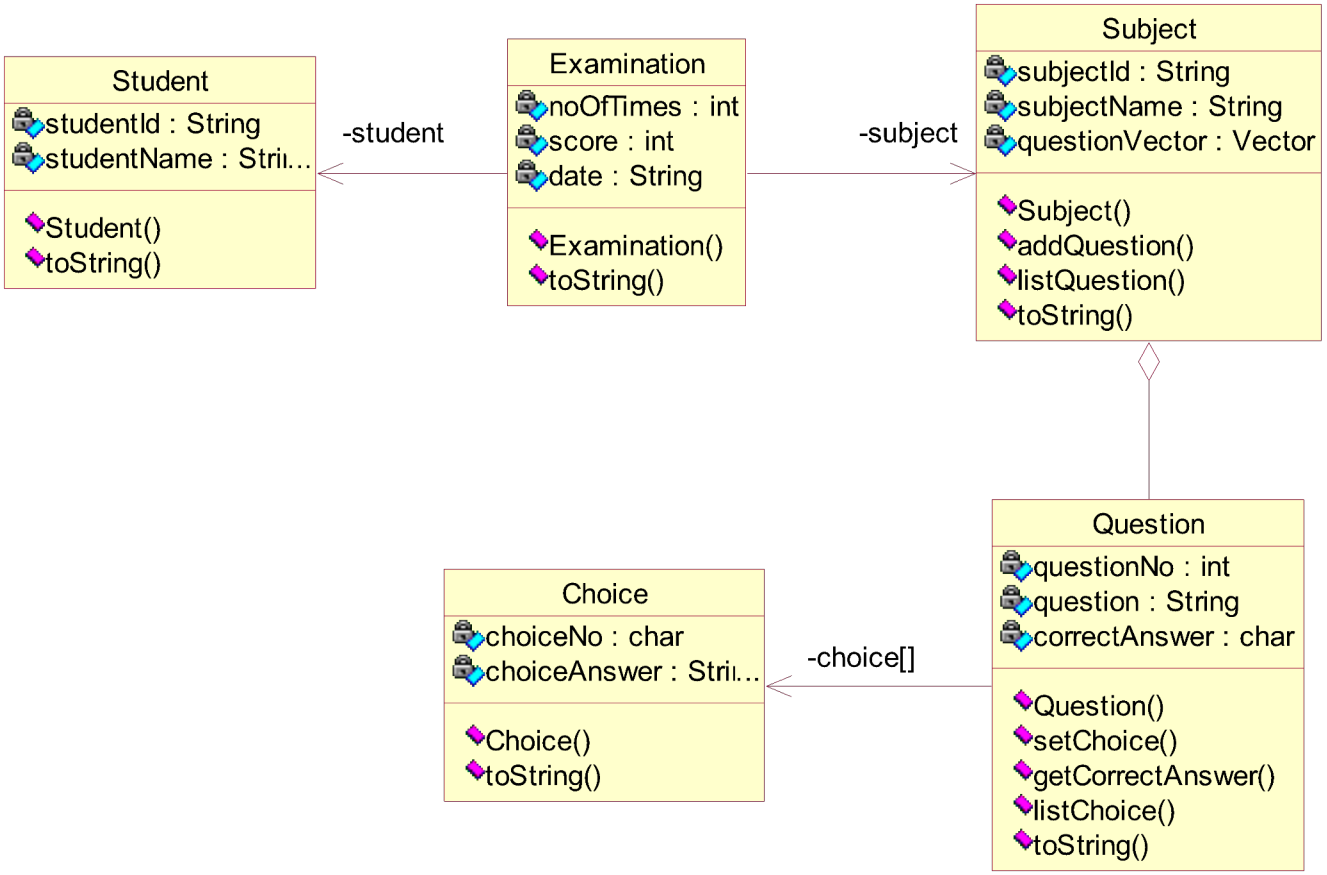

# Java 2

