกับค่าแอททริบิวท์ของออบเจ็กต์และเปลี่ยนแปลงไม่ได้ เมื่อออบเจ็กต์ถูกลบออกไปจากระบบ OID ของออบเจ็กต์นั้นก็จะถูกลบตามไปด้วยและจะไม่มีการนำ OID ที่ถูกลบไปแล้วกลับมาใช้ใหม่ ค่าของ OID จะไม่ผูกติดกับตำแหน่งทางกายภาพ (Physical Address) ของหน่วยความจำถาวร (Permanent Memory) ทำให้ระบบเชิงวัตถุมีความเป็นอิสระของข้อมูลทางกายภาพ (Physical Data Independence)

3.1.1.3 แอททริบิวท์ (Attributes หรือ Instance Variables)

แอททริบิวท์ (Attribute) หรือ Instance Variable ก็คือ ข้อมูลของออบเจ็กต์ อาจเป็นได้ทั้งข้อมูลชนิดพื้นฐานหรือเป็นออบเจ็กต์ก็ได้

3.1.1.4 สถานะของออบเจ็กต์ (Object State)

สถานะของออบเจ็กต์ขึ้นกับค่าของแอททริบิวท์ของออบเจ็กต์ ณ เวลาที่กำหนด ถึงแม้ว่าสถานะของออบเจ็กต์จะเปลี่ยนแปลงไปแต่ OID ยังคงเดิม ถ้าต้องการเปลี่ยนสถานะของออบเจ็กต์ต้องเปลี่ยนค่าของแอททริบิวท์โดยจะต้องส่งเมสเซจไปยังออบเจ็กต์ แล้วเมสเซจที่ส่งไปนี้จะเรียกเมธอดที่เกี่ยวข้องให้ทำงาน

3.1.1.5 เมสเซจและเมธอด (Message and Method)

ตามคุณสมบัติของเอนแคปซูเลชั่นการที่จะกระทำการใดๆ กับออบเจ็กต์ต้องกระทำผ่านเมธอดเท่านั้น เมธอดจะถูกใช้เพื่อเรียกดูหรือเปลี่ยนค่าแอททริบิวต์ของออบเจ็กต์ ในการเรียกใช้เมธอดนั้นจะต้องส่งเมสเซจไปยังออบเจ็กต์ เมสเซจที่ส่งจะต้องระบุออบเจ็กต์ที่จะรับเมสเซจ ชื่อเมธอด และพารามิเตอร์ที่เกี่ยวข้อง

3.1.1.6 เอนแคปซูเลชัน (Encapsulation)

ออบเจ็กต์จะเป็นการรวมเอาแอททริบิวท์และเมธอดไว้ด้วยกัน การจะเข้าถึงแอททริบิวท์ต้องทำผ่านเมธอดเท่านั้น ซึ่งจะช่วยปกป้องข้อมูลของออบเจ็กต์

3.1.1.7 คลาส (Class)

ในระบบเชิงวัตถุจะนำออบเจ็กต์ที่มีคุณสมบัติเหมือนกันเข้าไว้ด้วยกันเป็นคลาส หรืออาจพูดได้อีกอย่างว่า คลาสก็คือ Collection ของออบเจ็กต์ที่มีคุณสมบัติเหมือนกันและใช้แอททริบิวท์และเมธอดร่วมกัน

3.1.1.8 การสืบทอดคุณสมบัติ (Inheritance)

การสืบทอดคุณสมบัติเป็นการสร้างคลาสใหม่ (Subclass) โดยสืบทอดคุณสมบัติ (ทั้งแอททริบิวท์และเมธอด) จากคลาสที่มีอยู่แล้ว (Superclass) นอกจากนี้ยังสามารถเพิ่มรายละเอียดให้ Subclass ได้อีกด้วย การสืบทอดคุณสมบัตินี้มีประโยชน์ในเรื่องการนำกลับมาใช้ใหม่

Single Inheritance เป็นการสืบทอดคุณสมบัติมาจากคลาสเดียว

Multiple Inheritance เป็นการสืบทอดคุณสมบัติจากหลายๆ คลาส

3.1.1.9 Method Overriding and Polymorphism

Subclass สามารถสร้างเมธอดที่ชื่อซ้ำกับเมธอดที่สืบทอดมาจาก Superclass ได้ เพื่อทำให้ Subclass นั้นมีความเฉพาะเจาะจงมากขึ้น เมธอดที่ Subclass สร้างขึ้นนี้จะถูกเรียกใช้แทน (Override) เมธอดของ Superclass

พอลิมอร์ฟิซึม (Polymorphism) เป็นการทำให้ออบเจ็กต์ที่ต่างกันสามารถตอบสนองต่อเมสเซจเดียวกันได้ในหลายๆ วิธีการ พอลิมอร์ฟิซึมเป็นคุณสมบัติที่สำคัญที่สำคัญของระบบเชิงวัตถุเพราะช่วยให้แต่ละออบเจ็กต์มีความเฉพาะเจาะจงมากขึ้น ในระบบเชิงวัตถุคำว่าพอลิมอร์ฟิซึมจะหมายถึง

1. เมธอดสามารถใช้ชื่อเดียวกันได้ในหลายๆ คลาส
2. ผู้ใช้ส่งเมสเซจเดียวกันไปยังออบเจ็กต์จากคลาสต่างๆ กันก็ยังคงได้ผลลัพธ์ที่ถูกต้อง

3.1.1.10 Abstract Data Type

Abstract Data Type (ADT) เป็นคุณสมบัติที่ใช้สร้างชนิดของข้อมูลใหม่ขึ้นมา โดยกำหนดโครงสร้างข้อมูลและโอเปอร์เรชันที่ใช้จัดการข้อมูลขึ้นมาจากข้อมูลพื้นฐาน

จะสังเกตได้ว่า Abstract Data Type และคลาสมีความหมายใกล้เคียงกัน แต่ในระบบเชิงวัตถุสองคำนี้มีความหมายต่างกัน โดย Type จะหมายถึงโครงสร้างข้อมูลและเมธอดของคลาส และคลาสหมายถึง Collection ของ Object Instance เมื่อกำหนดคลาสใหม่นี้ขึ้นมาก็เป็นการกำหนด Type ใหม่ด้วย Type ที่กำหนดขึ้นจะถูกใช้เป็นต้นแบบในการสร้างออบเจ็กต์ใหม่ ซึ่งจะถูกจัดการโดยคลาสขณะ run-time

ด้วยคุณสมบัติของ ADT และ Inheritance จะสนับสนุนให้มี Complex Object โดย Complex Object ถูกสร้างขึ้นโดยนำออบเจ็กต์อื่นเข้ามารวมไว้ด้วยในรูปของ Set ของ Complex Relation

3.1.2 คุณลักษณะของ Object-Oriented Data Model (OODM)

OODM อย่างน้อยที่สุดควรมีลักษณะดังนี้

1. ต้องสนับสนุนการทำ Complex Object
2. Extensible : ต้องมีความสามารถในการกำหนด Data Type และ Operation ที่เกี่ยวข้องขึ้นมาใหม่ได้
3. ต้องสนับสนุน Encapsulation : รูปแบบของข้อมูลและการจัดการของเมธอดต้องถูกซ่อนจากภายนอก
4. ต้องมีคุณสมบัติ Inheritance : ออบเจ็กต์ต้องสามารถสืบทอดคุณสมบัติ (ข้อมูลและเมธอด) จากออบเจ็กต์อื่นได้
5. ต้องสนับสนุน Object Identity (OID)

เปรียบเทียบ OO และ ER-Model Component

OODM	ER Model
Type	Entity Definition
Object	Entity
Class	Entity Set
Instance Variable	Attribute
N/A	Primary Key
OID	N/A
Method	N/A
Class Hierarchy	ER Diagram (Database Schema)

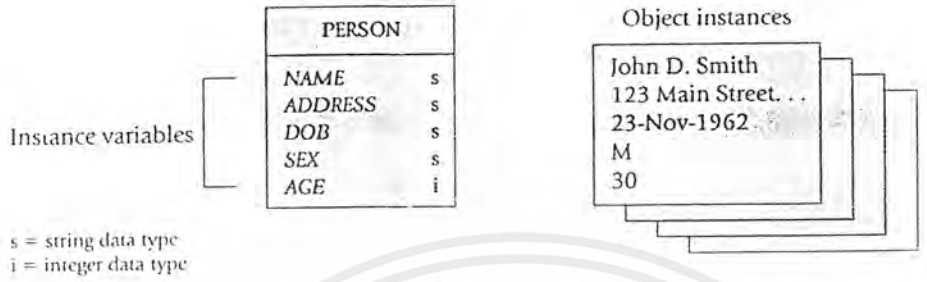
ตารางที่ 3-1 เปรียบเทียบ Object-oriented และ ER-Model Component

ที่มา : *Peter Rob "Database systems : design, implementation, and management"*

3.1.3 The Graphics Representation of Objects : Object Diagram

ใน Object Diagram จะแทนออบเจ็กต์ด้วยรูปสี่เหลี่ยมมุมฉาก มี Instance Variable อยู่ภายใน

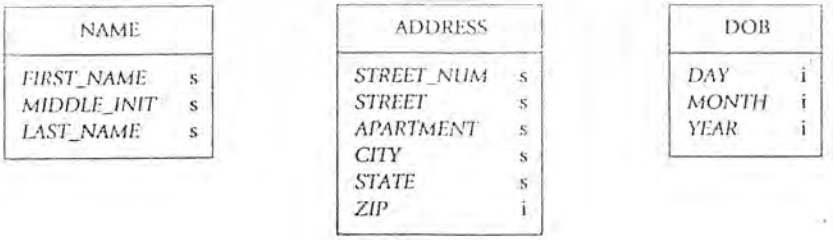
รูปที่ 3.1 แสดงคลาส Person มี Instance Variable คือ NAME, ADDRESS, DOB (Date Of Birth) และ SEX ซึ่งมีชนิดของข้อมูลเป็นสตริง และ AGE มีชนิดข้อมูลเป็นเลขจำนวนเต็ม (Integer)



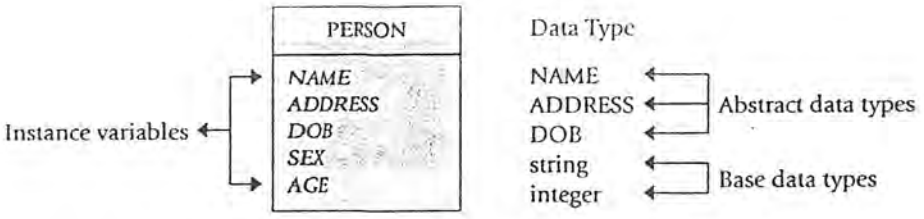
รูปที่ 3.1 คลาส Person

พิจารณาสถานะของออบเจ็กต์ Person ได้ในรูปที่ 3.2 ในระบบเชิงวัตถุสามารถที่จะสร้าง ADT ใหม่ได้จากข้อมูลชนิดพื้นฐานที่มีมาให้ เช่น ถ้าให้ NAME, ADDRESS และ DOB เปลี่ยนเป็น Composite Attribute ซึ่งสามารถทำได้โดยจัดการผ่านคลาสหรือ ADT ดังรูปที่ 3.3 เป็นผลให้คลาส Person มี Attribute ที่ชี้ไปยังออบเจ็กต์ของคลาสหรือ ADT อื่น ดังรูปที่ 3.4 แทนที่จะเป็นข้อมูลชนิดพื้นฐาน

รูปที่ 3.2 State ของออบเจ็กต์จากคลาส Person



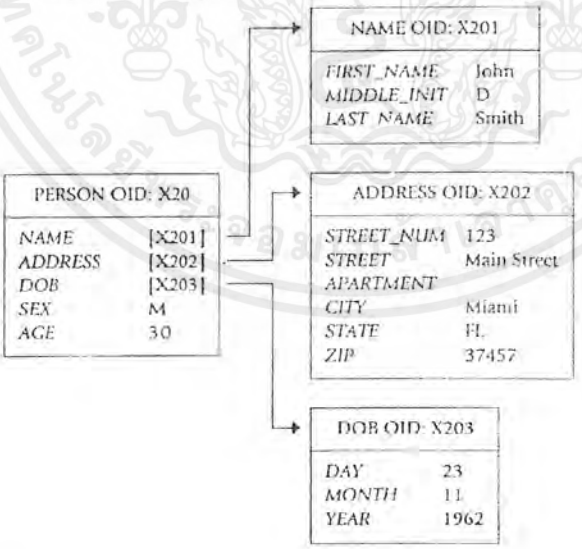
รูปที่ 3.3 การกำหนด Abstract Data Type 3 ชนิด



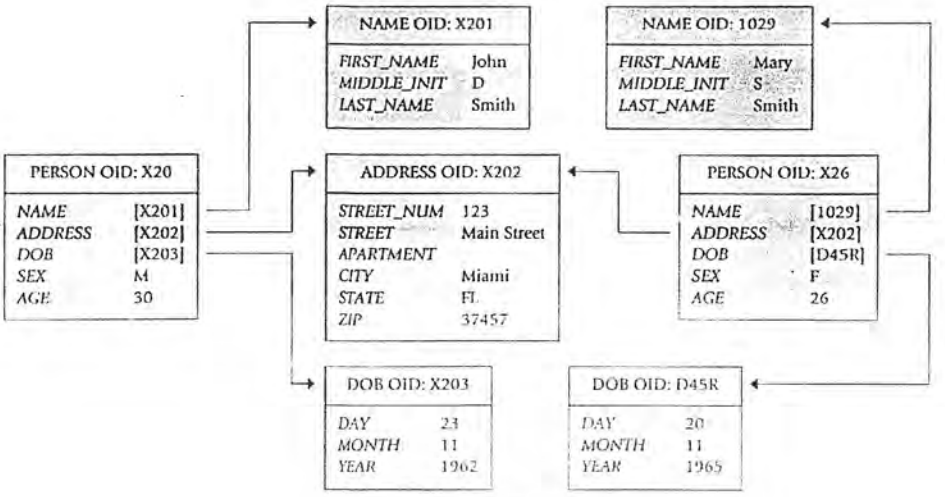
รูปที่ 3.4 แสดงออบเจ็กต์ของคลาส Person กับ ADT

คำว่า Object Space หรือ Object Schema ก็คือ Database Schema ณ เวลาที่กำหนด ใช้ Object Space เพื่อแสดงให้เห็นสถานะต่างๆ ของออบเจ็กต์ ณ เวลาหนึ่ง Object Space ของคลาส Person เป็นดังรูปที่ 3.5 สังเกตได้ว่าใช้ OID ในการอ้างอิงไปยังออบเจ็กต์อื่น โดยแอททริบิวต์ NAME, ADDRESS และ DOB ขณะนี้มีค่าเป็น OID ของออบเจ็กต์จากคลาสหรือ ADT ที่เกี่ยวข้อง แทนที่จะเป็นค่าของข้อมูลชนิดพื้นฐาน การใช้ OID ในการอ้างอิงถึงออบเจ็กต์อื่นทำให้สามารถหลีกเลี่ยงปัญหาความขัดแย้งของข้อมูล (Data Inconsistency) ได้ เพราะ OID ไม่ได้ขึ้นอยู่กับสถานะของออบเจ็กต์ ซึ่งปัญหานี้จะปรากฏอยู่ในระบบฐานข้อมูลเชิงสัมพันธ์เมื่อ Primary Key ถูกเปลี่ยนค่าโดยผู้ใช้

ตัวอย่างเช่น ถ้ามีคนสองคนอาศัยอยู่ในบ้านเดียวกันควรอ้างอิงไปยัง Object Instance ของคลาส ADDRESS ตัวเดียวกัน แทนที่จะเป็น 2 Object Instances ที่มีสถานะเดียวกัน สภาวะเช่นนี้ถูกเรียกว่า Referential Object Sharing ซึ่งการเปลี่ยนแปลงออบเจ็กต์จากคลาส Address ที่ใช้ร่วมกันอยู่ จะส่งผลกระทบต่อออบเจ็กต์จากคลาส Person ทั้งสองออบเจ็กต์ ดังรูปที่ 3.6 ซึ่งมี 4 คลาสประกอบด้วย คลาส Person (2 Instance) คลาส Name (2 Instance) คลาส Address และคลาส DOB (2 Instance)



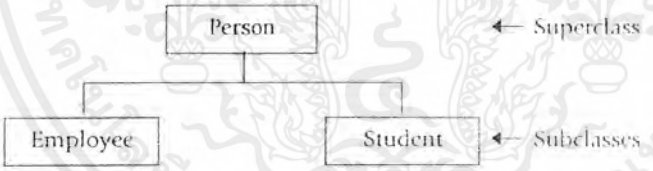
รูปที่ 3.5 สถานะของออบเจ็กต์ที่เป็น Instance ของคลาส Person ที่เป็น ADT



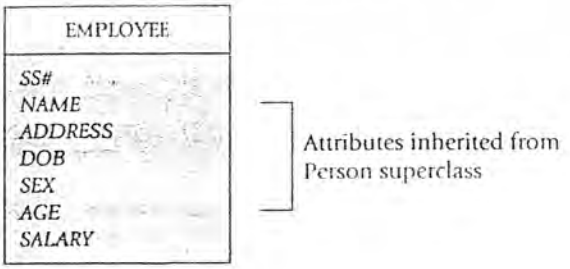
รูปที่ 3.6 Referential Object Sharing

3.1.4 ความสัมพันธ์ระหว่างคลาสและ Subclass (Class-Subclass Relationship)

ในการสืบทอดคุณสมบัติของคลาสมาจาก Superclass จะเรียกความสัมพันธ์แบบนี้ว่า 'is-a relationship' เช่น an employee is a person, a student is a person ตามรูปที่ 3.7 ซึ่งออบเจ็กต์ Employee ในรูปที่ 3.8 ประกอบด้วยแอททริบิวต์ต่างๆ ดังนี้ หมายเลขประกันสังคม (SS#) เก็บเป็นสตริง เงินเดือน (SALARY) เก็บเป็นตัวเลข และมี NAME, ADDRESS, DOB และ AGE ที่สืบทอดมาจาก Supercalss ความสัมพันธ์ระหว่างคลาสและ Subclass นี้จะเป็นแบบ 1:1 ถ้าเป็น Single Inheritance



รูปที่ 3.7 Class Hierarchy

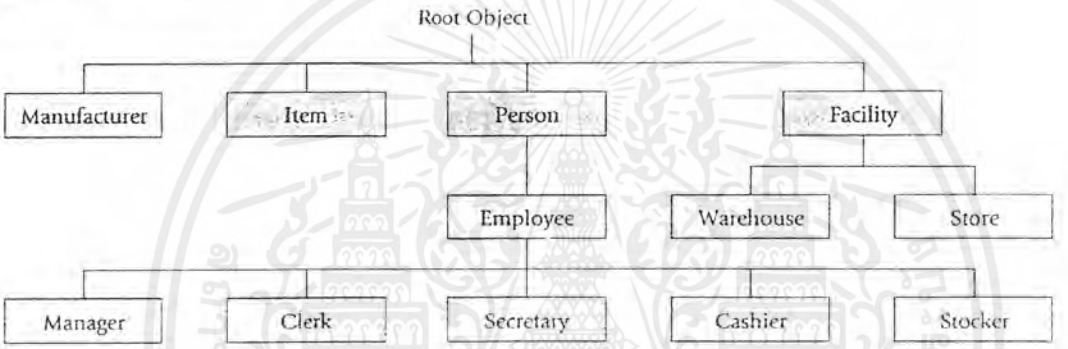


รูปที่ 3.8 แสดงออบเจ็กต์ของคลาส Employee

3.1.5 ความสัมพันธ์ระหว่างคลาส (Interclass Relationships : Attribute-Class Links)

นอกจากความสัมพันธ์ระหว่างคลาสและ Subclass แล้ว OODM ยังสนับสนุนความสัมพันธ์ระหว่างคลาส (Interclass Relationships) ด้วย ความสัมพันธ์ระหว่างคลาสจะเกิดขึ้นเมื่อชนิดข้อมูลของแอททริบิวต์มีการอ้างอิงไปยังชนิดข้อมูลของแอททริบิวต์อื่น ความสัมพันธ์ระหว่างคลาสมีความแตกต่างจากความสัมพันธ์ระหว่างคลาสและ Subclass ในหัวข้อก่อนหน้านี้ ลองพิจารณา Class Hierarchy ของ EDLP (Every Day Low Price) Retail Corporation ดังรูปที่ 3.9

ตามรูปที่ 3.9 ทุกๆ คลาสสืบทอดคุณสมบัติมาจากคลาส Root Object จากรูป Class Hierarchy ประกอบไปด้วยคลาส Manufacturer, Item, Person และ Facility คลาส Facility ประกอบไปด้วย Subclass Warehouse และ Store คลาส Person ประกอบไปด้วย Subclass Employee ที่มี Subclass Manager, Clerk, Secretary, Cashier และ Stoker จากตัวอย่างนี้จะแสดงให้เห็นถึงความสัมพันธ์แบบ 1:M และ M:N และจะ Implement ความสัมพันธ์แบบ M:N

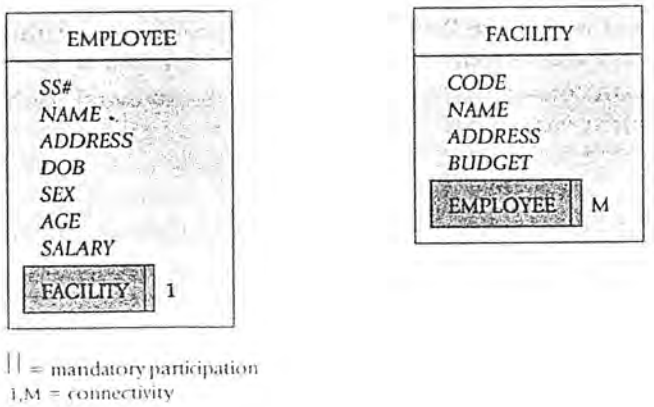


รูปที่ 3.9 Class Hierarchy ของ EDLP Retail Corp.

Representing 1:M Relationship : จากตัวอย่างในรูปที่ 3.9 ระหว่างคลาส Employee และ Facility มีความสัมพันธ์แบบ one to many หรือ 1:M เกิดขึ้น โดย Employee ทุกคนจะทำงานเพียงหนึ่ง Facility ในขณะเดียวกันหนึ่ง Facility จะมีหลายๆ Employee ทำงานอยู่

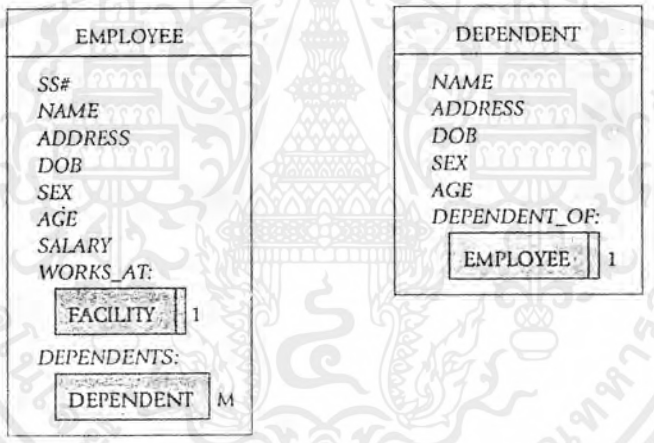
ตามรูปที่ 3.10 แสดงความสัมพันธ์ระหว่าง Employee และ Facility โดยออบเจ็กต์ Facility จะถูกรวมไว้ในออบเจ็กต์ Employee และออบเจ็กต์ Employee ก็ถูกรวมไว้ในออบเจ็กต์ Facility โดยมีรายละเอียดดังนี้

1. คลาสที่เกี่ยวข้องต้องถูกล้อมรอบโดยสี่เหลี่ยมเพื่อที่จะได้เห็นเด่นชัด
2. เส้นคู่ทางด้านขวาของรูปสี่เหลี่ยมแสดงว่าความสัมพันธ์นั้นเป็นแบบ Mandatory (ต้องมีค่าเสมอ)
3. Connectivity จะระบุโดยเขียนกำกับไว้ที่ข้างรูปสี่เหลี่ยมนั้น ในกรณีที่เขียน 1 ถัดจาก Facility ในออบเจ็กต์ Employee หมายถึง Employee ทุกคนจะทำงานเพียงหนึ่ง Facility และ M ที่อยู่ข้าง Employee ในออบเจ็กต์ Facility หมายถึงหนึ่ง Facility จะประกอบไปด้วยหลายๆ Employee



รูปที่ 3.10 แสดงความสัมพันธ์แบบ 1:M

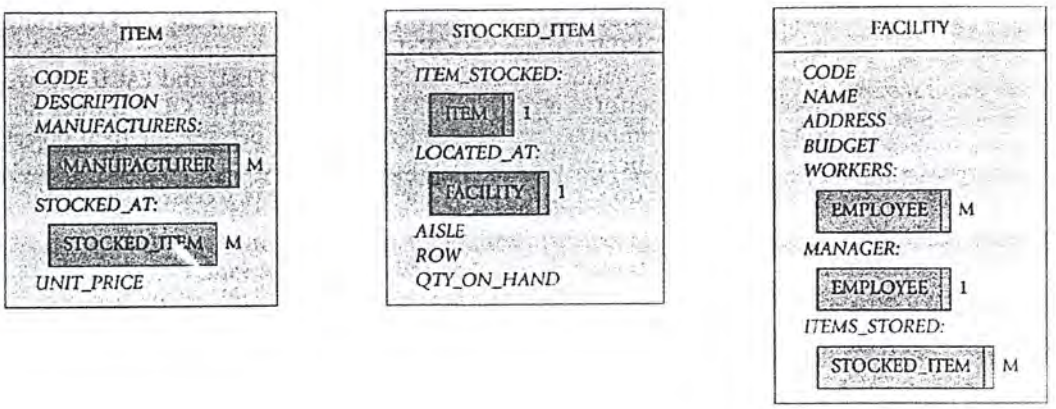
Representing M:N Relationship : ตามตัวอย่าง EDLP Retail Corp. ความสัมพันธ์แบบ many to many (M:N) เกิดขึ้นระหว่างคลาส Manufacturer และ Item ตามรูปที่ 3.11 ซึ่งแสดง Conceptual view โดยที่แต่ละ Item สามารถผลิตได้จากหลายๆ Manufacturer และแต่ละ Manufacturer ก็สามารถผลิตได้หลายๆ Item



รูปที่ 3.11 แสดงความสัมพันธ์แบบ M:N

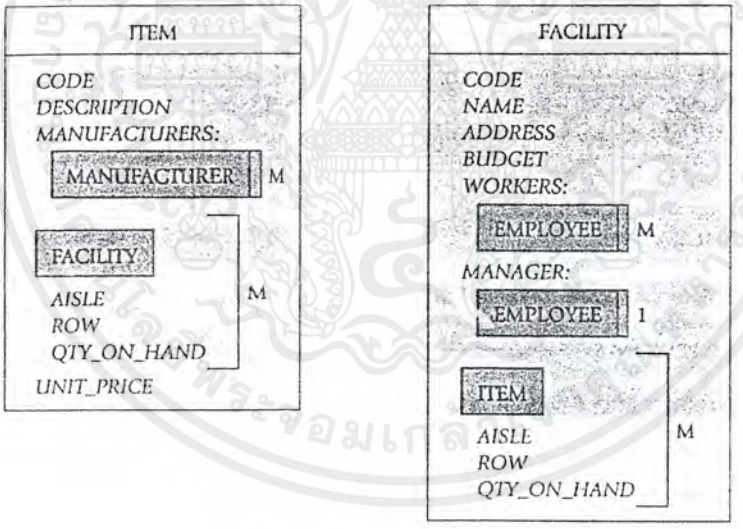
3.1.6 Representing M:N Relationships with an Intersection Class

ถ้าเพิ่มเงื่อนไขให้กับความสัมพันธ์แบบ M:N เพื่อให้สามารถติดตาม (Keep Track) ข้อมูลที่จะเพิ่มเข้าไปได้ เช่น ความสัมพันธ์ระหว่างคลาส Item และ Facility โดยที่แต่ละ Facility จะมีหลาย Item เก็บอยู่ และแต่ละ Item ก็จะเก็บได้หลายๆ Facility ถ้าต้องการรู้เพิ่มว่าแต่ละ Item ในแต่ละ Facility มีปริมาณเท่าไรและเก็บที่ตำแหน่ง (aisle และ row) ไหน โดยแสดงตัวอย่างที่รูป 3.12 วงเล็บใหญ่ ( ]) ที่ใช้ในรูปหมายถึงรวมแอททริบิวต์ทั้งหมดในวงเล็บถือเป็นหนึ่ง Logical Unit เพราะฉะนั้นแต่ละ Item Instance จะประกอบไปด้วยหลายๆ Facility ซึ่งแต่ละ Facility จะประกอบไปด้วยแอททริบิวต์ 3 ตัว ได้แก่ AISLE, ROW และ QTY_ON_HAND ซึ่งในกรณีกลับกันก็เป็นจริงสำหรับทุกๆ Facility

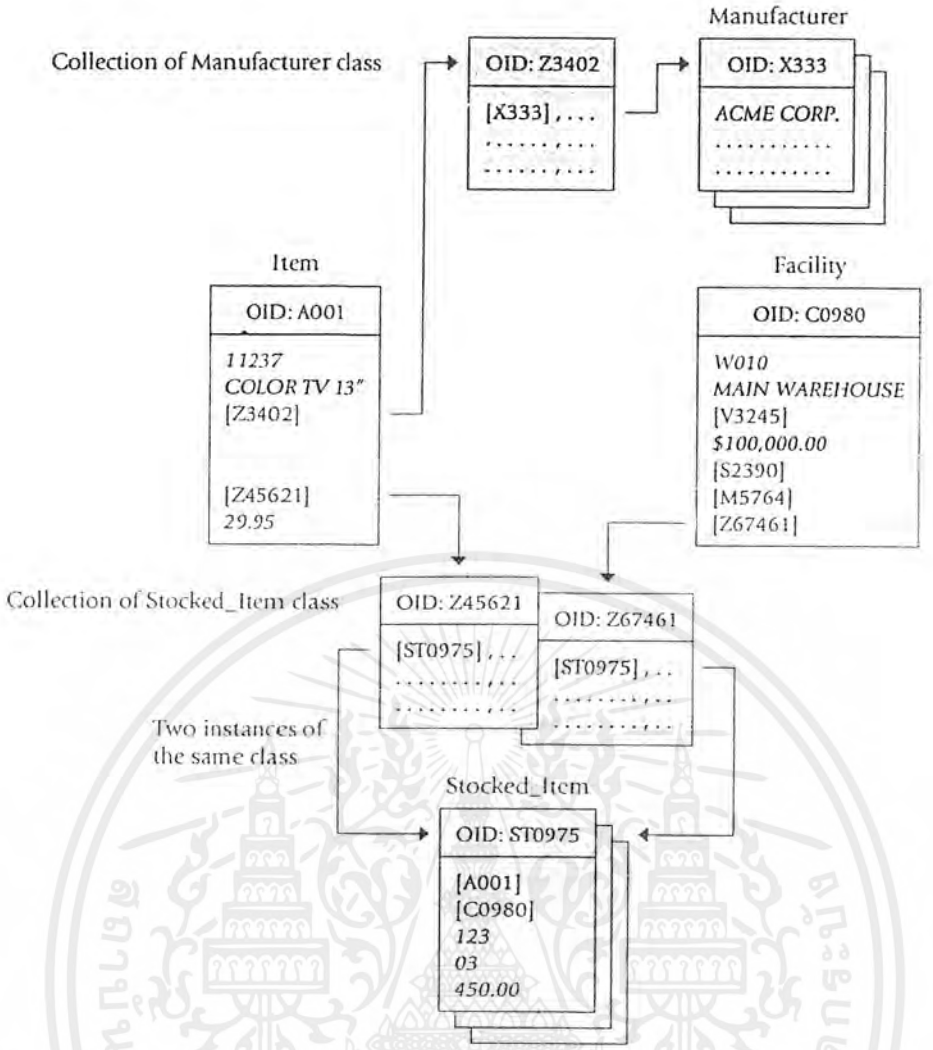


รูปที่ 3.12 แสดงความสัมพันธ์แบบ M:N และ attribute ที่เกี่ยวข้อง

ถ้ามองในมุมมองของฐานข้อมูลเชิงสัมพันธ์ สิ่งที่ควรจะเพิ่มขึ้นสำหรับความสัมพันธ์แบบ M:N นี้ ก็คือ Intersection (bridge) class เพื่อที่จะเชื่อมทั้ง Facility กับ Item และแอททริบิวต์ที่เกี่ยวข้อง โดยสร้างคลาส Stocked_Item ซึ่งประกอบไปด้วย Facility และ Item Instance และแอททริบิวต์ AISLE, ROW และ QTY_ON_HAND ตามรูปที่ 3.13



รูปที่ 3.13 ความสัมพันธ์แบบ M:N กับ Intersection class



รูปที่ 3.14 แสดง Object Space

รูปที่ 3.14 แสดง Object Space ตามแนวคิดข้างต้น อย่างไรก็ดีการออกแบบเช่นนี้ก็เป็นการออกแบบที่ต้องระมัดระวังเป็นพิเศษ โดยมีจุดที่ต้องสนใจคือ

- Stocked_Item Object Instance จะมีการอ้างอิงไปยัง Instance ของคลาสอื่นที่เกี่ยวข้อง ซึ่งการทำ Intersection Class เช่นนี้ก็เพื่อต้องการติดตามข้อมูลที่เพิ่มเข้ามาเท่านั้น
- Item Object Instance ใน Object Schema นี้ประกอบไปด้วย Collection of Stocked_Item Object Instance ซึ่งแต่ละอันก็ประกอบไปด้วย 1 Facility Object Instance ความสัมพันธ์ในทางกลับกันก็ยังคงเป็นจริง
- ใน Object Space การอ้างอิงระหว่างคลาสจะใช้ OID ของออบเจ็กต์ที่ถูกอ้าง
- ค่าที่อยู่ในวงเล็บ [ ] จะแทน OID ของ Object Instance ของบางคลาส เช่น ค่า Z3402 และ Z45621 เป็นต้น เป็น OID ของออบเจ็กต์ที่ถูกอ้างถึง ซึ่งก็คือ Collection ของ Manufacturer และ Stocked_Item ตามลำดับ

ใน Relational Model ตาราง ITEM จะไม่มีข้อมูลที่เกี่ยวข้องกับ MANUFACTURER หรือ STOCKED_ITEM เลย ซึ่งถ้าต้องการใช้ข้อมูลที่เกี่ยวพันกันของทั้ง 3 ตาราง จะต้อง Join ตารางเข้าด้วยกัน แต่

OODM ไม่ต้องการการ Join เพราะออบเจ็กต์ Item จะมีการอ้างอิงไปยังออบเจ็กต์อื่นที่เกี่ยวข้องอยู่แล้ว การอ้างอิงนี้จะนำออบเจ็กต์ Item เข้าสู่ Object Space โดยอัตโนมัติเมื่อถูกเรียกใช้

3.1.7 ระบบจัดการฐานข้อมูลเชิงวัตถุ (Object-Oriented Database Management System ; OODBMS)

OODBMS จะทำหน้าที่คอยจัดการกับฐานข้อมูลเชิงวัตถุ โดยจะใช้คุณสมบัติบางส่วนตามแนวคิดเชิงวัตถุ (OO Concept) และ OODM ที่เป็นเพียงบางส่วน ทั้งนี้เพราะยังไม่มีมาตรฐานที่ใช้กำหนดถึงคุณสมบัติของ OODBMS ที่ต้องสามารถทำได้ ทำให้ OODBMS ที่มีอยู่มีคุณสมบัติแตกต่างกัน อย่างไรก็ดี OODBMS ก็มีคุณสมบัติที่จำเป็นต้องมีอยู่ 13 ข้อด้วยกัน ซึ่งสามารถแบ่งได้เป็น 2 ส่วน ดังนี้

ส่วนที่ทำให้เป็น OO System

1. ระบบต้องสนับสนุน Complex Object
2. สนับสนุน Object Identity (OID)
3. ออบเจ็กต์ต้องถูก Encapsulation
4. ระบบต้องสนับสนุน Type และ Class
5. ระบบต้องสนับสนุนการสืบทอดคุณสมบัติ (Inheritance)
6. ระบบต้องหลีกเลี่ยง Premature binding
7. ระบบต้องเป็น Computational Complete
8. ระบบต้อง Extensible

ส่วนที่ทำให้เป็น DBMS

9. ระบบต้องสามารถจดจำตำแหน่งของข้อมูลได้
10. ต้องสามารถจัดการกับฐานข้อมูลที่มีขนาดใหญ่ได้
11. สนับสนุนการทำงานแบบ Concurrency
12. ต้องสามารถกู้ระบบ (Recovery) จากการทำงานที่ผิดพลาดของ Hardware และ Software ได้
13. ต้องสามารถทำการค้นหาข้อมูลได้ง่าย

3.2 Jasmine's Object Database Query Language (ODQL)

3.2.1 ODQL

คุณสมบัติของ ODQL

- Schema Definition : ผู้ใช้สามารถสร้างคลาสและกำหนดแอททริบิวต์และเมธอดได้ตามต้องการ
- Method definition : กำหนดรายละเอียดภายในเมธอด
- Queries : ค้นหาข้อมูลที่ต้องการ
- Procedural Construct : ODQL เป็นภาษาที่ computational complete และสนับสนุน procedural construct อย่างเช่น loop หรือเงื่อนไขต่างๆ

และสามารถรวมเอาทั้ง query และ procedural construct ไว้ภายใน ODQL program เดียวกันได้

3.2.1.1 Literals, Objects และ Collections ใน ODQL

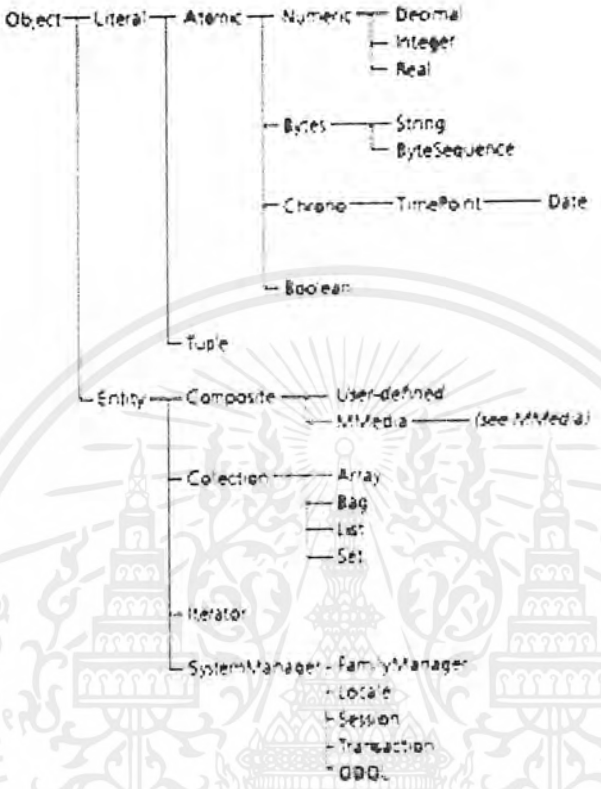
ในระบบฐานข้อมูลแบบเก่า คำว่า Entity Type นั้นจะหมายถึงคลาสและ Entity จะหมายถึงออบเจ็กต์ แต่ใน ODQL model คำว่า Entity และออบเจ็กต์มีความหมายไม่ต่างกันนัก ODQL สนับสนุน Complex Object ผ่าน Entity (ออบเจ็กต์) และ Literal (Value)

- **Class** : ทุกๆ ออบเจ็กต์จะเป็น Instance ของคลาส ในคลาสหนึ่งๆ จะประกอบไปด้วยแอททริบิวต์และเมธอด Value ของแอททริบิวต์อาจเป็นการอ้างอิงไปยังออบเจ็กต์อื่น หรือ Atomic Literal หรือ Collection อย่างเช่น Set ก็ได้ และคลาสก็สามารถมีแอททริบิวต์และเมธอดเป็นของตัวเองได้ เรียก Class Attribute และ Class Method ซึ่งจะต่างจาก Instance Attribute และ Instance Method
- **Object identity** : แต่ละ Instance ของคลาสจะมี Object Identity แต่ Literal จะไม่มี และสามารถใช้ออบเจ็กต์ร่วมกันได้ด้วยการทำ Referential Share
- **Literal** : คือ Atomic Value หรือ Structured atomic value อย่างเช่น set of integer โดย atomic literal นั้นอาจเป็น integer, date, decimal, real, Boolean, bytesequence รูปแบบของ decimal คือ [precision, scale] precision หมายถึง จำนวนหลักของเลขหน้าจุดทศนิยม มีค่าตั้งแต่ 1-18 scale หมายถึง จำนวนหลักหลังจุดทศนิยม string และ bytesequence มีขนาดไม่เกิน 64 k
- **Collection** : ODQL สนับสนุน collection หลายๆรูปแบบ ไม่ว่าจะเป็น set, bag, list, array และสามารถกำหนดเป็น collection ของ literal หรือ object ก็ได้ แต่ไม่สามารถ referential share ได้ set คือ collection ที่แต่ละค่าใน collection ไม่ซ้ำกันและไม่เรียงลำดับ bag จะเหมือน set แต่ค่าใน collection ซ้ำได้ list คือ collection ของข้อมูลที่มีลำดับ และเราสามารถจะ retrieve ค่าของสมาชิกตัวแรก ตัวสุดท้าย หรือ ณ ตำแหน่งที่ต้องการได้ array คือ collection mมีจำนวนสมาชิกแน่นอน และสามารถ access ค่าได้จากตำแหน่งของ element นั้น
- **Tuple** : รูปแบบของ tuple คือ $[a_1:v_1, a_2:v_2, \ldots, a_n:v_n]$ attribute $a_1, a_2, \ldots, a_n$ จะหมายถึง attribute name ส่วน value $v_1, v_2, \ldots, v_n$ จะเป็น atomic literal value (อย่างเช่น string, integer) หรือ object หรือ collection of literal value หรือ collection of object ก็ได้ tuple จะถูกใช้เพื่อวัตถุประสงค์เดียวเท่านั้น นั่นคือ เพื่อกำหนด data structure ซึ่งจะกำหนดไว้ใน ODQL expression และจะไม่ปรากฏใน persistent Jasmine database
- **Relationship** : แอททริบิวต์ของออบเจ็กต์สามารถอ้างอิงไปยังออบเจ็กต์อื่นๆ ได้ ODQL สนับสนุนทั้ง single-value (single instance ของ object) และ collection-value (set of object)

3.2.1.2 ODQL Complex Object Terminology

- **Characteristic** : Jasmine ใช้คำนี้ในความหมายถึง state และ behavior ของ class ซึ่ง behavior ก็คือ เมธอดของคลาสนั่นเอง ส่วน state นั้นจะเก็บอยู่ใน property ของคลาส
- **Property** : state ของ class จะเก็บอยู่ใน property และเราสามารถจะอ่านหรือแก้ไขค่านั้นได้ property มี 2 ประเภทคือ attribute และ relationship

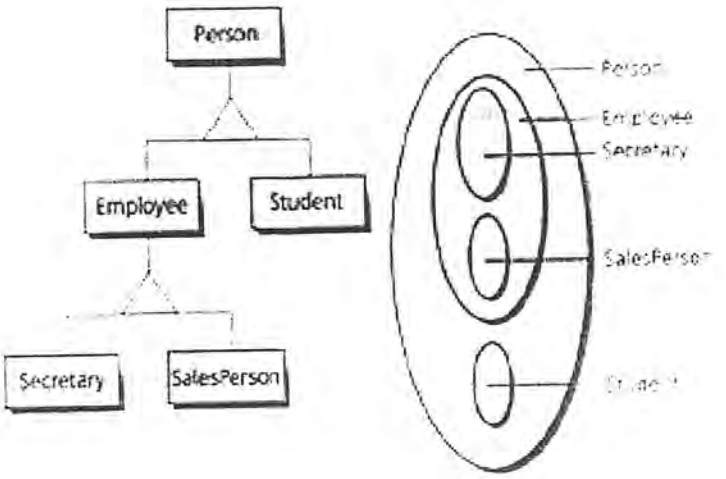
- **Attribute** : อาจเป็นได้ทั้ง literal และ single-value หรือ collection of literal (set, bag, list, array) และ multivalue attribute value จะไม่มี identity (ไม่สามารถ referential share ได้)
- **Relationships** : สามารถเป็นได้ทั้ง single-value object attribute และ multivalue collection of object ตัว collection ไม่ใช่ object แต่สามารถเก็บ object ที่ referential share ได้



รูปที่ 3.15 Jasmine system class

3.2.1.3 Inheritance ใน ODQL

ODQL สนับสนุนทั้ง single และ multiple inheritance class ที่ผู้ใช้กำหนดขึ้นจะสืบทอดมาจากคลาสอื่นหรือเป็น root class ก็ได้ แต่ root class ก็จะเป็น subclass ของ Composite system class โดยอัตโนมัติ

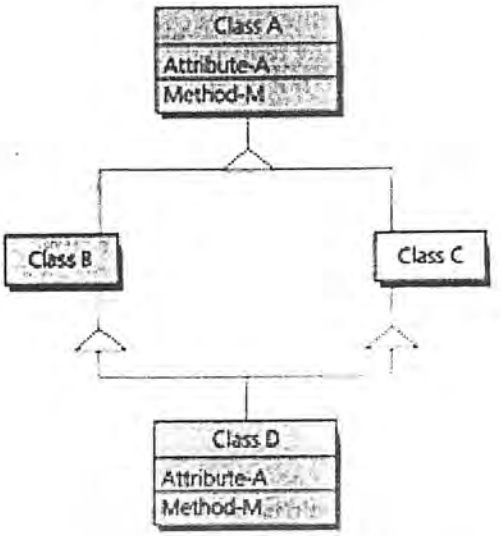


รูปที่ 3.16 inheritance hierarchy และ inheritance set

จากรูปที่ 3.16 เมื่อมีการ selection หรือ query บน superclass การค้นหาจะไม่ได้ทำบน superclass เท่านั้น แต่จะหาใน subclass ด้วย เช่น ถ้าเราต้องการทราบว่ามี Employee คนใดบ้างที่ได้เงินมากกว่า 20,000 ต่อปี การค้นหาก็จะไปหาทั้งใน Secretary, SalesPerson และทุกๆ subclass ของ Employee ซึ่งการ insert, update, delete ก็จะเป็นเช่นเดียวกัน

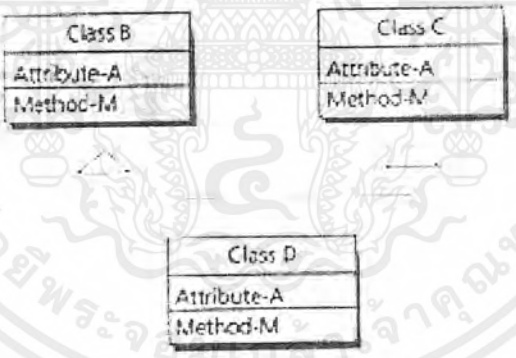
ถ้า class B inherit จาก class A ทำให้ B เป็น subclass ของ A และ A เป็น superclass ของ B Jasmine ใช้คำ subordinate และ superior อธิบายความสัมพันธ์แบบ transitive ของ inheritance hierarchy คลาสใดๆ จะเป็น subordinate ของ class A ก็ต่อเมื่อเมื่อ class นั้นเป็น subclass ของ class A หรือเป็น subordinate ของ subclass ของ A และ superior ก็จะเป็นไปในทางกลับกัน จากรูปที่ 3.16 class Employees เป็น immediate subclass ของ class Person และ class Secretary เป็น subordinate class ของ class Person เช่นเดียวกัน class Person เป็น immediate superclass ของ class Employee และเป็น superior class ของ class Secretary

ส่วนที่ซับซ้อนที่สุดของการสนับสนุน multiple inheritance คือ กฎการ override method หรือ attribute ของ subclass คือ ถ้า method หรือ property เดียวกัน (หรือ method/property ที่มีชื่อเดียวกัน) พบใน class มากกว่า 1 class ไม่ว่าจะเป็นการกำหนดโดยตรงหรือผ่านทางการ inheritance ก็ตาม ก็จะไม่สามารถสร้าง subclass ที่ inherit จาก class เหล่านั้นได้ ยกตัวอย่างเช่น class A มีproperty ชื่อ Po ชนิดของตัวแปรเป็นstring และ class B มี property ชื่อ Po เช่นเดียวกันแต่เป็นตัวแปรชนิด integer Jasmine จะไม่อนุญาตให้สร้าง subclass C ซึ่ง inherit มาจากทั้ง class A และ B



รูปที่ 3.17 multiple inheritance ที่มี superior class ร่วมกัน

จากรูปที่ 3.17 superclass มี property และ method ที่ถูกกำหนดไว้ร่วมกันใน superior class ถ้า superclass ไม่ได้มีการแก้ไข property หรือ method ที่รับมาจาก superior class เดียวกันก็อาจจะเป็นไปได้ที่จะสร้าง class ที่ inherit มาจาก superclass ทั้งสอง class นั้น และถ้า superclass ได้กำหนด property หรือ method ของตัวเองเพิ่มขึ้นมา subclass ก็ต้อง override property หรือ method เหล่านั้น



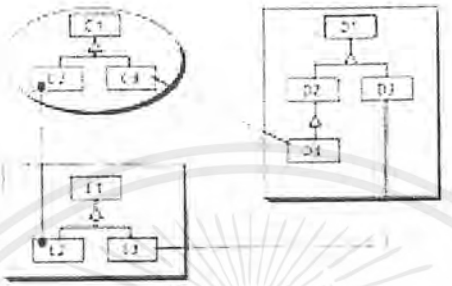
รูปที่ 3.18 multiple inheritance โดยไม่มี superior class ร่วมกัน

จากรูปที่ 3.18 property หรือ method ไม่ได้มาจาก superior class เดียวกัน ถ้า superclass มากกว่า 1 class มี property หรือ method ซ้ำกัน subclass ก็ต้อง override property หรือ method เหล่านั้น แต่ถ้า property หรือ method ไม่ซ้ำกันในระหว่าง superclass subclass จะ override หรือไม่ก็ได้

3.2.2 ODQL Syntax and Semantics

การกำหนด class ใน ODQL

ใน ODQL คลาสจะถูกจัดให้อยู่ใน Class Family (class family คือ logical organization ของ class) อย่างเช่น class ที่เกี่ยวกับ user และ group อาจจัดอยู่ใน class family เดียวกัน class family สามารถอ้างอิง object ใน class family อื่นได้ class สามารถ inherit กันได้จาก class ภายใน class family เดียวกัน แต่สามารถมี attribute (property) ที่อ้างอิง object ใน class family อื่น ดังในรูปที่ 3.19



รูปที่ 3.19 class family

เมื่อเรากำหนด class เราจำเป็นต้องกำหนด class family ด้วย คลาสหนึ่งๆ จะอยู่ใน class family ได้เพียง class เดียวเท่านั้น ชื่อของ class จะต้องห้ามซ้ำภายใน class family หนึ่งๆ แต่เราสามารถมี class ชื่อเดียวกันใน class family ที่ต่างกันได้

Persistent Object ใน ODQL จะต้องเป็น Instance ของ class และในการกำหนด class นั้นจำเป็นต้องมี

- **ชื่อ class**
- **superclass** : จะมีหรือไม่ก็ได้ ทุกๆ user-defined class จะเป็น subclass หรือ subordinate class ของ system class Composite โดยอัตโนมัติ superclass ที่กำหนดใน class definition จะเป็น immediate superclass ของ class ที่กำหนด เช่นเดียวกับ object-oriented language ทั่วๆไป
- **characteristic (method และ property) ของ class** : Jasmine สนับสนุน characteristic หลายๆรูปแบบ ดังนี้

  - *Instance characteristic* : เป็น characteristic ที่สำคัญที่สุด ได้แก่ instance-level property และ method
  - *Class characteristic* : ได้แก่ class-level property และ method โดย class-level property จะถูก share โดยทุกๆ instance ของ class นั้น เช่น class User มี attribute average salary ซึ่งเก็บเงินเดือนเฉลี่ยของพนักงานทุกคนใน class นั้น
  - *Method on collection* : ODQL สนับสนุน method ที่จะทำงานกับ collection ของ object หรือของ class อย่างเช่น method ที่ทำหน้าที่นับจำนวนของ instance ภายใน collection หรือ method ที่ return instance ตัวแรกและตัวสุดท้ายใน list และ method ที่

ทำงานกับ collection of class นั้น class ใน collection ควรจะเป็น class นั้น หรือเป็น subordinate ของ class นั้น

เราอาจกำหนดค่าเริ่มต้นให้กับ class-level หรือ instance-level property ได้ และสำหรับ instance-level property นั้นเราสามารถกำหนด constraint อื่นๆ เช่น กำหนดให้ property นั้นเป็นแบบ "mandatory" คือ จะห้าม property ตัวนั้นมีค่าเป็น null เป็นต้น และสำหรับ property ที่ไม่ใช่ collection เราอาจกำหนดให้ property ตัวนั้นต้อง "unique" ซึ่งก็เหมือนกับ uniqueness constraint ใน relational DBMS ทั่วๆ ไป คือค่าของ property ตัวนี้จะต้อง unique ใน class นั้นๆ ซึ่ง Jasmine system จะคอยตรวจสอบทุกๆครั้งที่มีการสร้าง object ใหม่ขึ้นมาและทุกครั้งที่ค่าของ property นี้ถูกแก้ไข

การกำหนด method สามารถทำได้โดย กำหนด return type ของ method นั้น ชื่อ method และ parameter ที่ต้องการ

รูปแบบทั่วๆไปของ class definition คือ

defineClass class-identifier
[**super** : class-identifier [{,class-identifier}…]]
[**description** : string-constant]
{ [**maxInstanceSize** : size;]
characteristic } ;

ตัวอย่างของการกำหนด class

```
defineClass      User
super :  applicationObject
description : "the class implementing users"
{
  class :
   User createObject(Session Session, String lastName
                     String firstName, String mi,
                     String loginName, String password);
/* This method will retrieve all users except the
** user that have the status of "Out of Service".
*/
    User getUserWithName(Session Session, String name);
  Instance:
```

```
    String lastName         default: "";
    String firstName        default: "";
    String middleInitial    default: "";
    Department              userDepartment;
    Set<Manager>                    managers;
    Manager                 mainManager;
    Integer salary;
    String rank             default: "MTS";
    BasicAddress            address;
    Company                 userCompany;
    Date                            dataHired;
    Real hourRate                   default: 0;

    Void setUserName(Session Session,
                    String lastname, String firstName,
                    String middleInitial);
/* The last, first amd middle name through these accessors
** are obtained and stored in the parameters
*/
    String getLastName(Session Session);
    String getFirstName(Session Session);
    String getMiddleInitial(Session Session);
    String getFullName(Session Session);
    Void setHomeAddress(Session Session, BasicAddress newAddress);
    BasicAddress getHomeAddress(Session Session);
    Void setHourRate(Session Session);
    Real getHourRate(Session Session);

/* Other method */
    Set<Manager> getManagers();
    Real evaluationBonus();
    Manager getMainManager();
...
};
```

3.2.3 ODQL Programming Construct

โดยทั่วไปแล้วมาตรฐานของ database programming ก็จะประกอบไปด้วย data definition sublanguage ที่ใช้ในการกำหนด persistent database object, Querying ใช้เพื่อเรียกค้น object จาก database, data manipulation construct เพื่อ modify, insert, delete object ซึ่งในฐานข้อมูลเชิงสัมพันธ์มาตรฐานสำหรับ data definition และ data manipulation (ทั้งการ query, insert, update, delete data) คือ SQL Data Definition Language (DDL) และ SQL Data Manipulation Language (DML) ซึ่งตาม object-oriented ODMG Standard จะเรียก Object Definition Language (ODL) และ Object Query Language (OQL) ตามลำดับ

ODMG standard (version 2.0) นั้นไม่ได้ให้ความสนใจกับปัญหาใหญ่ข้อหนึ่งของ object-oriented database มากนัก ปัญหานั้นคือ ความไม่เข้ากันระหว่าง programming language และ database sublanguage (the impedance mismatch between a programming language and a database sublanguage) มีการพูดถึงปัญหาข้อนี้มาตั้งแต่กลางทศวรรษที่ 80 เนื่องจากความไม่เข้ากันของ host programming language อย่างเช่น C กับ database sublanguage อย่าง SQL เนื่องจาก SQL เป็นภาษาที่ไม่ computational complete ผู้พัฒนาระบบจึงต้องยุ่งเกี่ยวกับภาษาทั้ง 2 ภาษา ยังไม่นับถึงปัญหาเรื่องการ matching value, ชนิดของตัวแปร และอื่นๆอีกมากมาย

แต่มาตรฐานของ ODMG ก็ยังไม่ได้สนใจที่จะแก้ปัญหาข้อนี้ กลับไปเพิ่ม CORBA's Interface Definition Language (IDL) (ซึ่งเรียกว่า ODL) และ query sublanguage (เรียก OQL) เข้ามา ซึ่งทำให้สามารถติดต่อ (binding) กับ object-oriented language อื่นๆได้เป็นจำนวนมาก และมีข้อกำหนดสำหรับ C++ binding, Smalltalk binding หรือล่าสุด Java Binding ซึ่งทำให้สามารถมี method ที่ส่ง OQL statement เป็น argument ไปประมวลผลยัง object-oriented database engine ที่อยู่ด้านล่างได้

และสำหรับ Jasmine ODQL นั้น นอกจากจะสามารถฝัง ODQL ไว้กับ C/C++ ได้แล้ว ตัว ODQL เองก็เป็นภาษาที่ computational complete คือ มี control structure, ตัวแปร หรือโครงสร้างอื่นๆ ที่สามารถทำการกำหนรายละเอียดของ method ไปจนถึงพัฒนา application ได้ ซึ่งนอกจากจะเป็นการแก้ปัญหาการไม่เข้ากันข้างต้นแล้ว ยังช่วยแก้ปัญหาได้อีกหลายประการ เช่น การรวมเอา library ที่มีอยู่แล้วเข้าด้วยกัน เป็นต้น
โครงสร้างการเขียนโปรแกรมของ ODQL อาจสรุปได้ดังนี้

3.2.3.1 ตัวแปร

การประกาศตัวแปรใน ODQL มีรูปแบบดังนี้

<Class Name> <Variable Name>;

อย่างเช่น User e;

เป็นการประกาศว่า e เป็น instance ของ class User

ตัวแปรใน ODQL สามารถอ้างอิงกับ entity (object หรือ instance), atomic literal, class, tuple รวมทั้ง collection of entity อย่างเช่น bag of object หรือ collection of class เช่น ทุก subclass ของ class Person หรือ collection of tuple ตัวอย่างเช่น

String name;

Bag<User> us;

Document class DC;

```
ResultTuple [String] lastName,
                Company userCompany, Integeer salary]
UserInfo;
```

scope ของตัวแปรใน ODQL จะเหมือนกับใน blocked-structured language ทั่วๆอย่างเช่น C\C++ ตัวแปรที่ประกาศไว้ใน block ก็จะคืนหน่วยความจำเมื่อออกจาก block ถ้าเราประกาศตัวแปรใน ODQL interpreter prompt (ซึ่งไม่เป็น block) ค่าของตัวแปรก็จะคงอยู่ไปเรื่อยๆจนกว่าจะจบ session การทำงาน

3.2.3.2 Expression ใน ODQL

นอกจาก query และการประกาศตัวแปรแล้ว ODQL ยังสามารถใช้ expression ทางคณิตศาสตร์ ทางตรรกศาสตร์ได้ด้วย

Expression ทางคณิตศาสตร์จะใช้เครื่องหมายทางคณิตศาสตร์ทั่วไป ได้แก่ บวก(+) ลบ(-) คูณ(*) หาร(/) หารเอาเฉพาะเศษ(remainder, %) โดยจะทำการคูณหารก่อนการบวกลบ จากซ้ายไปขวา และสามารถคำนวณได้ทั้งตัวเลขและตัวแปร ตัวอย่าง expression ทางคณิตศาสตร์ เช่น

```
(15 + YearsInCompany) * 2 – 20.5 * Bonus
```

โดยที่ YearsInCompany เป็นตัวแปรชนิด integer และ Bonus เป็นตัวแปร real expression ทางคณิตศาสตร์อาจอยู่ได้ทั้งใน ด้านขวาของ assignment statement หรืออยู่ใน query expression หรือที่อื่นๆก็ได้

Expression ทางตรรกศาสตร์ก็จะประกอบไปด้วยเครื่องหมาย เท่ากับ(==) ไม่เท่ากับ(!=) น้อยกว่า (<) น้อยกว่าหรือเท่ากับ(<=) มากกว่า(>) มากกว่าหรือเท่ากับ(>=) โดยสามารถใช้ได้กับ literal ทุกชนิด และเครื่องหมายเท่ากับและไม่เท่ากับสามารถใช้กับ object ได้ด้วย ซึ่งเครื่องหมายเท่ากับ(==) จะให้ค่า TRUE เมื่อ argument ทั้งสองอ้างอิงไปยัง object เดียวกัน

นอกจากนั้น ODQL ยังสนับสนุนเครื่องหมาย and, or, not ซึ่ง operator จะต้องเป็น Boolean expression เช่น

```
Boolean b;
Integer year;
User john, bob;
...
b = (year <= 1998 and john.evaluateBonus() < 100.0)
    or (john.getMainManager() ==
    bob.getDepartment().getMainManager());
```

Boolean expression จะใช้ใน where clause ของ query expression เพื่อระบุ object ที่ต้องการ และนอกจากนั้นยังใช้ใน if และ while เพื่อควบคุมลำดับการทำงานของ ODQL program

3.2.3.3 Queries

มีรูปแบบทั่วๆไปดังนี้

<target specification> **from** <from specification> **where** <where specification>

ซึ่งจะได้ผลลัพธ์ออกมาเป็น collection ของ object ซึ่งสามารถนำมาเก็บไว้ในตัวแปรได้ และตัวแปรต่างๆก็สามารถนำมาใช้ได้ทั้งใน target specification, from specification หรือ where specification ตัวอย่างเช่น

```
List<User> ee;
ee = User from User where User.Salary >= 50000;
```

โดยทั่วไปแล้ว ผลของการ query จะเก็บไว้ใน collection เดียวกับใน from specification ซึ่งโดยมากก็จะเป็น instance ของ class นั้นนั่นเอง

เช่น เราสนใจชื่อพนักงานและชื่อผู้จัดการใน class User โดยให้ผลลัพธ์มาเป็น list ของ tuple ซึ่งกำหนดใน target specification

```
List< empNameManagers [empName : String, empManagers : Set<User>]> ee;
Integer s;
…
ee=[u.lastName, u.Managers] from Users u
    where u.Salary >=s and u.rank=="Senior";
```

จะเห็นว่าเราสามารถใช้ตัวแปรใน where specification ได้อย่างไร และ target specification เป็น tuple ซึ่งทำให้ผลลัพธ์ที่ได้มาอยู่ในรูปของ set of tuple และ type ของผลลัพธ์ที่ได้ออกมากับตัวแปร ee จะต้องเป็นชนิดเดียวกัน

3.2.3.4 Path Expressions และ joins

path expression จะถูกใช้ในการ traverse relationship ระหว่าง class เช่น สมมติให้ u เป็น instance ของ class Users เราสามารถอ้างถึงเงินเดือนของผู้จัดการของพนักงานคนนั้นได้ดังนี้

```
u.mainManager.salary
```

path expression ไม่เพียงสามารถอ้างถึง object-valued property(object reference) ได้เท่านั้น แต่ยังสามารถอ้าง method ได้ด้วย อย่างเช่น

```
u.getMainManager().salary
```

เนื่องจากเราสามารถกำหนด path expression ได้โดยตรงเช่นนี้ จึงมักมีเพียงหนึ่ง collection เท่านั้นใน from specification เช่น ถ้าเราต้องการ lastname ของ user ที่ main manager department อยู่ใน New York ก็สารถทำได้ง่ายๆโดยใช้คำสั่ง

```
Set<String> us;
Us = u.lastaName from User u where
        u.mainManager.userDepartmant.address.city == "New York";
```

ซึ่งถ้าไม่ใช้ path expression และความสามารถในการอ้างอิงกันของ object แล้ว มันก็จะเป็น query ที่ซับซ้อนมาก

หรือในบางครั้งเราอาจมีมากกว่าหนึ่ง collection ใน from specification ก็ได้ อย่างเช่น query ด้านล่างนี้จะให้ผลมาเป็น ทุกๆ pair of user ที่อาศัยอยู่ในเมืองเดียวกัน

[u1.lastName, u2.lastName] **from** User u1, User u2

where u1.basicAddress.city=u2.basicAddress.city;

หรืออาจเป็นคนละ collection กันก็ได้ เช่น query ที่ให้ lastName ของหนักงานทุกๆคนที่ได้รับเงินเดือนมากกว่า manager สามารถเขียนได้ดังนี้

u.lastName **from** User u, Manager m **where** u.salary>=m.salary;

3.2.3.5 Exclusive Queries

ใน form specification ถ้าชื่อ class นำหน้าด้วย keyword "alone" นั่นหมายความว่า การค้นหาจะทำที่ instance ของ class นั้นเท่านั้น จะไม่นำเอา instance ของ subordinate class ของ class นั้นมารวม

3.2.3.6 Control Flow Constructs

if-else Statement : มีรูปแบบทั่วไปดังนี้

```
if (<boolean expression>) {
        <statement>
}
else {
        <statement>
};
```

ถ้าค่าใน boolean expression เป็น TRUE ก็จะไปทำ <statement> ใน if clause ถ้าไม่ก็จะไปทำใน else clause

while statement : มีรูปแบบดังนี้

while(<boolean expression>) { <statement> };

ถ้าค่าใน boolean expression เป็นจริง ก็จะทำ <statement> ซ้ำๆไปเรื่อย จนกว่าค่าใน boolean expression จะเป็น FAULSE หรือ NIL ก็จะหยุดทำ แล้วออกจาก loop และเราสามารถใช้คำสั่ง **break** เพื่อบังคับให้ออกจาก loop ทันทีที่โปรแกรมทำงานมาจนถึงคำสั่งนี้ได้อีกด้วย

scan Statement : เป็นคำสั่งที่ใช้กำหนดการทำงานซ้ำๆบนแต่ละ element ของ collection เช่น print ค่าของทุกๆ element เป็นต้น รูปแบบคือ

```
scan (<collection expression variable>,
        <collection element variable> )
{ <statement> };
```

<collection expression variable> หมายถึง collection ที่ต้องการจะทำซ้ำๆ ส่วน <collection element variable> ใช้อ้างถึง element ใน collection เช่น ถ้าเราต้องการแสดงชื่อของทุกๆ element ใน group ก็จะทำได้ดังนี้

```
group    ge;
set<group> GS;
...
scan ( GS, ge) {
        m = ge.name();
        m.print();
}
```

เมื่อครบทุกๆ element ใน GS แล้ว ก็จะออกจาก loop และ ge ก็จะถูก set ให้เป็น NIL และสามารถใช้คำสั่ง **break** ได้เช่นเดียวกับ while โดย ge ก็จะต้องถูก set ให้เป็น NIL เช่นเดียวกัน

3.2.3.7 Collection (Set) Operators

ใน ODQL เราสามารถนำคำสั่งที่เกี่ยวกับ set มาใช้กับ collection ได้ ถ้ามีคำสั่ง Op มากระทำกับ set s1 และ s2

```
S1 Op s2
```

ODQL syntax จะเป็น

```
s1.Op(s2)
```

s1 คือ target object Op คือ operation r2 เป็น argument โดย operation ที่ ODQL สนับสนุนได้แก่ union, intersection และ differ

ยกตัวอย่างเช่น บริษัท 2 บริษัท Acme1 และ Acame2 มี method branchesInCities ซึ่งจะ return set ของเมืองทุกเมืองที่บริษัทมีสาขาอยู่ ถ้าเราต้องการทราบว่าเมืองใดใน California บ้างที่ทั้งสองบริษัทนี้มีสาขาอยู่ ก็จะเขียนได้ดังนี้

```
caAcme1 = Acme1.branchesInCities(CA);
caAcme2 = Acme2.branchesInCities(CA);
commonCities = caAcme1.intersect(caAcme2);
```

หรือถ้าต้องการทราบว่าทั้งสองบริษัทมีสาขาอยู่ที่เมืองใดบ้าง ก็จะเขียนได้ว่า

```
UnionOfBranches = caAcme1.union(caAcme2);
```

หรือถ้าต้องการทราบว่ามีสาขาในเมืองใดบ้างที่บริษัท Acme1 มีสาขาอยู่แต่ Acme2 ไม่มี

```
Acme1Only = caAcme1.differ(caAcme2);
```

3.2.3.8 Populating และ Depopulating Collections

Jasmine ODQL จะมี method สำหรับเพิ่มเข้าและลบ element ออกจาก collection ได้ โดยการเพิ่ม element เข้าใน collection ทำได้โดยใช้ method **add()** และ **directAdd()** ส่วนการลบ element ออกจาก collection จะใช้ method **remove()** และ **directRemove()** เช่น

```
City walnutCreek;
City set branches;
...
Company acme;
...
branches = acme.hasBranches;
branches = branches.add(walnutCreek);
acme.hasBranches = branches;
...
```

คำสั่งข้างต้นนี้จะ load สาขาทั้งหมดที่บริษัทนี้มีไว้ใน working space แล้ว เพิ่ม element เข้าไป จากนั้นก็นำกลับไปใส่ใน property ชื่อ hasBranches ตามเดิม ซึ่ง วิธีนี้มีข้อเสียคือ สิ้นเปลืองทั้งเวลา เนื้อที่ และ I/O ที่ต้องขนถ่ายข้อมูลที่ไม่ได้ใช้ไปมา

เพื่อหลีกเลี่ยงปัญหาเหล่านี้ Jasmine จึงมี method **directAdd()** ขึ้นมา โปรแกรมข้างต้นสามารถเขียนแทนได้ด้วย

```
Acme.hasBranches.directAdd(walnutCreek);
```

และสำหรับการลบ element ออกจาก collection ก็จะใช้คำสั่ง **remove()** หรือ **directRemove()** ซึ่งก็เช่นเดียวกับคำสั่ง **add()** คือ **remove()** จะกระทำกับ collection บน working space ส่วน **directRemove()** ก็จะทำกับ DBspace โดยตรง

3.2.3.9 Method อื่นๆของ Collections

- Aggregate function: เช่น method **count()** ซึ่งจะนับจำนวนของ element ใน collection นั้น
- สำหรับ element ของ collection ที่เป็น numeric จะมี method **sum()**, **average()**, **max()** และ **min()** ซึ่งใช้ในการรวมค่า หาค่าเฉลี่ย ค่าที่มากที่สุด น้องที่สุดของ element ใน collection
- Sorting: method **sort()** จะมี argument 2 ตัวคือ attribute ที่ต้องการเรียงและลำดับการเรียง (จากน้อยไปมา (ASCEND) หรือมากไปน้อย (DESCEND)) เช่น ถ้าเราต้องการเรียงลำดับและ print collection ชื่อ branches โดยเรียงตามชื่อเมืองแบบมากไปน้อยก็จะเขียนได้ว่า

  ```
  branches.sort("city name", DESCEND).print();
  ```

- Duplicate elimination: ถ้า collection ของเรามี literal หรือ object ที่ซ้ำกันอยู่ เราสามารถใช้ method **unique()** กำจัดตัวซ้ำเหล่านั้นได้

3.2.3.10 Update, Creating and Deleting Persistent Objects

ใน ODQL เราสามารถ create และ delete object ได้แม้กระทั่งใน transaction การ update จะมองเห็นเฉพาะ work space ของตนเองจนกระทั่ง transaction นั้น commit ผู้ใช้อื่นจึงเห็นการเปลี่ยนแปลงนั้น

- การ Update object: มี 2 แนวทาง วิธีแรกคือ เรียก method สำหรับ update attribute นั้น ซึ่งเป็นวิธีที่ดี เพราะจะทำให้การเปลี่ยนแปลง implementation ภายใน class ไม่มีกระทบกับ application program เช่น

```
john.changeStatus("vacationing");
```

และวิธีที่ 2 ใช้ assignment statement ใน ODQL version ปัจจุบันจะยังไม่มีการ encapsulation ทำให้เราสามารถแก้ไข property ของ object ได้โดยตรง เช่น

```
john.status = "Vacationing";
```

- การสร้าง object: ODQL จะมี method **new()** สำหรับ create object เช่น

```
classTask t;
t = classTask.new();
```

หรือ

```
t = classTask.new(priority := "High",
                  Status := "Open");
```

- การลบ object: Jasmine ODQL จะมี method **delete()** ใช้ในการลบ object ออกจาก database เช่น

```
t.delete();
```

แต่ ODQL จะไม่จัดการ relationship ระหว่าง object ให้ เช่น ถ้ามี object reference object ที่เราต้องการลบ ODQL ก็จะไม่จัดการให้ ซึ่งอาจทำให้เกิด database error ขึ้นได้

ปัญหานี้อาจแก้ไขได้โดยเราเป็นผู้จัดการกับ relationship นั้นเอง เช่น object user จะอ้างอิงไปถึง object task เราก็อาจกำหนดให้ method delete ของ task ไปลบตัวเองออกจาก object user ด้วย เป็นต้น

3.2.4 Physical Database Construct in Jasmine

3.2.4.1 Stores

class family (CF) ที่เราสร้างขึ้นจะถูกเก็บไว้ใน store และใน store อาจมีได้มากกว่าหนึ่ง CF ถ้าเราสร้างหนึ่ง store ต่อหนึ่ง CF ก็จะทำให้สามารถจัดการ physical storage ของแต่ละ CF ได้โดยอิสระ และถ้า CF นั้นถูก access โดยไม่เกี่ยวข้องกับ CF อื่นบ่อยๆ การแยกแต่ละ CF ให้อยู่บน store ของมันเองจะทำให้การ concurrent ทำได้ดีขึ้น และในทางตรงกันข้าม ถ้าเราสร้าง CF หลายๆ ตัวไว้บน store เดียวกันก็จะทำให้เกินปัญหา course-grained clustering ขึ้นมาได้ เนื่องจาก object จากต่าง CF กันก็จะถูกเก็บอยู่บน store เดียวกัน การรวมเอา CF ที่มักจะต้องถูก access พร้อมๆกันไว้ใน store เดียวกันจะเป็นการลด access time ได้

เมื่อ create store ขึ้นมา เราสามารถกำหนดได้ว่า store นั้นจะเก็บแยกอยู่บนกี่ file ซึ่งทำให้เราสามารถแยกให้ store กระจายไปอยู่ในหลายๆ drive ได้ ซึ่งทำให้เราสามารถใช้เนื้อที่ disk ได้อย่างมีประสิทธิภาพยิ่งขึ้น

Jasmine อนุญาตให้จัดการกับ store ได้ 4 วิธี คือ **createStore, extendStore, deleteStore** และ **listStore** และก่อนที่จะ run คำสั่งเหล่านี้ จะต้องให้ Jasmine server run อยู่ใน development mode เสียก่อน เพราะคำสั่งเหล่านี้จะไปเรียก method ของ system ในฝั่งของ server

คำสั่ง **createStore** ใช้ในการสร้างส่วยขยาย (extent) แรกของ store ขึ้นมา **extendStore** ใช้สำหรับสร้างส่วนขยายต่อๆมาของ store ที่มีอยู่แล้ว **deleteStore** ใช้ลบ store ที่กำหนด ส่วนการจะดูรายละเอียดของ store อย่างเช่น page size จำนวน page ที่ใช้ ก็จะใช้คำสั่ง **listStore**

3.2.4.2 Indexs

เมื่อ store, CF และ class ถูกสร้างขึ้นมาเรียบร้อยแล้ว เราอาจสร้าง index ขึ้นมาเพื่อปรับปรุงประสิทธิภาพของ database

Jasmine จะสนับสนุน index 2 ประเภทคือ system-defined primary index และ user-defined index โดย index ทั้งสองประเภทจะมีโครงสร้างเป็น B+ tree

เมื่อ class ถูกสร้างขึ้น primary index ก็จะถูกสร้างขึ้นมาตาม OID โดยอัตโนมัติ ส่วน index ชนิดที่สองสร้างโดยใช้ method **createIndex()** โดยสามารถสร้างได้บน single-value instance-level attribute เท่านั้น Jasmine version ปัจจุบันยังไม่สามารถกำหนด index บน class-level attribute หรือ set-value attribute ได้

Index จะไม่ถูก inherit โดย subclass เพราะหาก query ที่ compile แล้วมีการใช้ index ที่ถูก drop ไปแล้ว จะทำให้เกิดปัญหาขึ้นได้

เราสามารถ create หรือ drop index ได้ตลอดเวลาที่ server run เป็น development mode และ ระดับของ constraint check เป็น intermediate โดยการ drop สามารถทำได้โดยใช้ method **dropIndex()** ถ้า class ถูก delete ทุกๆ index บน class นั้นจะถูก drop โดยอัตโนมัติ และเมื่อเราสร้าง index ขึ้นมาจะทำให้การ update ของ class นั้นช้าลงตามไปด้วย

IndexScan เป็นคำสั่งที่ใช้ index ในการ scan instance ของ class ซึ่งทำให้ได้ใช้ประโยชน์จาก index ในแง่ performance อย่างเต็มที่ แต่จะทำให้ application ของเรานั้นไม่ physical data independence เพราะถ้า index นั้นถูก drop application นั้นก็จะต้องถูกแก้ไขแล้ว compile ใหม่

รูปแบบของคำสั่ง indexScan คือ

```
IndexScan ( <class>, <variable>, <index>, <comparison>, <constant-value>)
{       <statement>     }
```

ซึ่งจะ scan เฉพาะ class ที่กำหนดไว้เท่านั้น ไม่ scan ต่อใน subordinated class โดยคำสั่งภายใน indexScan เราไม่ควรทำอะไรที่จะมีผลกระทบกับ index ไม่ว่าจะเป็นการ create delete instance, update

attribute ของ index นั้น, ใช้คำสั่ง indexScan ซ้อนบน index เดียวกัน, create หรือ drop index ของ class และของ superclass, delete target class, เปลี่ยน uniqueness property หรืออื่นๆที่จะกระทบกับ index

ตัวอย่างนี้จะเป็นการ scan collection ของ instance ของ class Video และ print instance ชื่อ "Days of being wild" ไปจนถึง "Don Quixote"

```
Video v;
Video.createIndex("tindex", "title");

IndexScan(Video, v, "tindex", ODB_EQGREATER, "Days of being wild") {
    If (v.title > "Don Quixote") { break; };
};
```

3.2.5 Using Jasmine's ODQL

เราอาจใช้ ODQL สั่งงาน Jasmine ได้ 3 วิธีคือ

1. **Interpreted ODQL** : ผู้พัฒนาระบบรวมทั้ง administrator สามารถ run the client ODQL Interactive Environment (CODQLIE) ในการสั่งงาน Jasmine ซึ่งก็คล้ายๆ กับ SQL Interpreter ทั่วๆไป
2. **Embedded ODQL** : ใช้สำหรับกำหนด method ใน คำสั่ง defineProcedure() และ addProcedure() ซึ่งสามารถไปฝังอยู่กับภาษา host อย่างเช่น C/C++ ได้
3. **C API** : เป็น library ซึ่งให้เราติดต่อกับ Jasmine server ได้อย่างเต็มที่ ทั้งเพื่อการจัดการ database และพัฒนา application

3.3 ระบบจัดการฐานข้อมูลเชิงวัตถุ : Jasmine

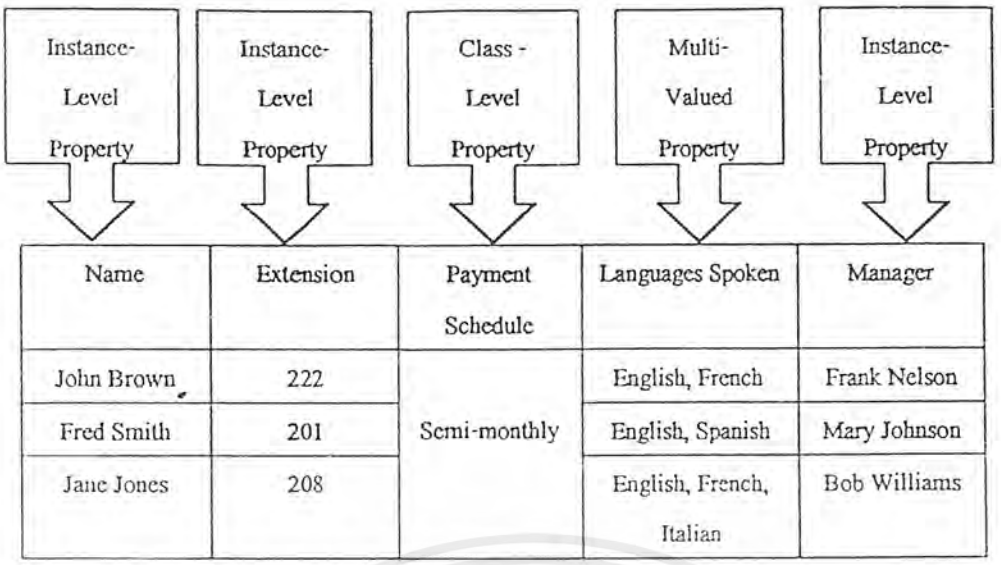
3.3.1 ภาพโดยรวมของระบบจัดการฐานข้อมูล Jasmine

ในมุมมองของ Jasmine object จะมีองค์ประกอบพื้นฐาน 2 ส่วนหลักคือ

- **Property** : เป็นสิ่งที่บ่งบอกถึงค่าหรือข้อมูลบน object นั้น เช่น พนักงานจะมีชื่อ, หมายเลขบัตรประชาชน, เงินเดือน เป็น property
- **Method** : เป็นการแสดง behavior ของ object เช่น การเปลี่ยนงาน, การขึ้นเงินเดือน ซึ่งมีรายละเอียดดังนี้

3.3.1.1 Property

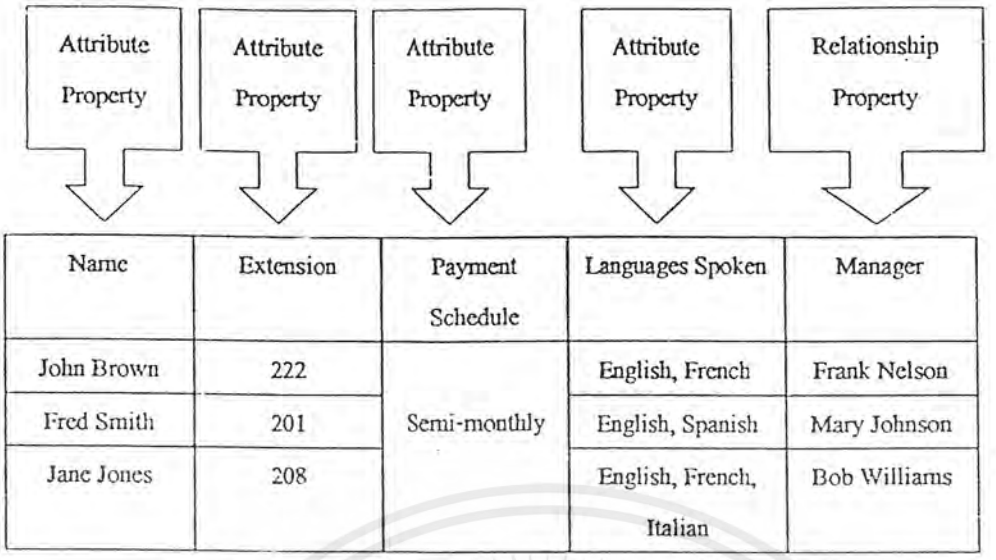
Property จะบอกเราว่า object นั้นมีลักษณะอย่างไรและช่วยแบ่ง object ต่างๆ ออกเป็นกลุ่มได้ property ของ object หนึ่งก็ถูกกำหนดเป็นส่วนหนึ่งของข้อมูลภายใน class และเราจะเรียกข้อมูลของ property ต่างๆ ภายใน class ว่า metadata



รูปที่ 3.20 แสดงตัวอย่าง property แต่ละชนิด

Jasmine แบ่ง peroperty ออกเป็น 5 ส่วนคือ

1. **Instance-level property :** เป็นคุณสมบัติของแต่ละ object เช่นจากรูปที่ 3.20 ด้านบน Instance-level property คือ name และ extension
2. **Class-level property :** เป็นคุณสมบัติของ class และ object ทั้งหมดใน class นั้น class-level property ใช้ในกรณีที่ instance ทั้งหลายใน class ต้องการใช่ค่านี้ร่วมกัน เช่น property ที่เก็บเงินเดือนเฉลี่ยของพนักงานทั้งหมดใน class นั้น
3. **Multi-value property :** ทั้ง instance และ class-level property มีคุณสมบัติเป็น multi-value ได้ คือ สามารถมีค่าแสดง property เดียวได้มากกว่า 1 ค่า
4. **Attribute and Relationship property :** attribute คือ property ที่มีลักษณะเป็นรูปแบบข้อมูลพื้นฐานไม่ซับซ้อน เช่น number, string หรือ date ส่วน relationship คือ property ที่เป็นรูปแบบข้อมูลที่มีลักษณะเป็นออบเจ็กต์ซับซ้อน และการอ้างอิงจะใช้ OID เพื่อแสดงความสัมพันธ์กับออบเจ็กต์อื่นดังรูปที่ 3.21
5. **Many-to-Many Relationship :** คล้ายกับ relationship property แต่มีข้อดีเหนือ relational database เนื่องจาก relational database ต้องการคีย์เมื่อต้องการเชื่อมกับตารางอื่น



รูปที่ 3.21 แสดง property แบบ attribute และ Relationship

3.3.1.2 Method

แบ่งออกเป็น 3 แบบ คือ

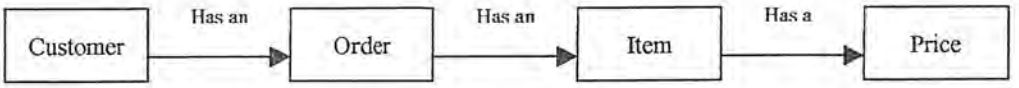
1. **Instance-level Method :** เป็น method ที่อยู่ในระดับออบเจ็กต์ เช่น method ที่บอกถึงปีที่แต่ละ employee ทำงานให้กับองค์กร
2. **Class-level Method :** เป็น method ที่อยู่ในระดับคลาส เป็น method ที่ทำหน้าที่ในการสร้างใหม่ หรือ method ที่สามารถสร้าง instance ให้กับคลาส โดยใช้การข้อมูลใน external file
3. **Collection-level Method :** Collection คือการรวมออบเจ็กต์ที่มีลักษณะเดียวกันเข้าไว้ด้วยกัน ซึ่ง instance และ class-level method ก็สามารถเป็น collection ได้ เช่น method ที่ทำหน้าที่ในการเพิ่มเงินเดือนให้กับกลุ่มของ employee

3.3.2 Inheritance (is-a relationship)

คือความสามารถในการอนุญาตให้สร้าง class ใหม่ (เรียก subclass) ที่มี property และ method ทั้งหมดที่อยู่ใน class ข้างต้นอยู่แล้ว โดยสามารถเพิ่ม property และ method ใหม่ๆ หรือทำการเปลี่ยนแปลงแก้ไขได้ ใน Jasmine สนับสนุน multiple inheritance นั่นคือการที่ subclass สามารถ inherit property และ method จาก superclass มากกว่าหนึ่ง class ได้ แต่เนื่องจาก superclass อาจมี property หรือ method ชื่อซ้ำกันได้ Jasmine แก้ปัญหานี้โดยการให้ผู้ใช้ยืนยันว่า method ใด property ใดเป็นของใคร

3.3.3 Aggregation (has-a relationship)

ใช้อธิบายความสัมพันธ์ระหว่าง object โดยเกิดจากการที่ object หนึ่งได้อ้างอิงกับ object อื่น ดังรูปที่ 3.22



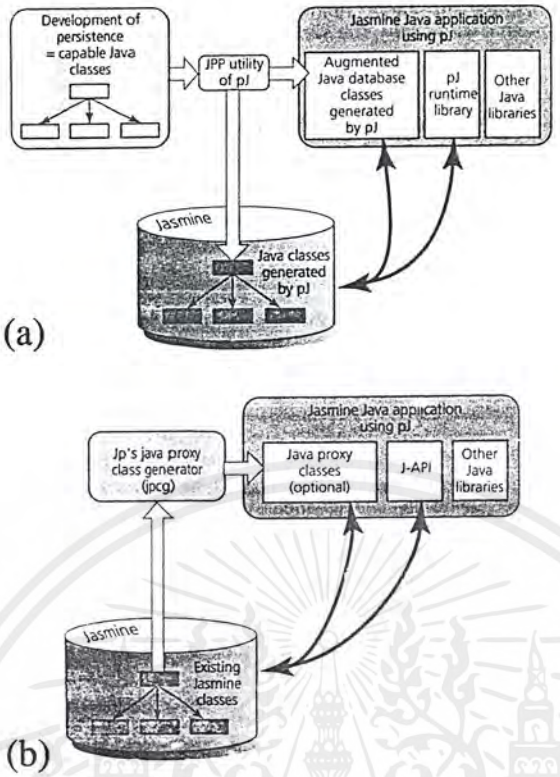
รูปที่ 3.22 ความสัมพันธ์แบบ aggregation

3.3.4 Jasmine กับ Java

การใช้ภาษา Java ติดต่อกับ Jasmine สามารถทำได้ 2 วิธีคือ

1. **Persistence Java (pJ) :** ทำได้โดย กำหนดคลาสขึ้นมาด้วยภาษา Java และใช้ preprocessor ที่ชื่อ pJ's JPP ในการ map จากคลาสของ Java ไปเป็นคลาสของ Jasmine ซึ่งแนวคิดนี้จะเหมือนกับ ODMG 2.0 Java language binding architecture
2. **Java Proxy (Jp) :** วิธีนี้ ผู้ใช้จะต้องกำหนด Schema ขึ้นมาบน Jasmine และใช้ tool ที่ชื่อว่า Java class generator tool (jpcg) generate ออกมาเป็น Java คลาส (เรียก proxy คลาส) ซึ่งสามารถนำไป compile และใช้ใน Java ได้ ซึ่งผู้พัฒนาโปรแกรมสามารถเข้าถึงและจัดการกับ Jasmine คลาสและ Instance ได้โดยผ่าน proxy คลาสนี้

นอกจากนั้น Jp จะมี Java application programming interface (J-API) ให้ผู้พัฒนาสามารถเข้าถึงและจัดการกับ Jasmine database โดยผ่าน Java ได้อีกด้วย



รูปที่ 3.23 เปรียบเทียบ pJ และ Jp โดย (a) คือการพัฒนา Java application ด้วย pJ และ(b) คือการพัฒนา Java pplication ด้วย Jp

บทที่ 4
ฐานข้อมูลเชิงเวลาบนฐานข้อมูลเชิงวัตถุ

4.1 การนำ OID มาใช้

Object Identity (OID) เป็นสิ่งที่ใช้ระบุหรืออ้างอิงออบเจ็กต์ ซึ่งแต่ละออบเจ็กต์จะมี OID ไม่ซ้ำกัน OID จะถูกกำหนดให้โดยระบบตั้งแต่เมื่อออบเจ็กต์ถูกสร้างขึ้นและจะไม่สามารถเปลี่ยนได้

ปัญหาของการพัฒนาฐานข้อมูลเชิงเวลาบนระบบเชิงสัมพันธ์คือ Identifier เปลี่ยน เพราะใน Relational Model จะใช้ External Identifier ซึ่งก็คือ field (หรือกลุ่มของ field) ข้อมูลมาเป็น Identifier หากไม่ได้รับการออกแบบที่ดี External Identifier นี้อาจมีเปลี่ยนแปลงได้ เช่นในบางครั้งรหัสวิชาอาจเปลี่ยนได้

แต่ในระบบที่เป็นเชิงวัตถุทุกออบเจ็กต์จะมี OID เป็นของตัวเอง ซึ่งตัวระบบจะกำหนดให้เมื่อออบเจ็กต์นั้นถูกสร้างขึ้นมา และ OID นี้จะไม่ซ้ำกันอย่างแน่นอน เรียก OID นี้ว่าเป็น Internal Identifier ซึ่งจะไม่มีส่วนเกี่ยวข้องใดๆ ทั้งสิ้นกับข้อมูล ไม่ว่าข้อมูลจะเปลี่ยนไปอย่างไร OID ก็จะเป็นค่าเดิมสำหรับออบเจ็กต์เดิมเสมอ

ระบบที่เป็นเชิงวัตถุซึ่งมี OID นี้สามารถช่วยแก้ปัญหาเรื่อง Identifier เปลี่ยนของระบบที่เป็นเชิงสัมพันธ์ ได้ เพราะแม้ว่า External Identifier เช่น รหัสวิชา จะเปลี่ยนไป แต่ Internal Identifier หรือ OID ก็จะคงเดิมเสมอ

4.2 Extension for Time in Database

ในการ implement temporal application นั้น สิ่งที่ non-temporal database system ต้องสนับสนุนมี 3 ประการคือ

1. data structure จะต้องสามารถเก็บข้อมูลเกี่ยวกับเวลาได้

การเพิ่ม date/time attribute เข้าใน data structure นั้นสามารถทำได้ง่ายมาก เราอาจเก็บ valid time โดยเพิ่ม attribute VTS (Valid Time Start) และ VTE (Valid Time End) ให้กับ class Date ส่วน transaction time ก็เช่นเดียวกัน

แต่สำหรับ operation สำหรับ temporal จะซับซ้อนกว่า เช่น ผลต่างของสองช่วงเวลาอาจได้ผลออกมาเป็นช่วงเวลาหลายๆช่วง (set of interval) ซึ่งโดยทั่วไปจะใช้ temporal element ซึ่งเป็น set of interval เข้ามาช่วยเรื่อง timestamp

สำหรับ relational data model นั้นส่วนที่ซับซ้อนที่สุดคือคือ timestamp ของส่วนของ data structure อย่างเช่น tuple หรือ attribute timestamp ซึ่งใน object data model อาจทำได้ 3 แนวทาง คือ

1. timestamp ที่ attribute หรือ timestamp ในระดับ object
2. timestamp ในระดับ type
3. timestamp ที่ระดับ identifier

2. operation ต่างๆสำหรับ temporal เพื่อการ query และ modify database

การเพิ่ม temporal operation เป็นเรื่องที่มีปัญหามาก จาก temporal algebra ที่ได้มีการนิยามไว้แล้ว เราอาจนำไป implement โดยตรงกับ system หรือทำเป็น layer อยู่บน non-temporal system ซึ่งก็จำเป็นต้องมีส่วนขยายของ query language ให้สามารถใช้ temporal operation ร่วมกับส่วนที่เป็น non-temporal อื่นๆได้

แต่ OODBMS อย่างเช่น Jasmine เราสามารถเพิ่ม class เกี่ยวกับ temporal และ function เกี่ยวกับเวลา ซึ่งจะทำงานเข้ากันกับ non-temporal function ที่มีอยู่แล้วได้ง่าย

3. temporal constraint

ใน temporal relational database ตัว system ทำหน้าที่จัดการ temporal constraint อย่างเช่น referential integrity ให้ แต่สำหรับ OODBMS รวมทั้ง Jasmine จะให้ programmer จัดการกับ temporal constraint ที่เพิ่มขึ้นมาเอง

4.3 ประเภทของ TOM

การนำเอา object-oriented database มาประยุกต์ใช้กับ temporal นั้นอาจทำได้ใน 2 วิธีคือ

- ขยาย data model เดิมให้สนับสนุน temporal เช่น Temporal Object Data Model (TOM) ซึ่งขยายมาจาก Object Model (OM)
- แบบ root class คือ ให้มี class เกี่ยวกับเวลา แล้วให้เป็น superclass ของ class ที่ต้องการให้มี timestamp บน OODBMS ต่างๆ เช่น Jasmine

1. Temporal Object Model (TOMs)

ในตัวอย่างนี้จะใช้ temporal object data model (TOM) ซึ่งขยายมาจาก Object Model (OM)

TOM จะทำ timestamp ที่ระดับ object โดยจะเพิ่ม timestamp เข้าไปกับชื่อของ instance คือ จะไม่เพิ่มที่ type แต่เพิ่มที่ object identifier โดย temporal object identifier (toid) จะเป็น

Toid := <<oid; ls>>

ซึ่ง oid คือ object identifier (OID) ส่วน ls คือ lifespan ของ object จะเห็นได้อย่างชัดเจนว่าจะไม่ทำ timestamp ที่ value ของ object แต่ทำที่ตัว object เลย

การทำ timestamp บน relationship ของ object ก็สามารถทำได้ อย่างเช่น บนความสัมพันธ์ (o_1,o_2) ระหว่าง object o_1 และ o_2 เมื่อแปะ timestamp เข้าไปแล้วจะได้เป็น

<<(o_1,o_2); ls>>

association และ collection ของ object ก็สามารถทำ timestamp ได้เช่นเดียวกัน

สำหรับการใส่ timestamp ให้กับ attribute ของ object นั้น เนื่องจากระบบของเราทำ timestamp ที่ระดับ object เราจะใช้วิธียก attribute ที่ต้องการ stamp เวลาขึ้นมาเป็น object หรือ relationship ใหม่เลย เช่นถ้า

ต้องการเก็บประวัติเงินเดือนของพนักงาน เราอาจเก็บเงินเดือนของพนักงานไว้เป็น object ชื่อ salaryHistory และ attribute salary ของ object employee ก็จะเก็บ set ของ timestamped object ชื่อ salaryHistory เป็นต้น

การสร้าง class และ timestamped collection บน TOM สามารถทำได้ดังนี้

```
create type department(
        Dno: integer;
        Name: string;
        Numbers: collection(employee) );
create type employee(
        Name: string;
        Salary: integer;
        Dept: department);

create collection Departments
        type department
        LifeSpan { (80-Inf) };
create collection IS_Staff
        type department
        LifeSpan { (90-Inf) };
create collection Math_Staff
        type department
        LifeSpan { (80-Inf) };
```

query language ของ TOM นั้นจะมีลักษณะคล้ายกับ ODL แต่ keyword "valid" ที่อยู่ตอนต้นของ query นั้นเป็นตัวบอกว่า query นี้เป็น temporal เช่น ถ้าต้องการทราบว่าพนักงานคนใดได้รับเงินเดือนสูงสุด ก็อาจเขียนได้ว่า

```
valid
  select s1.Name, s1.Salary
  from s1 in IS_Stasff
  where not exists
        ( select *
          from s2 in IS_staff
          where s1.Salary < s2.Salary);
```

หรือถ้าต้องการทราบว่าใครที่เป็น staff ทั้ง IS_Staff และ Math_Staff ก็จะเขียนได้ว่า

valid

IS_Staff intersect Math_Staff

2. Jasmine

จะเป็นการพัฒนา root class ที่สนับสนุน attribute และ method เกี่ยวกับเวลาขึ้นมาก เมื่อคลาสหรือออบเจ็กต์ใดต้องการใช้ความสามารถด้านเวลา ก็จะสืบทอดคุณสมบัติไปจากคลาสนี้

ซึ่งในครั้งนี้เราจะพัฒนาบน OODBMS ชื่อ Jasmine โดยใช้ภาษา ODQL ในการเขียน

2.1 Timestamps in Jasmine

ช่วงเวลา (time interval) ซึ่งเป็นหน่วยของเวลาขั้นพื้นฐานของเรากำหนดได้ดังนี้

```
defineClass Interval {
  Date VTS;
  Date VTE;
};
```

ซึ่งขอบเขตล่างจะเป็นขอบเขตปิด ขอบเขตบนจะเปิด เช่น [90-96) จะหมายถึงวันที่ 1 มกราคม 1990 ถึง 31 ธันวาคม 1995

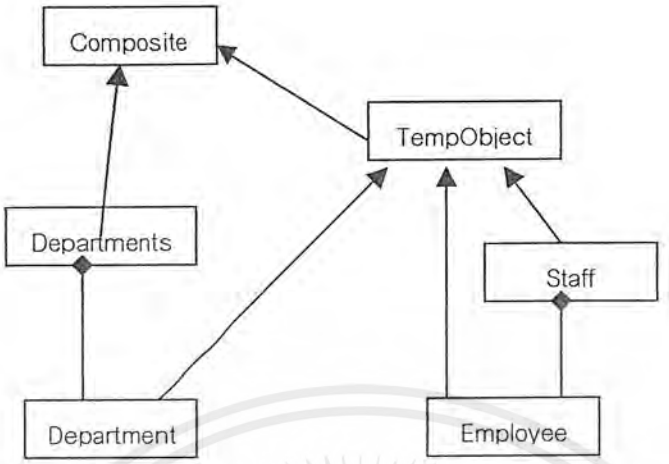
object ที่เป็นสมาชิกของ set สอง set จะต้องมีค่าของ attributre เหมือนกัน แต่ในความเป็นจริงแล้ว object หนึ่งๆอาจมีได้หลายๆ role เช่น พนักงานชื่อ Midas เข้าเป็นทั้งสมาชิกในกลุ่ม IS และ Math เราก็ต้อง create object ชื่อ Midas ขึ้นมา 2 object แต่ valid time ของ Math_staff อาจเป็น [93-97) แต่ valid time ของ IS_Staff เป็น [96-∞) ก็ได้ ซึ่งก็จะเกิดปัญหาว่า object ใดบ้างที่ในความเป็นจริงแล้วเป็น obejct เดียวกัน (same real world entity) การแก้ปัญหาอาจทำได้ในหลายๆวิธี เช่น เพิ่ม attribute ที่เป็น set ของ obejct ที่เป็น same real world entity เข้าไปในแต่ละ object หรือเราอาจให้มี object เก็บ role ของ object อื่น ซึ่งทั้งสองวิธีนี้จะทำให้เราต้องจัดการกับ pointer จำนวนมาก วิธีที่ดีกว่าน่าจะเป็นเพิ่ม attribute ชื่อ key เข้าไปใน object ซึ่งค่านี้จะ unique สำหรับ real world entity เดียวกัน

class TempObject กำหนดได้ดังนี้

```
defineClass TempObject
{
  Instance:
    Set<Interval> VALID;
    Integer KEY;
    ...
};
```

ซึ่ง class ใดต้องการให้ instance ของตนเองมี timestamp ก็จะ inherit จาก class TempObject นี้ ตัวอย่างเช่น class Departments เป็น set ของ object department ซึ่งแต่ละ department เราจะเก็บ department

number, ชื่อ และสมาชิก โดยสมาชิกของแต่ละ department จะเก็บใน object Staff ซึ่งก็คือ set ของ object employee อีกทีหนึ่ง เราสามารถกำหนด class ได้ดังนี้



รูปที่ 4.1 class hierarchy ของ Department และ user

```
defineClass Departments
{
  instance:
    set<Department> depts;
};

defineClass Department
super: TempObject
{
  instance:
     integer Dno;
     string Name;
     Staff  Members;
};

defineClass Staff
super: TempObject
{
  instance:
     set<Employee> members;
```

```
};

defineClass Employee
super: TempObject
{
  instance:
    string Name;
    integer  Salary;
    Department Dept;
};
```

และถ้าเราต้องการเก็บประวัติของเงินเดือนของพนักงาน เราก็จะทำเช่นเดียวกับใน TOM คือ ยก Salary_history ขึ้นเป็นอีก object หนึ่ง ซึ่ง class ของ Salaty_history ได้ดังนี้

```
defineClass Salary_history
super: TempObject
{
  instance:
    integer Salary;
};

defineClass Employee
super: TempObject
{
  instance:
    string Name;
    Set<Salary_history> Salaries;
    Department  Dept;
};
```

2.2 Operations on Timestamps

method พื้นฐานที่ทำให้ผู้ใช้สามารถเขียน temporal query ได้เท่ากับการใช้ temporal relational algebra ได้แก่ **T_INTERSECT, T_MINUS, T_FLATTEN**

T_INTERSECT จะใช้ในการ intersection ระหว่าง set สอง set เช่น การ intersection ระหว่าง [94-∞) กับ [90-96) จะได้เป็น [94-96) ส่วน T_MINUS จะใช้ในการหาผลต่างระหว่าง set สอง set เช่น ผลต่างระหว่าง

[94-∞) กับ [90-96) จะได้เป็น [96-∞) ส่วน method T_FLATTEN จะใช้ในการ flatten set ของ temporal element จากการ query ที่ return result เฉพาะ attribute timestamp เท่านั้น

เรากำหนด class TempObject เพิ่มเติมดังนี้

```
DefineClass TempObject
{
  instance:
    Set<Interval> VALID;
    integer KEY;

    Set<Interval>  T_INTERSECT( Set<Interval>  T);
    Set<Interval> T_MINUS( set<Interval>  T);
    Set<Interval> T_FLATTEN( set<Interval)>  T);
    ...
};
```

หรือนอกจากนี้จะมี temporal predicate ที่ใช้ในการเปรียบเทียบ เช่น **T_BEFORE, T_MEETS, T_OVERLAP** หรืออื่นๆก็ได้

ตัวอย่างการใช้งาน method เหล่านี้ เช่น เรามีข้อมูลของ Staff และ Department ดังนี้

Name	Salary	Dept	KEY	VALID
Andreas	10000	IS	1	{[93-∞)}
Alain	9000	IS	2	{[95-∞)}
Antonia	11000	IS	3	{[96-∞)}
Martin	8000	IS	4	{[92-94)}
Martin	10500	IS	4	{[94-∞)}
Moira	20000	IS	5	{[94-∞)}
Midas	30000	IS	6	{[96-∞)}
Moira	8000	Math	5	{[86-90)}
Midas	40000	Math	6	{[93-97)}
John	45000	Math	7	{[94-∞)}

ตารางที่ 4-1 ข้อมูลของ Staff

Dno	Name	Members	KEY	VALID
3	Inf Systems	IS_Staff	10	{[90-∞)}
9	Mathematics	Math_Staff	11	{[80-∞)}

ตารางที่ 4-2 ข้อมูลของ Department

ถ้าเราต้องการทราบ history ของเงินเดือนสูงที่ของ IS_Staff เราจะเขียน temporal query ได้ดังนี้

```
[Name : s1.Name,
 Salary : s1.Salary,
 VALID : s1.T_MINUS
          (s1.T_FLATTEN
           (s1.T_INTERSECT(s2.VALID)
            from s2 in IS_Staff.members
            where s1.Salary < s2.Salary) ) ]
from IS_Staff.members s1;
```

ซึ่งจะได้ผลลัพธ์ดังนี้

Name	Salary	VALID
Andreas	10000	{[93-94)}
Alain	9000	{ }
Antonia	11000	{ }
Martin	8000	{[92-93)}
Martin	10500	{ }
Moira	20000	{[94-96)}
Midas	30000	{[96-∞)}

ตารางที่ 4-3 ผลลัพธ์จากการค้นหา

ช่อง VALID ที่เป็น set ว่างหมายถึง พนักงานเหล่านั้นไม่เคยได้รับเงินเดือนมากกว่าใครเลย

4.4 เปรียบเทียบ

1. Query language

จากตัวอย่างข้างต้นเราจะเห็นได้ว่า query บน Jasmine นั้น เป็นเรื่องที่ยุ่งยากมาก query จะง่ายกว่านี้มากถ้า operator ของ temporal จะเป็น infix operator ไม่ใช่ function อย่างข้างต้น

ถ้าเราให้ temporal operator ของเรามีลักษณะเป็น keyword พิเศษ อยู่ข้างหน้า non-temporal query ธรรมดา อย่างในตัวอย่างของ TOM จะไม่เพียงลด error และเพิ่มความเข้าใจให้กับ programmer แล้ว ยังให้การย้ายจาก non-temporal มาเป็น temporal query ได้ง่ายขึ้นด้วย เช่น ตัวอย่างของ Jasmine เราสามารถเขียนแทนได้ด้วย

```
valid
    select tuple (Name : s1.Name,
                  Salary : s1.Salary)
    From s1 in IS_Staff.members
    Where not exist s2 in IS_Staff.members : s1.Salary < s2.Salary;
```

แทนที่จะเปลี่ยน syntax ของ query language ระบบบางระบบจะยอมให้เราเปลี่ยน semantic ของ query โดย override operation บางตัวได้ เพื่อตอบสนองความต้องการพิเศษของแต่ละ application อย่าง เช่น temporal ต้องการ

2. Constraint

constraint ใช้เป็นตัวกำหนดว่า state ที่ถูกต้อง database จะเป็นอย่างไร ซึ่งเมื่อเราเพิ่มเวลาเข้าไปใน database เราก็ต้องมี constraint เพิ่มขึ้นมาด้วย

semantic ของ referential integrity จะเปลี่ยนไป เราไม่เพียงต้องตรวจสอบว่า object ที่ถูกอ้างถึงนั้นมีอยู่จริง (exist) เท่านั้น เราต้องดูต่อว่ามันมีอยู่จริงในช่วงเวลานั้นด้วยหรือไม่

ความสัมพันธ์ของ subclass ก็เป็นสิ่งที่สำคัญใน temporal database คือ instance ของ subclass จะมีอยู่ได้ในช่วงที่มันเป็น instance ของ superclass เท่านั้น เช่น object ของ class Employee จะมีอยู่ (exist) ได้ก็เฉพาะในช่วงที่มันเป็น instance ของ superclass Person เท่านั้น

temporal membership constraint เช่น object Employee เป็นสมาชิกของ instance ของ class staff ได้เฉพาะช่วงชีวิต (lifespan) ของมันเท่านั้น และเช่นเดียวกัน instance ของ class Staff ก็จะเก็บเฉพาะ object Employee ที่มีช่วงชีวิตอยู่ในช่วงเดียวกันมันเท่านั้น

ถ้าเราใช้ database system อย่างเช่น Jasmine เราสามารถสร้าง method สำหรับตรวจสอบ constraint ต่างๆ เมื่อมีการ update ค่าใน attribute ซึ่งจะทำให้ programmer ต้องยุ่งยากกว่าการให้ system ทำหน้าที่จัดการ constraint ให้

ส่วน TOM นั้นจะสนับสนุน temporal referential integrity, temporal subclass relationship, temporal membership, temporal partition, cover และ intersection constraint ซึ่งจะ check เมื่อ user commit

3. Optimization

เมื่อเรา delete object ใน Temporal database object นั้นจะไม่ได้ถูกลบจริงๆ จะเป็นการ set VTE เป็นวัน/เวลาปัจจุบันเท่านั้น ซึ่งทำให้ระบบของเราจะต้องเก็บและจัดการข้อมูลจำนวนมากมาย และ temporal operation ก็ทำให้ความซับซ้อนในการ query เพิ่มขึ้นด้วย

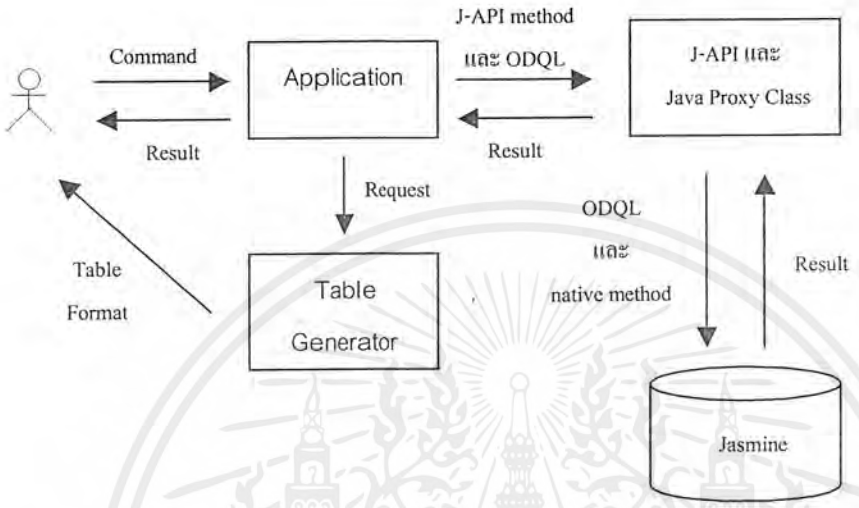
การเพิ่ม function และ predicate เกี่ยวกับเวลาใน object-oriented database system นั้น จะทำให้ข้อมูลแบบ temporal ของเราไม่สามารถใช้ประโยชน์จากการ optimization ของ system ได้ เพราะตัว database จะไม่รู้จัก semantic ของ data หรือ function ที่เกี่ยวกับเวลาเลย

แต่ถ้าเราเพิ่ม temporal data structure และ operation เข้าไปในระบบ ก็จะทำให้ system สามารถใช้ประโยชน์จากการ optimization ได้

แต่แทนที่จะสร้าง database system ขึ้นมาเป็นจำนวนมากเพื่อสนับสนุน data ชนิดต่างๆ ระบบส่วนมากจะยอมให้ผู้ใช้แก้ไข (modify) หรือขยาย (extend) semantic ของ application นั้นเพื่อสามารถเพิ่มประสิทธิภาพในการประมวลผลข้อมูลแบบที่ต้องการได้

บทที่ 5
ระบบสารสนเทศนักศึกษาเชิงเวลาบนฐานข้อมูลเชิงวัตถุ

5.1 สถาปัตยกรรมของระบบ



รูปที่ 5.1 สถาปัตยกรรมของโปรแกรม

5.2 อุปกรณ์ที่ใช้

Hardware :

Server and Local Client Computer

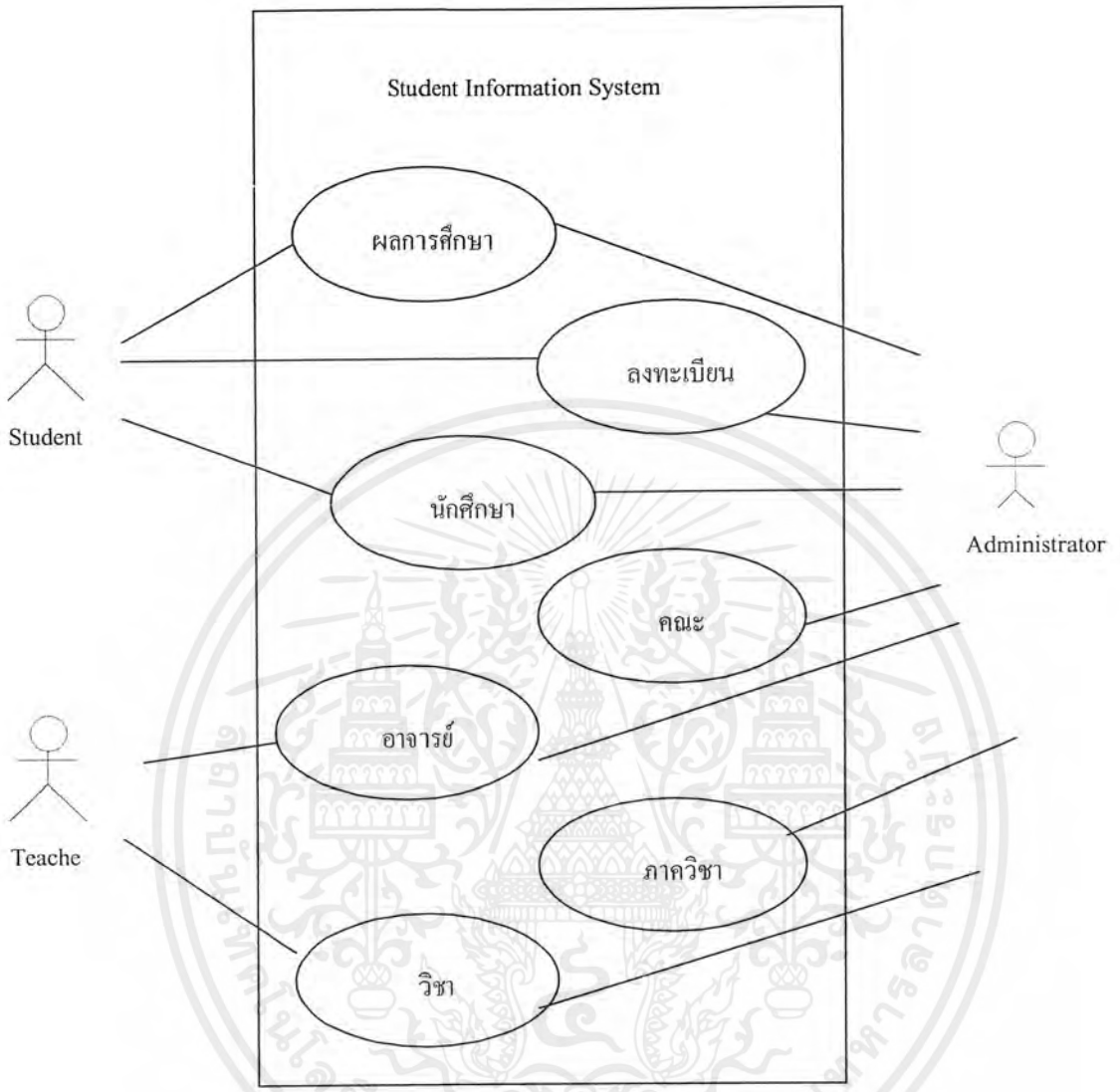
- Celeron 633 MHz
- RAM 160 MB
- Harddisk 60 GB

Software :

- ระบบปฏิบัติการ Windows NT 4 Server Service Pack 6
- Jasmine Developer Edtion 1.2
- Microsoft Visual C++ 6
- Java Development Kit (JDK) 1.1.8
- Visual Age for Java Enterprise Edition version 2

5.3 การออกแบบ

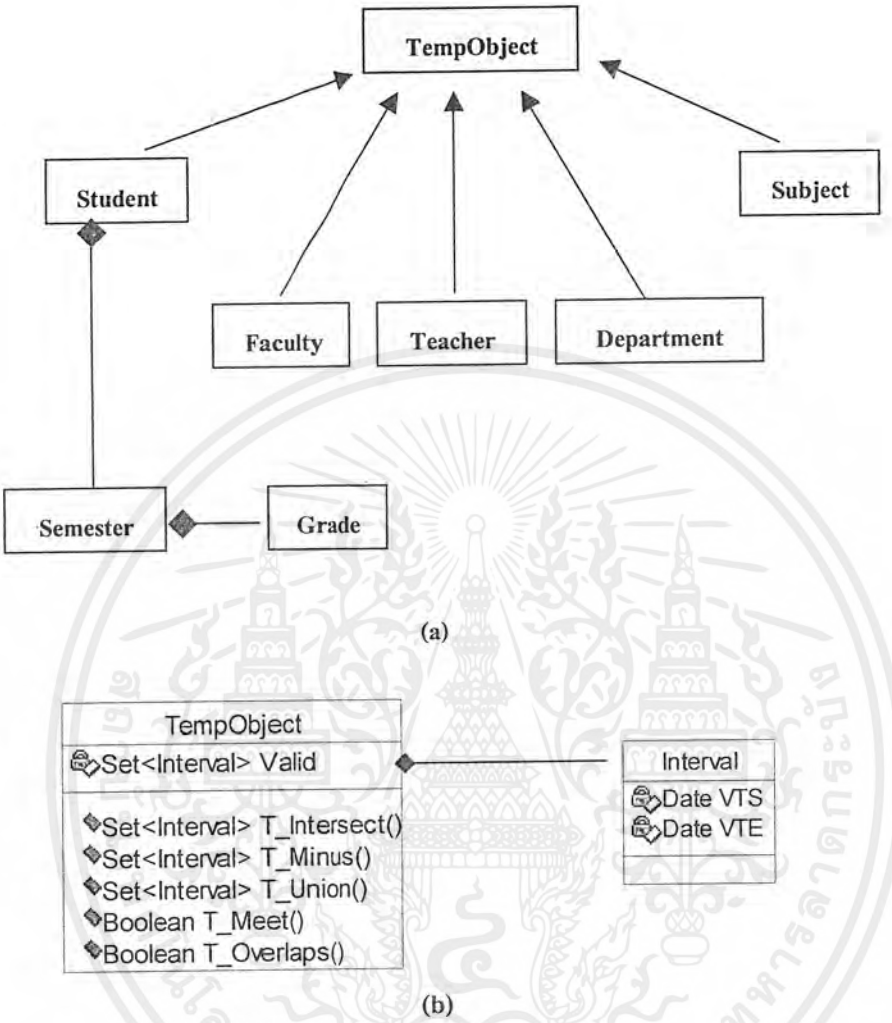
1. Use Case Diagram



รูปที่ 5.2 Use Case Diagram

2. Class Diagram

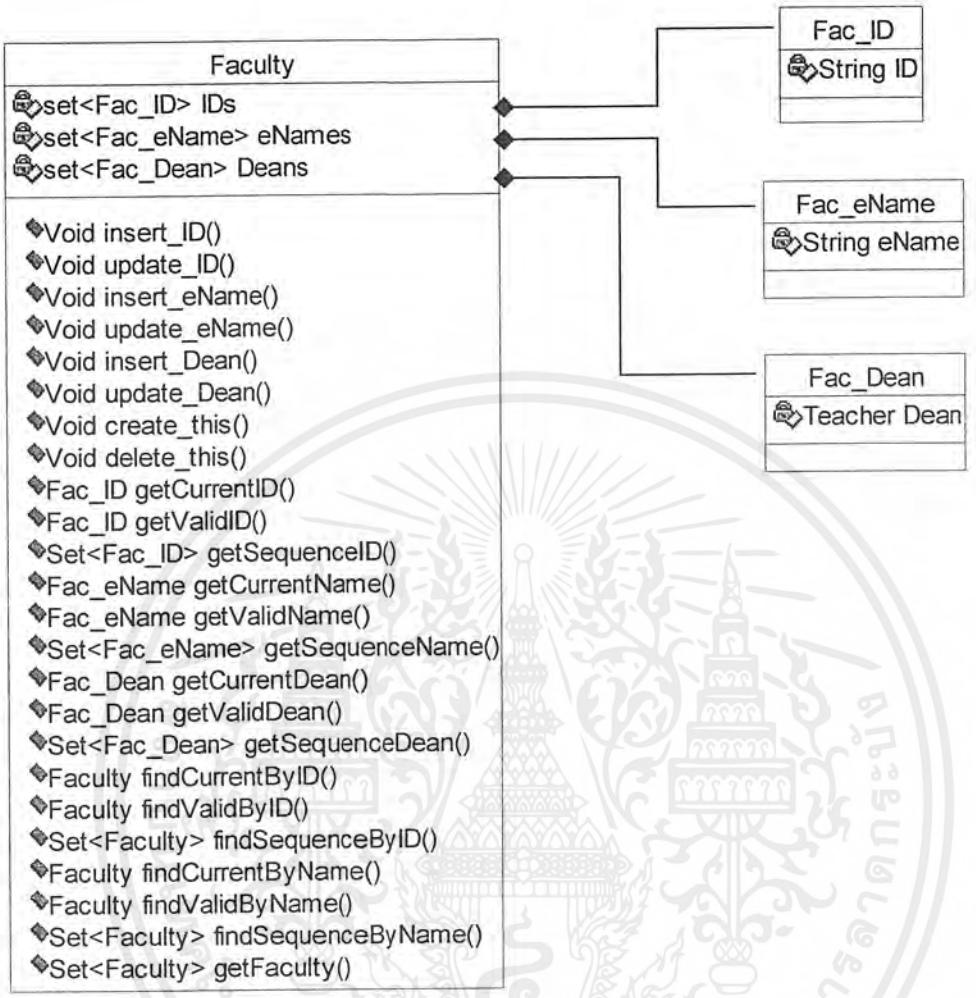
2.1 ส่วนของคลาส TempObject



รูปที่ 5.3 (a) คลาสไดอะแกรมแสดงภาพรวมของระบบ
(b) คลาสไดอะแกรมของคลาส TempObject

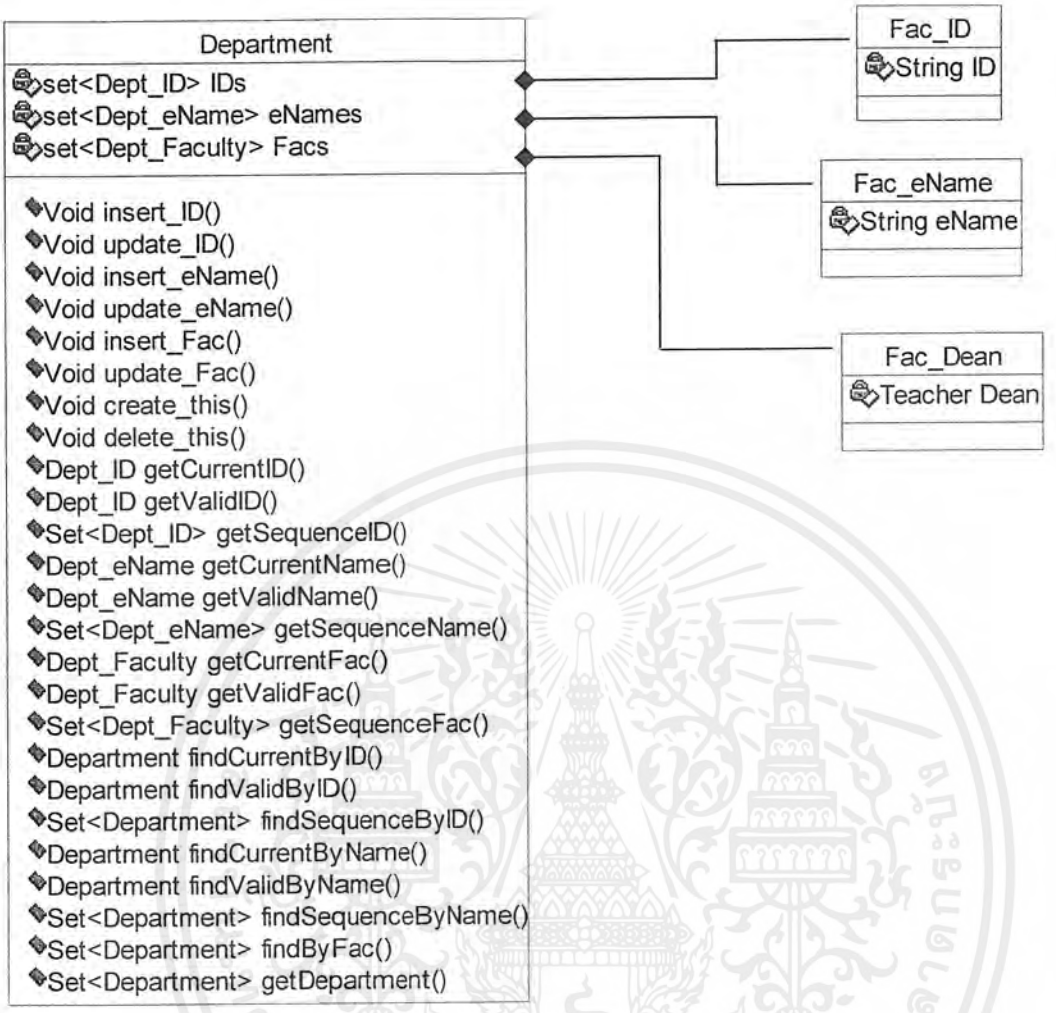
ทุกๆ คลาสจะสืบทอดคุณสมบัติ (inheritance) จากคลาส TempObject

2.2 ส่วนของคลาส Faculty



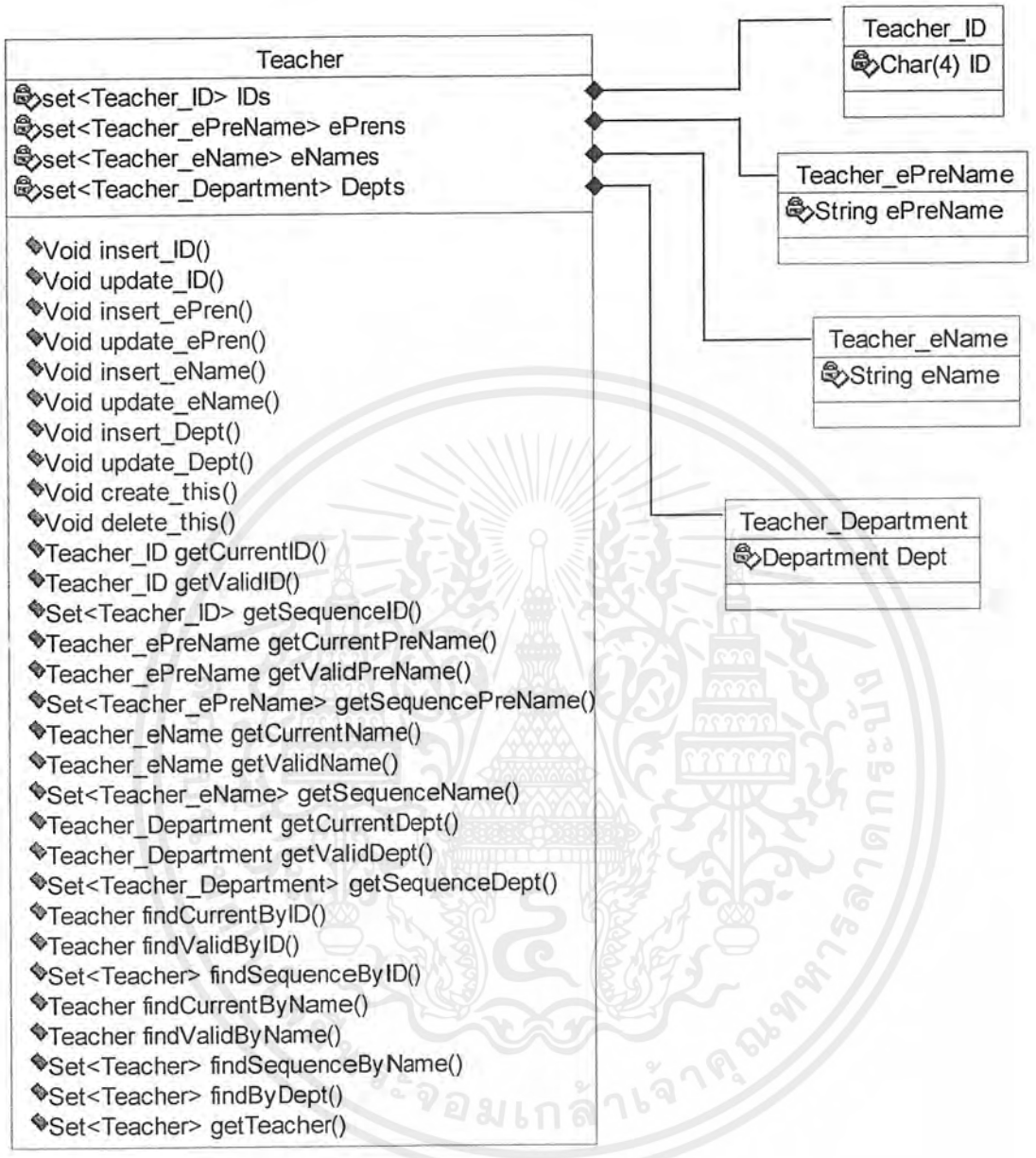
รูปที่ 5.4 คลาสไดอะแกรมของคลาส Faculty

2.3 ส่วนของคลาส Department



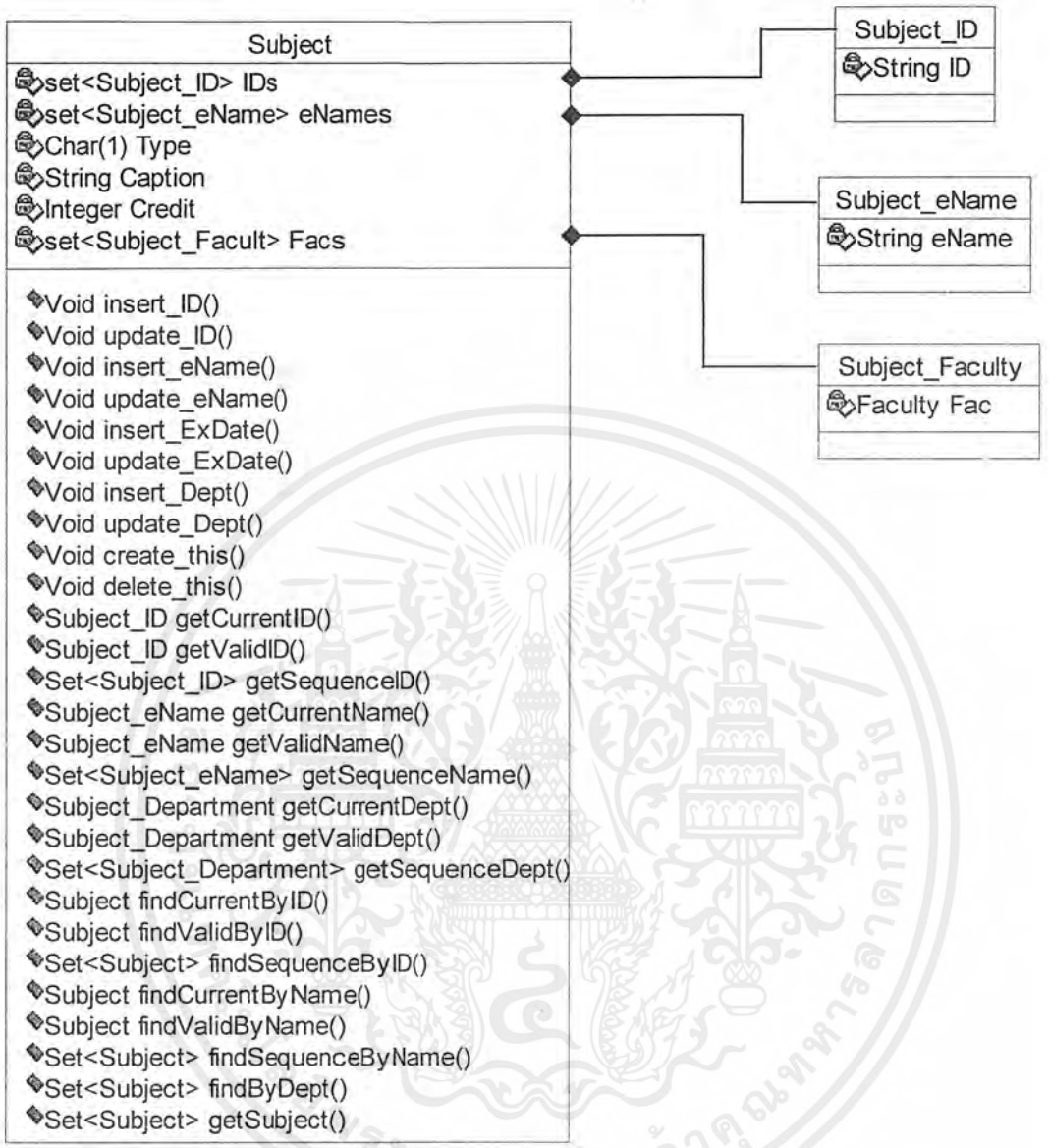
รูปที่ 5.5 คลาสไดอะแกรมของคลาส Department

2.4 ส่วนของคลาส Teacher



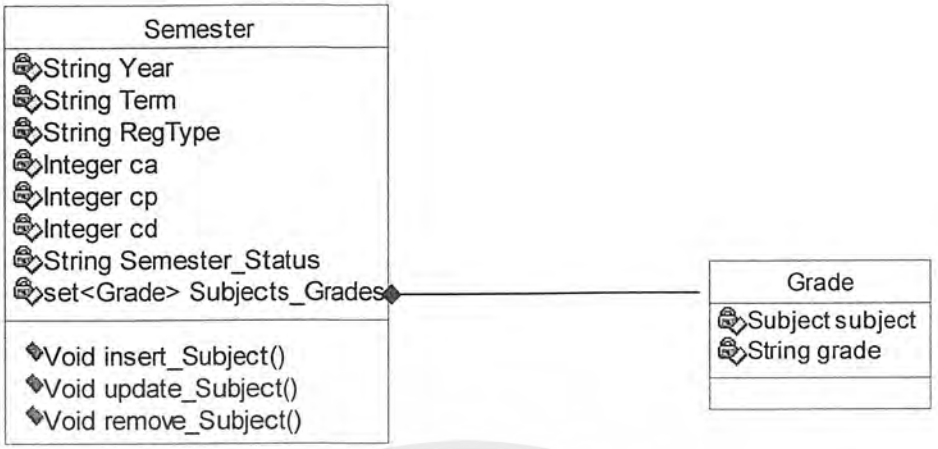
รูปที่ 5.6 คลาสไดอะแกรมของคลาส Teacher

2.5 ส่วนของคลาส Subject



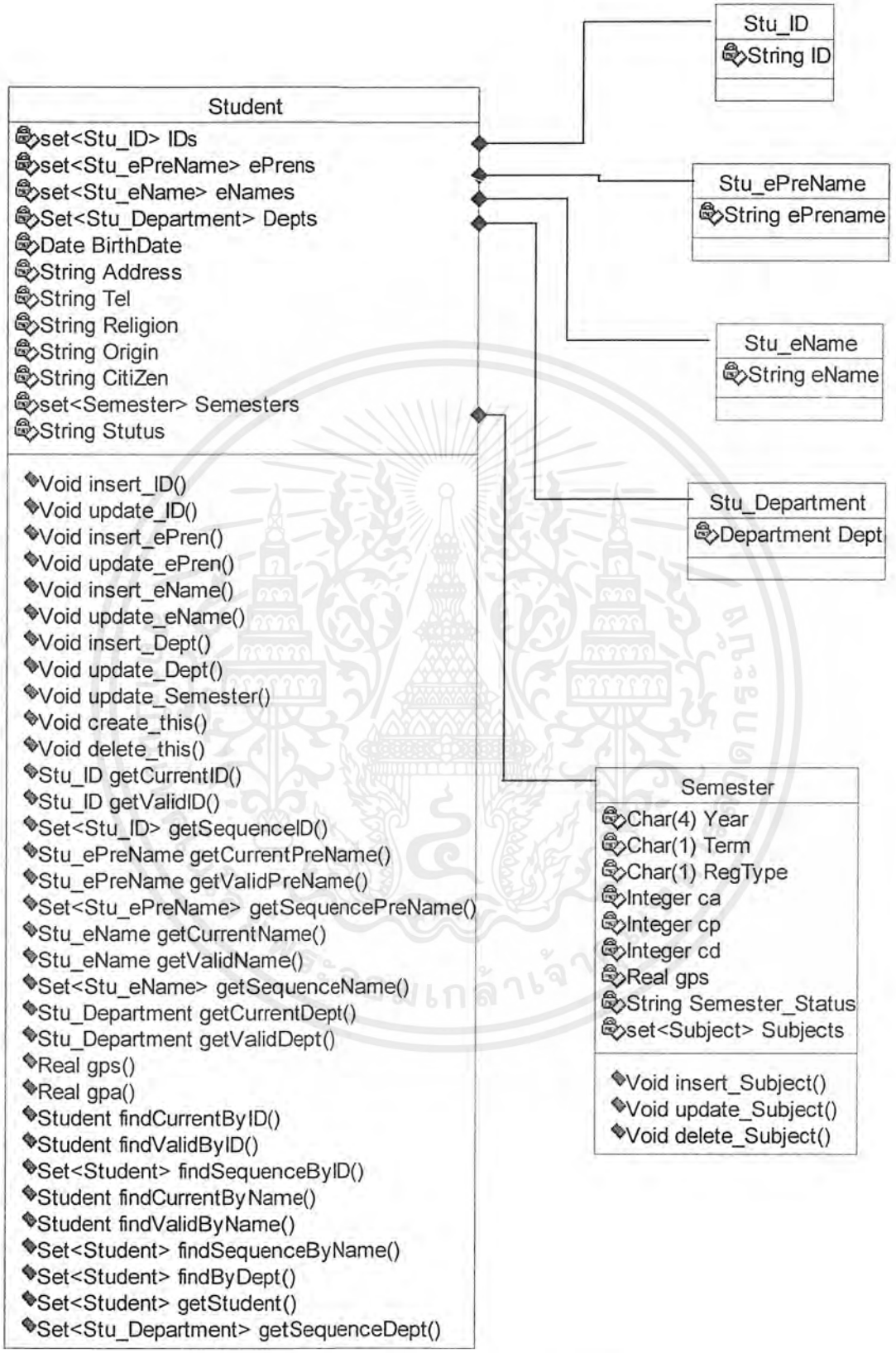
รูปที่ 5.7 คลาสไดอะแกรมของคลาส Subject

2.6 ส่วนของคลาส Semester



รูปที่ 5.8 คลาสไดอะแกรมของคลาส Semester

2.7 ส่วนของคลาส Student

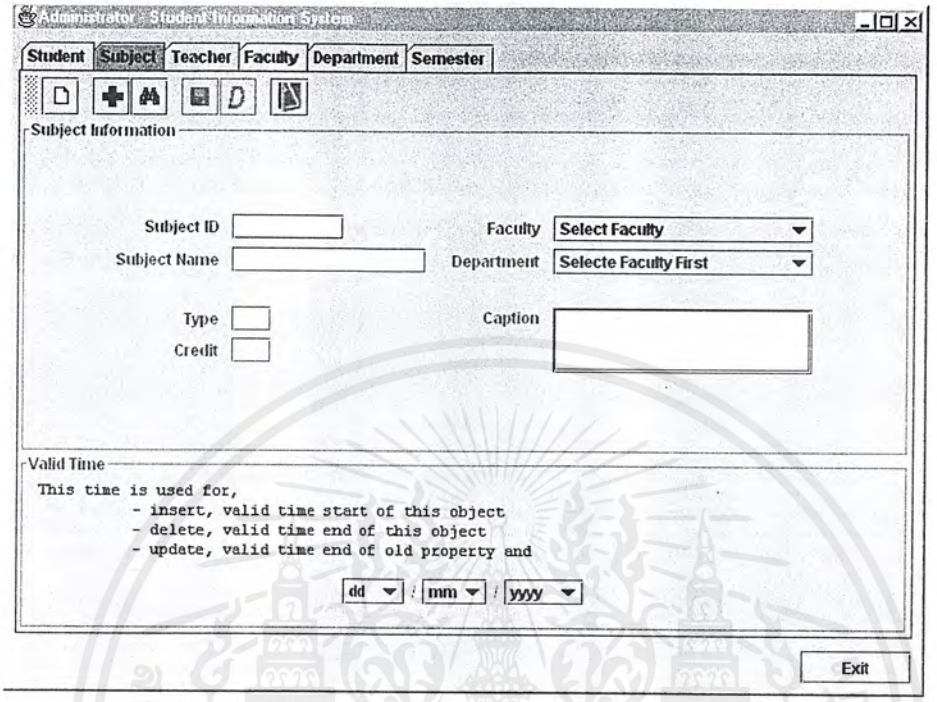


รูปที่ 5.9 คลาสไดอะแกรมของคลาส Student

5.4 ตัวอย่างโปรแกรม

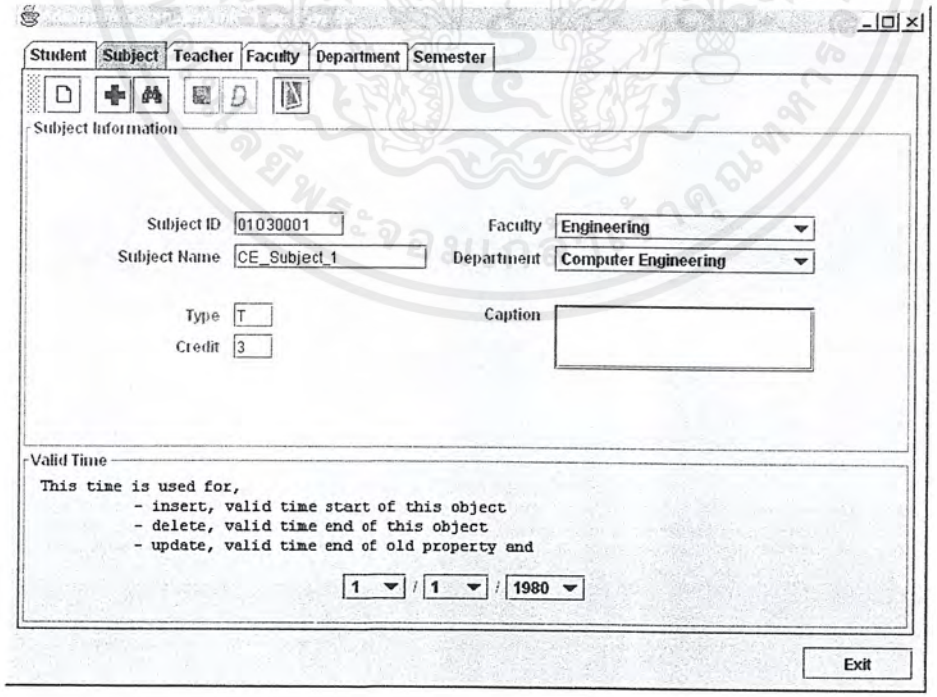
1. รายวิชาใหม่

1.1 เข้าการทำงานในส่วนของ Administrator เลือก Tab Subject



รูปที่ 5.10 แสดงหน้าจอหลักของโปรแกรม

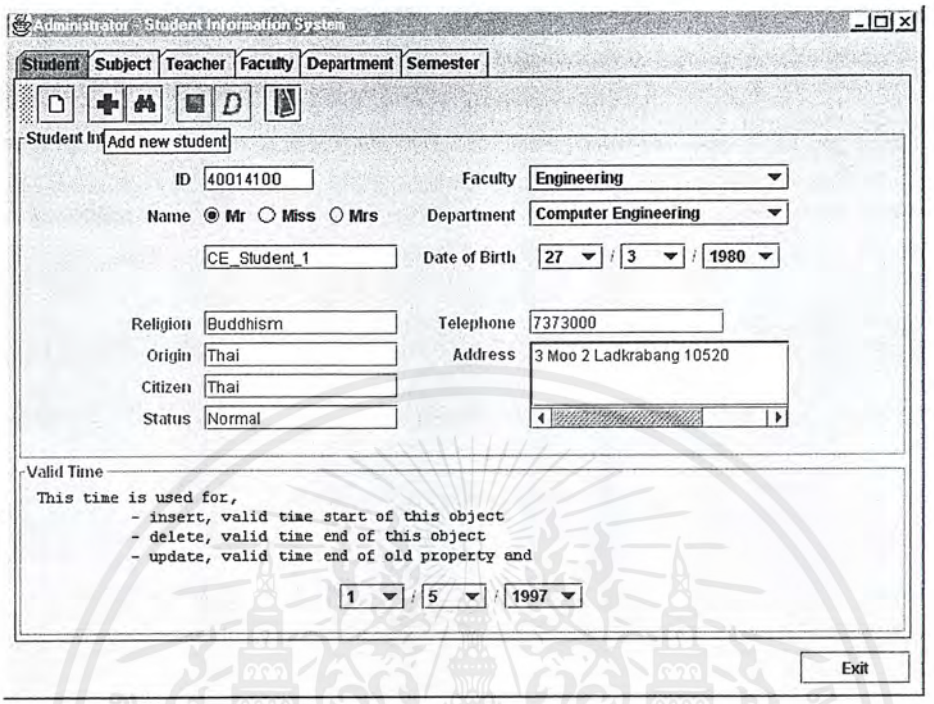
1.2 กรอกข้อมูลรายวิชา Click Add



รูปที่ 5.11 การเพิ่มรายวิชาใหม่

2. มีนักศึกษาเข้าใหม่

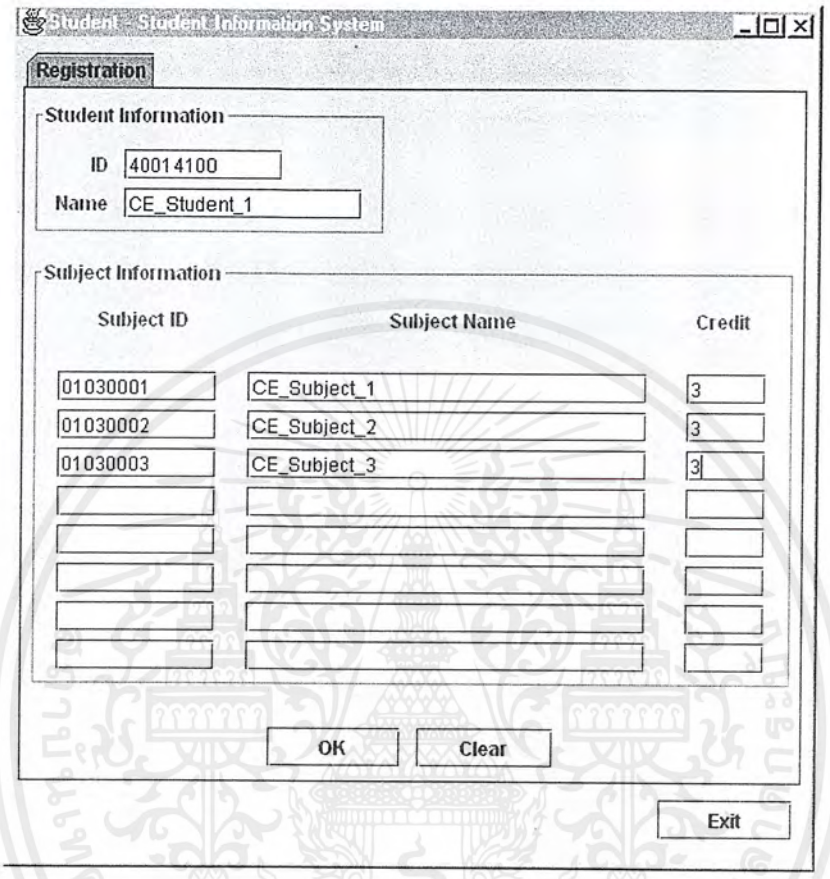
2.1 กรอกข้อมูลของนักศึกษา Click Add



รูปที่ 5.12 หน้าจอโปรแกรมแสดงการเพิ่มนักศึกษาใหม่

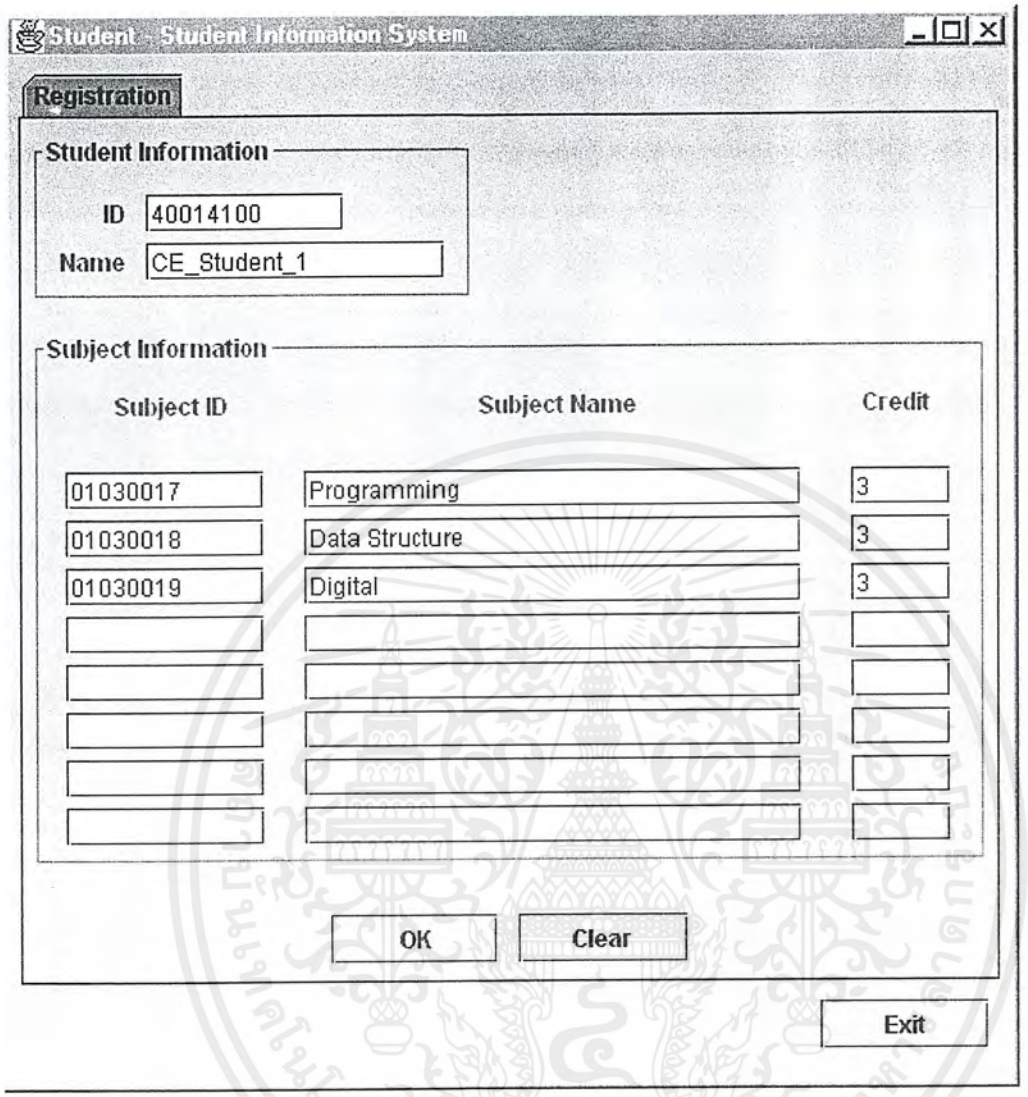
3. นักศึกษาลงทะเบียน

3.1 ลงทะเบียนเทอม 1/1997



รูปที่ 5.13 ลงทะเบียนปีการศึกษา 1/1997

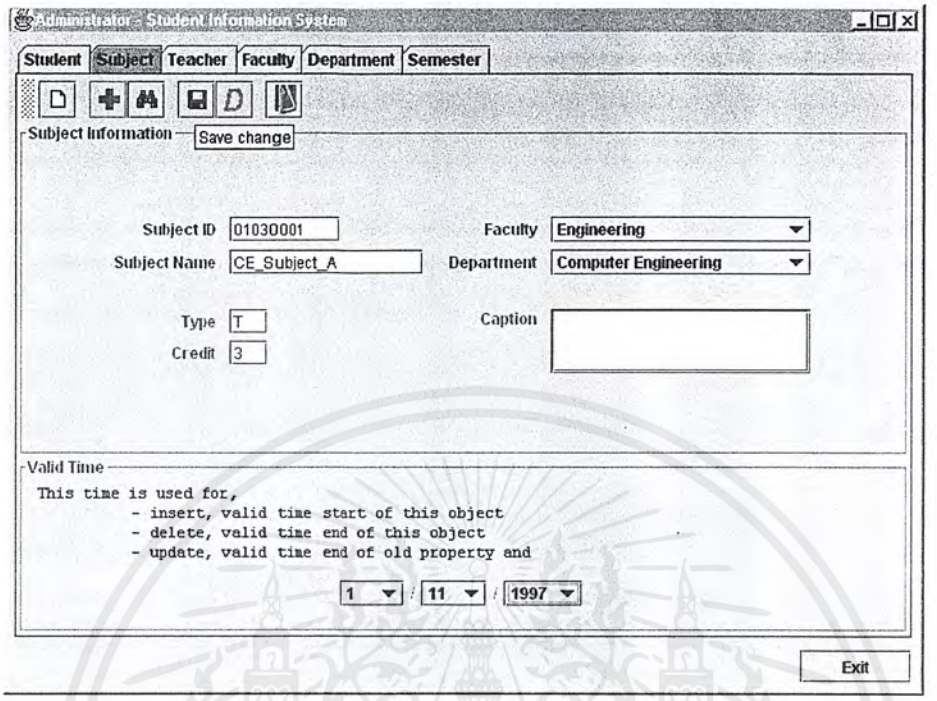
3.2 ลงทะเบียนเทอม 2/1997



รูปที่ 5.14 ลงทะเบียนปีการศึกษา 2/1997

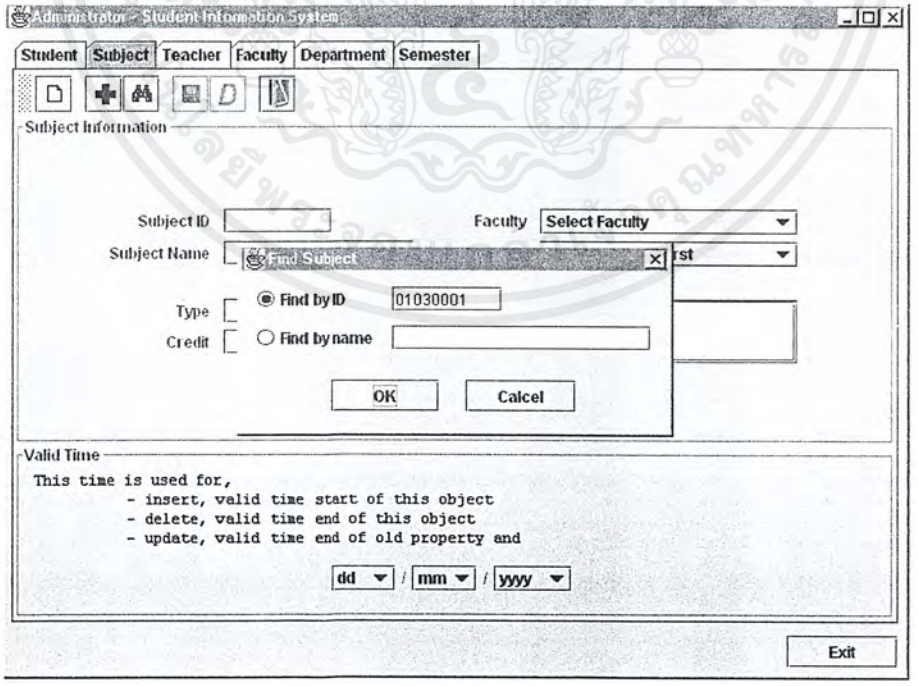
4. ชื่อวิชาเปลี่ยน

4.1 ค้นหาชื่อวิชาปัจจุบัน แล้วเปลี่ยนชื่อ แล้ว Click Save



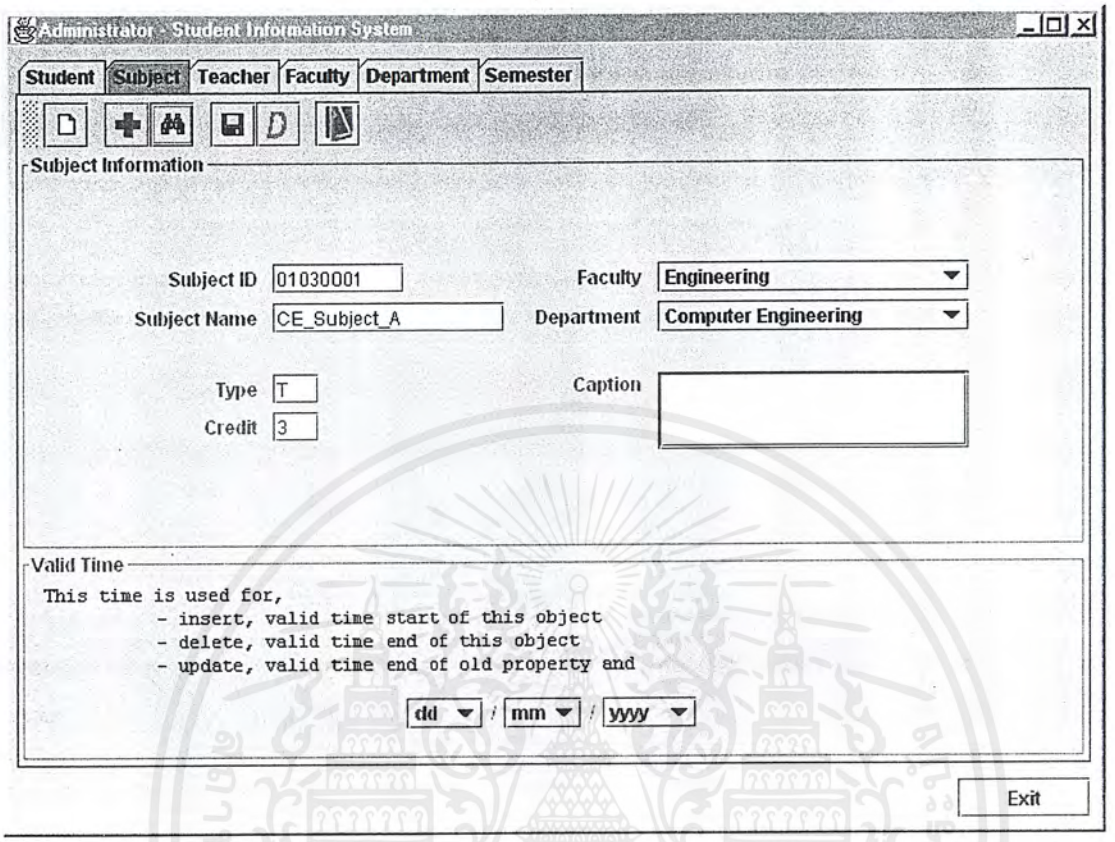
รูปที่ 5.15 แสดงการ เปลี่ยนชื่อวิชาจาก CE_Subject_1 เป็น CE_Subject_A

4.2 ลองค้นหาโดยใช้รหัสเดิม



รูปที่ 5.16 แสดงค้นหาโดยใช้รหัสวิชาเดิม

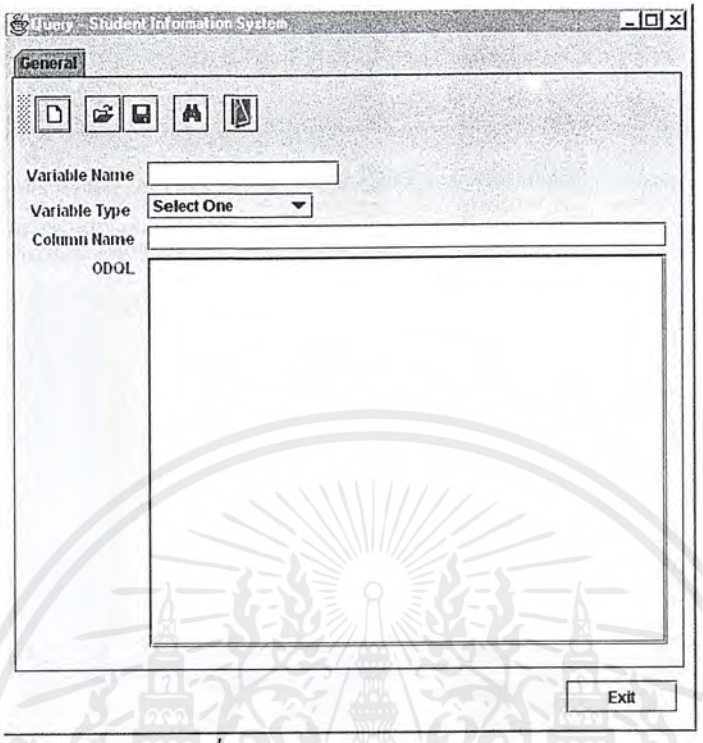
4.3 ผลลัพธ์



รูปที่ 5.17 แสดงผลลัพธ์จากการค้นหาข้อมูลรายวิชา

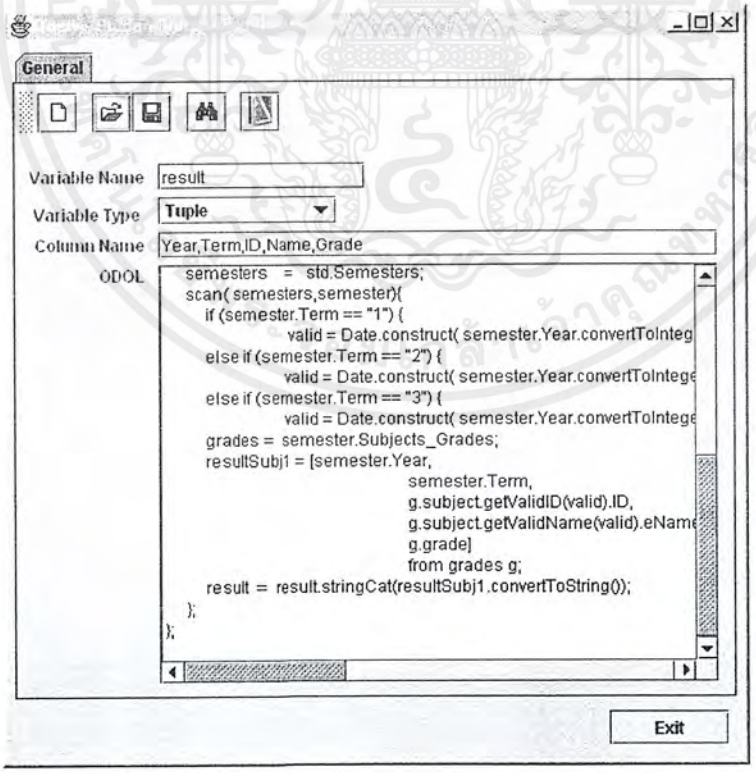
5. แสดงรายละเอียดวิชาที่เคยลงทะเบียน

5.1 เปิดหน้าจอ Query



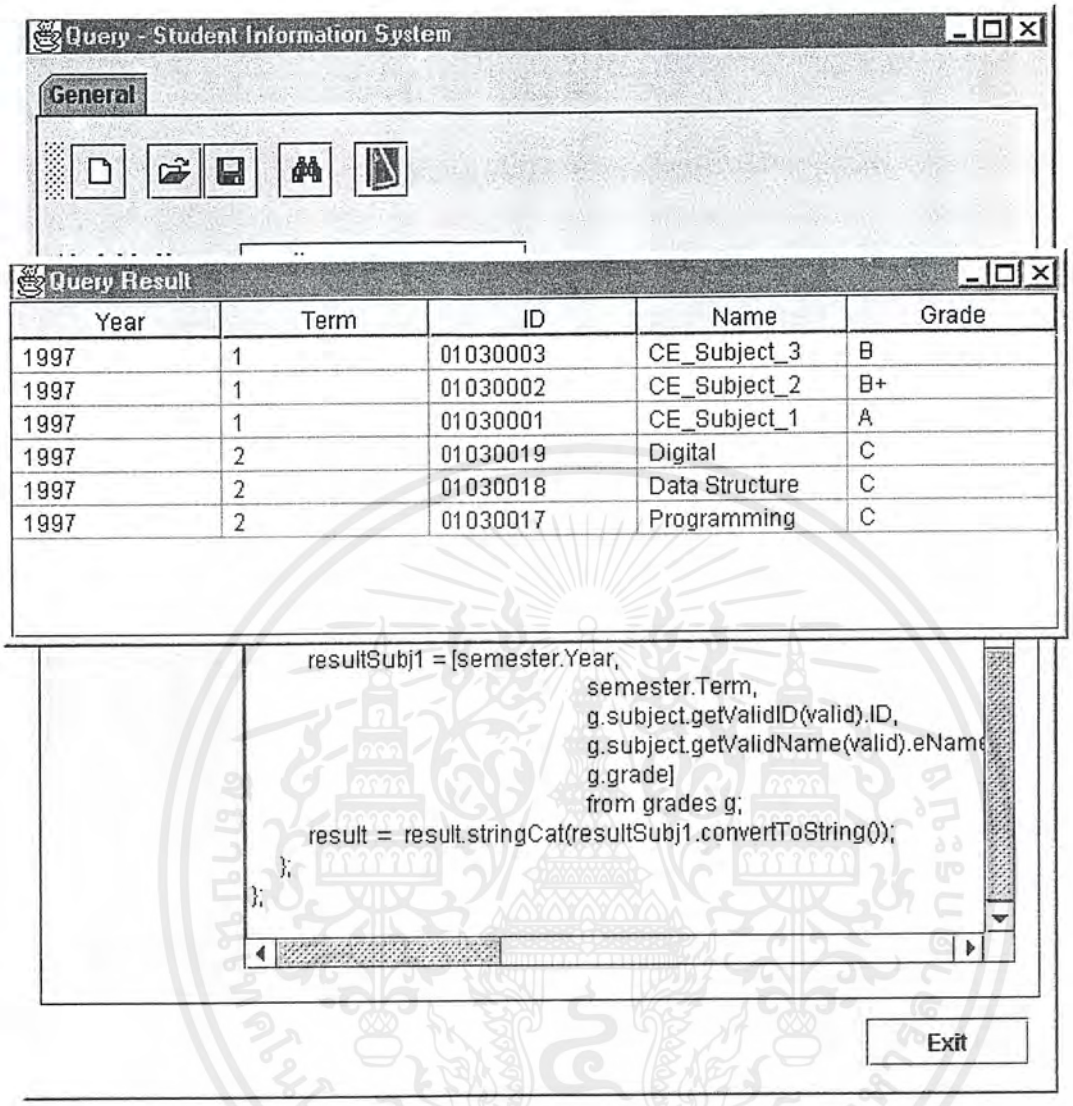
รูปที่ 5.18 หน้าจอของโปรแกรม Query

5.2 กรอกข้อมูล ให้ครบถ้วน



รูปที่ 5.19 กรอกข้อมูลเพื่อทำการค้นหา

5.3 Click Query



รูปที่ 5.20 แสดงผลลัพธ์จากการค้นหา

บทที่ 6
สรุปผลการวิจัยและข้อเสนอแนะ

6.1 สิ่งที่ทำได้

1. สามารถเพิ่มโครงสร้างข้อมูลเชิงเวลาให้กับฐานข้อมูลเชิงวัตถุได้
2. สร้าง Temporal Operation ขึ้นเพื่อการแก้ไขและค้นหาข้อมูลจากฐานข้อมูล ได้แก่ intersection, union, minus, meet, overlap

6.2 สิ่งที่ทำไม่ได้

1. Temporal Constraint เนื่องจากยังไม่มีข้อกำหนดที่แน่นอนในเรื่องนี้
2. Temporal Optimizer : OODBMS ไม่สามารถนำ temporal ที่เพิ่มเข้าไปมาใช้ในกระบวนการ optimizer ได้

6.3 ข้อดี

1. ด้วยคุณสมบัติของฐานข้อมูลเชิงวัตถุ ทำให้สามารถนำ Temporal Operation และ Basic Operation ตามลักษณะของข้อมูลไว้บนฐานข้อมูล ทำให้ผู้พัฒนาโปรแกรมไม่ต้องจัดการเรื่องเหล่านี้เอง สามารถเรียกใช้ได้เลย
2. ในการเลือกใช้ Root Class Model ทำให้สร้าง Temporal Object-Oriented Database ได้ง่ายและรวดเร็ว

6.4 ข้อเสีย

1. การทำ Timestamp ในระดับของออบเจ็กต์เมื่อใช้ Root Class Model ทำให้ Property ที่ต้องการติดตามการเปลี่ยนแปลงในเชิงเวลาต้องสืบทอดคุณสมบัติมาจาก Root Class ทำให้เกิดคลาสย่อยๆ ขึ้นเป็นจำนวนมาก
2. การ Query ทำได้ยาก เพราะภาษา ODQL ไม่ได้สนับสนุน temporal โดยตรง
3. ผู้พัฒนาโปรแกรมต้องจัดการกับ Temporal Constraint เอง เนื่องจากไม่ได้สร้างการจัดการในเรื่องนี้ไว้ที่ฐานข้อมูล

6.5 ข้อเสนอแนะ

ภาษา OQL ตามมาตรฐานของ ODMG จะต้อง Computation Complete โดยจะต้องมีตัวแปรมารับผล Query ที่ได้ซึ่งตามปกติจะเป็นตัวแปรที่เป็น Collection ถ้าหากข้อมูลที่ต้องการมาจากหลายออบเจ็กต์จะทำให้มีปัญหาในการประกาศตัวแปร แต่ภาษา ODQL ของ Jasmine มีข้อมูลชนิด Tuple จึงสามารถใช้ Collection ของ Tuple มารับผลการ Query ได้ แต่อย่างไรก็ตามใน J-API ไม่ได้มี Operation ที่ใช้จัดการกับข้อมูลชนิด Tuple ทำให้ต้องแปลงข้อมูลเป็นสตริงก่อนแล้วจึงนำข้อมูลออกมาได้ ทำให้ข้อมูลที่ได้มาสูญเสีย Semantic ของชนิดข้อมูลไป

ภาคผนวก ก
การติดตั้ง และเรียกใช้ Jasmine

1. การติดตั้ง Jasmine

1.1 ความต้องการของระบบ

- **Hardware**
  - คอมพิวเตอร์ที่มีหน่วยประมวลผล Pentium
  - RAM 32 MB (ควรเป็น 64 MB)
  - เนื้อที่ว่างใน harddisk อย่างน้อย 200 MB (ควรเป็น 300 MB)
- **Software**
  - ระบบปฏิบัติการ Windows NT 4.0
  - ถ้าต้องการ access remote jasmine database เครื่องจะต้องเชื่อมต่อกับเครือข่าย TCP/IP
  - Microsoft Visual C++ compiler version 4.0 ขึ้นไป เพื่อใช้ compile method ของ Jasmine database และเพื่อใช้เป็น Jasmine C API development tool

1.2 การติดตั้ง Jasmine

1. ใส่แผ่น Jasmine CD-ROM เข้าใน drive CD
2. ถ้า Jasmine Setup Wizard ไม่ปรากฏขึ้นมา ให้ double-click ที่ CD-ROM drive letter
3. ทำตามขั้นตอนที่ปรากฏบนจอภาพ เพื่อติดตั้ง Jasmine database

1.3 การ Uninstall Jasmine

1. stop Jasmine database
2. shutdown Weblink server
3. shutdown Jasmine HTTP Server และ WWW Server
4. เลือก Uninstalling Jasmine
5. ทำตามขั้นตอนบนจอภาพ เพื่อ uninstall Jasmine

2. การเรียกใช้ Jasmine

2.1 การสร้าง Store

1. start Jasmine Database
2. เข้าที่ command prompt
3. พิมพ์ **createStore** แล้วตามด้วยชื่อ store ที่ต้องการ (storeName) และชื่อไฟล์และไดเรกทอรี่ที่ต้องการไว้เก็บ store (fileName) เช่น store ชื่อ tempStore เก็บไว้ที่ C:/Jasmine/Strore

C:\>createStore tempStore C:/Jasmine/Store

หรือใส่ parameter อื่นเพิ่มตามต้องการ

2.4 การสร้าง Class Family

1. เข้าที่ command prompt
2. พิมพ์ **createCF** ตามด้วยชื่อ class family (CFName) และชื่อ store ที่ต้องการให้เก็บ class family นั้น เช่น ต้องการสร้าง class family ชื่อ tempCF ไว้ที่ tempStore

   C:\>createCF tempCF tempStore

   หรือใส่ parameter อื่นเพิ่มตามต้องการ

2.3 การสร้างคลาส

1. เข้าที่ command prompt
2. พิมพ์ CODQLIE เพื่อเข้าโปรแกรม ODQL Interpretor
3. ใช้คำสั่ง **defineClass** เพื่อสร้างคลาสพร้อมด้วย attribute และ method ตามต้องการ เช่น ต้องการคลาสชื่อ test โดยมี class-level attribute คือ A มีชนิดเป็น Integer และมี class-level method B ซึ่งมี return type เป็น Integer เช่นเดียวกัน จะสามารถเขียนได้ดังนี้

   defineClass test {
   Class:
   Integer A;
   Integer B();
   };

4. เรียกคำสั่ง **buildClass** ตามด้วยชื่อคลาสที่ต้องการ เช่น ต้องการ build class ชื่อ test

   buildClass test;

สำหรับโครงงานนี้จะใช้วิธีสร้างคลาสทั้งหมดไว้เป็น file แล้ว redirection เข้าไปที่ CODQLIE เพื่อความสะดวก ซึ่งจะได้ผลลัพธ์เหมือนกันทุกประการ

2.4 การสร้างเมธอด

1. เข้าที่ command prompt
2. พิมพ์ CODQLIE เพื่อเข้าโปรแกรม ODQL Interpretor
3. ใช้คำสั่ง **defineProcedure** เพื่อกำหนดรายละเอียดภายใน method ที่เราได้กำหนดไว้แล้วตอนสร้างคลาส เช่น method B ในคลาส test จะสามารถเขียนได้ดังนี้

   defineProcedure Integer test::class:B(){
   .....
   .....

```
};
```

4. เมื่อเขียนครบทุก method ในคลาสแล้วเรียกคำสั่ง compileProcedure ตามด้วยชื่อคลาส เพื่อคอมไพล์ method ที่เขียนไว้ เช่น ต้องการคอมไพล์ method ในคลาส test

```
compileProcedure test;
```

สำหรับโครงงานนี้จะใช้วิธีเขียนรายละเอียดของ method ทั้งหมดไว้เป็น file แล้ว redirection เข้าไปที่ CODQLIE เพื่อความสะดวก ซึ่งจะได้ผลลัพธ์เหมือนกันทุกประการ

2.5 การสร้างออบเจ็กต์

1. เข้าที่ command prompt
2. พิมพ์ CODQLIE เพื่อเข้าโปรแกรม ODQL Interpretor
3. ใช้ method **new()** เพื่อสร้าง instance ของคลาสที่ต้องการ เช่น ต้องการสร้าง instance ของคลาส test ที่ได้สร้างไว้แล้ว ก็จะเขียนได้ดังนี้

```
test t;
...
t = test.new();
```

ซึ่งสามารถสร้างเป็น file ไว้ แล้ว redirection เข้าไปที่ CODQLIE ได้เช่นเดียวกัน

ภาคผนวก ข

การเขียน Java ติดต่อกับ Jasmine

1. ความต้องการของระบบ

- Windows ที่ install Jasmine และ Jp แล้ว
- JDK version 1.1.5 หรือสูงกว่านั้น
- กำหนด NT Environment variable ดังนี้
  1. JP_HOME กำหนดเป็น directory ที่ install Jp ไว้ (สำหรับตัวอย่างนี้ จะลงไว้ที่ c:\Jp\)
  2. JAS_SYSTEM กำหนดเป็น directory ที่ install Jasmine ไว้

2. ตัวอย่างการติดต่อกับ Jasmine ด้วย Jp

1. กำหนดคลาส employee

1.1 start Jasmine Database

1.2 สร้าง Class Family ใหม่ชื่อ demoCF โดยสั่ง

```
C:\>createCF demoCF system
```

ที่ command prompt ของ Windows NT

1.3 กำหนดคลาส employee ไว้ใน demoCF โดยสั่ง

```
C:\>codqlie < Employee.def
C:\> codqlie < Employee.method
C:\> codqlie < Employee.data
```

(ไฟล์ Employee.def, Employee.method, Employee.data จะอยู่ที่ด้านท้ายของบทนี้)

2 สร้างและ compile คลาสภาษา Java

2.1 Stop Jasmine server

2.2 สร้างคลาสภาษา java : ใช้ jpcg tool สร้างคลาสภาษา Java จากคลาสของ Jasmine โดยสั่ง

```
C:\> c:\Jp\vin\jpcg demoCF c:\demoOutput
```

จะได้ไฟล์

```
c:\demoOutput\jp\jasmine\demoCF\Employee.java
c:\demoOutput\jp\jasmine\demoCF\demoCF_TC.java
```

2.3 compile java class โดยพิมพ์

```
C:\>javac c:\demoOutput\ jp\jasmine\demoCF\*.java
```

จะได้ไฟล์

```
c:\demoOutput\jp\jasmine\demoCF\Employee.class
```

c:\demoOutput\jp\jasmine\demoCF\demoCF_TC.class

2.4 start Jasmine server

2.5 เพิ่ม path demoOutput เข้าใน CLASSPATH โดยสั่ง

C:\>set PATH=.;C:\demoOutput;%CLASSPATH%

3 **พัฒนา application ด้วยภาษา Java** โดยใช้คลาสที่ generate ออกมาด้วย Jp เช่น สร้างไฟล์ JpTest.java ไว้ที่ demoOutput ซึ่งมีรายละเอียดดังนี้

```
import java.util.*;
import jp.jasmine.japi.*;
import jp.jasmine.rmi.*;
import jp.collection.*;
import jp.jasmine.demoCF.*;
public class jpTest {
public static void main(String args[])
        throws java.rmi.RemoteException, java.lang.Exception {
        try {
                Database db = new Database();
                db.startSession(); //start a database session
                db.startTransaction(); //start a database transaction
                //
                // Print names of all Employees
                //
                ODQLStatement query = new ODQLStatement(db);
                query.execute("Set<demoCF::Employee> allEmps;");
                query.execute("allEmps = demoCF::Employee from demoCF::Employee;");
                DBCollection dbc = (DBCollection) query.getVar("allEmp");
                Enumeration e = dbc.elements();
                while (e.hasMoreElements()) {
                        DBObject emp = (DBObject) e.nextElement();
                        System.out.println("oid = " + emp);
                        String name = (String) emp.getProperty("name");
                        System.out.println("name : " + name);
                        }
```

```
                //
                // Create a new Employee in Jasmine
                //
                Employee alex = new Employee(db); // create a Jasmine object
                alex.setId(1234);
                alex.setName("Alex Smith");
                alex.setDeptname("Software");
                BagOfString myDegrees = new BagOfString();
                myDegrees.add("Ph.D from Stanford");
                        myDegrees.add("M.S from MIT");
                myDegrees.add("B.S from Harvard");
                alex.setDegrees(myDegrees);
                System.out.println("alex.id = " + alex.getId());
                System.out.println("alex.name = " + alex.getName());
                System.out.println("alex.deptName = " + alex.getDeptname());
                Employee anyBody = Employee.findEmployeeById(db, 1234);
                System.out.println("anyBody.id = " + anyBody.getId());
                System.out.println("anyBody.name = " + anyBody.getName());
                System.out.println("anyBody.deptName = " + anyBody.getDeptname());
                db.endTransaction();
                db.endSession();
        }
        catch (DatabaseException e) {
                System.out.println("exception in jpTest: " + e.getMessage());
                e.printStackTrace();
        }
    }
    }
```

4 compile application

4.1 CLASSPATH จะต้องมี c:\jp\classes\jp.jar และ c:\demoOutput รวมอยู่ด้วย

4.2 Compile JpTest.java โดยสั่ง

```
C:\>javac jpTest.java
```

ซึ่งจะได้ไฟล์ jpTest.class ขึ้นมา

5 run application โดยสั่ง

C:\demoOutput>java jpTest

Content of Employee.def file

```
defineClass demoCF::Employee
description: "Employee class definition"
{
instance:
        systemCF::Integer id;
        systemCF::String name;
        systemCF::String deptName;
        systemCF::Date hireDate;
        demoCF::Employee manager;
        systemCF::Real salary;
        Bag<systemCF::String> degrees;
        Set<systemCF::Integer> numbers;
        mediaCF::MMTiffFile picture;
class:
        systemCF::Integer maxOvertimeHours;
        demoCF::Employee findEmployeeById(systemCF::Integer id);
        demoCF::Employee findEmployeeByName(systemCF::String name);
        Bag<demoCF::Employee> findEmployeesByDept(systemCF::String dept);
};
buildClass demoCF::Employee;
```

Content of Employee.methods file

```
defineProcedure demoCF::Employee demoCF::Employee::class:findEmployeeByName(systemCF::String
name)
{
$Bag<demoCF::Employee> ee;
$demoCF::Employee e;
$demoCF::Employee e2;
$ee = demoCF::Employee from demoCF::Employee where demoCF::Employee.name == name;
$if (ee.count() == NIL) {
$return (NIL);
};
$if (ee.count() == 0) {
$return (NIL);
};
$scan(ee, e) {
$e2 = e; break;
};
$return (e2);
};
defineProcedure demoCF::Employee demoCF::Employee::class:findEmployeeById(systemCF::Integer id)
{
$Bag<demoCF::Employee> ee;
$demoCF::Employee e;
$demoCF::Employee e2;
$ee = demoCF::Employee from demoCF::Employee where demoCF::Employee.id == id;
$if (ee.count() == NIL) {
$return (NIL);
};
$if (ee.count() == 0) {
$return (NIL);
};
$scan(ee, e) {
$e2 = e; break;
};
$return (e2);
```

```
};
defineProcedure Bag<demoCF::Employee> demoCF::Employee::class:findEmployeesByDept
(systemCF::String dept)
{
$Bag<demoCF::Employee> ee;
$ee = demoCF::Employee from demoCF::Employee where demoCF::Employee.deptName == dept;
$return (ee);
};
compileProcedure demoCF::Employee;
```

Content of Employee.data file

```
Bag<systemCF::String> d1, d2, d3, d4;
Set<systemCF::Integer> ss, ss2;
demoCF::Employee mgr;
demoCF::Employee e;
d1 = Bag { "B.S." };
d2 = Bag { "M.S.", "B.S." };
d3 = Bag { "Ph.D.", "M.S.", "B.S." };
d4 = Bag { "M.B.A", "Ph.D.", "M.S.", "B.S." };
ss = Set { 22, 2222, 222222, 33, 3333, 333333 };
ss2 = Set { 22, 2222, 222222, 33, 3333, 333333, 444, 444444 };
mgr = demoCF::Employee.new(id := 99011,
name := "John Rafter",
deptName := "Software",
hireDate := Date.construct(1990, 3, 12),
manager := NIL,
salary := 65000.00,
degrees := d1,
numbers := ss);
e = demoCF::Employee.new(id := 99022,
name := "Donald Knuth",
deptName := "Software",
hireDate := Date.construct(1967, 5, 16),
```

```
    manager := mgr,
    salary := 99000.00,
    degrees := d3,
    numbers := ss);
    e = demoCF::Employee.new(id := 99034,
    name := "Mary Smith",
    deptName := "Sales",
    hireDate := Date.construct(1990, 2, 2),
    manager := mgr,
    salary := 45000.00,
    degrees := d2,
    numbers := ss);
    e = demoCF::Employee.new(id := 99034,
    name := "Julie Madden",
    deptName := "Human Resources",
    hireDate := Date.construct(1989, 6, 20),
    manager := mgr,
    salary := 75000.00,
    degrees := d2,
    numbers := ss);
    e = demoCF::Employee.new(id := 99088,
    name := "David Foster",
    deptName := "Hardware",
    hireDate := Date.construct(1979, 6, 20),
    manager := mgr,
    salary := 66000.00,
    degrees := d4,
    numbers := ss2);
```

บรรณานุกรม

[1] Setrag Khoshafian, Surapol Dasananda, Norayr Minassian (1999) : *"The Jasmine object database : multimedia applications for the web"*, CA: Morgan Kaufmann Publishers, San Francisco, 1999.

[2] Peter Rob (1995) : *"Database systems : design, implementation, and management"* Denvers, MA Boyd&Fraser, 1995.

[3] Andreas Steiner, Moira C. Norrie (1997) : *"Implementing Temporal Databases in Object-Oriented System"* Zurich, Switzerland: Institute of Informations Systems, 1997.

[4] Christian S. Jensen, Richard T. Snodgrass (1997) : *"Temporal Database Management"* , TimeCenter, 1997

[5] ไพศาล ขวัญเมือง, ภราดร สมัครพันธ์ (2541) : *"การพัฒนาระบบฐานข้อมูลเชิงวัตถุ"*, ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง, 2541.

[6] วีระศักดิ์ ซึ่งถาวร,ดร. (2541) : *"Fundamental of JAVA Programming Volume 1"*, Sum Publishing,กรุงเทพฯ, 2541

[7] วีระศักดิ์ ซึ่งถาวร,ดร. (2543) : *"Fundamental of JAVA Programming Volume 2"*, ซีเอ็ดยูเคชั่น, กรุงเทพฯ, 2543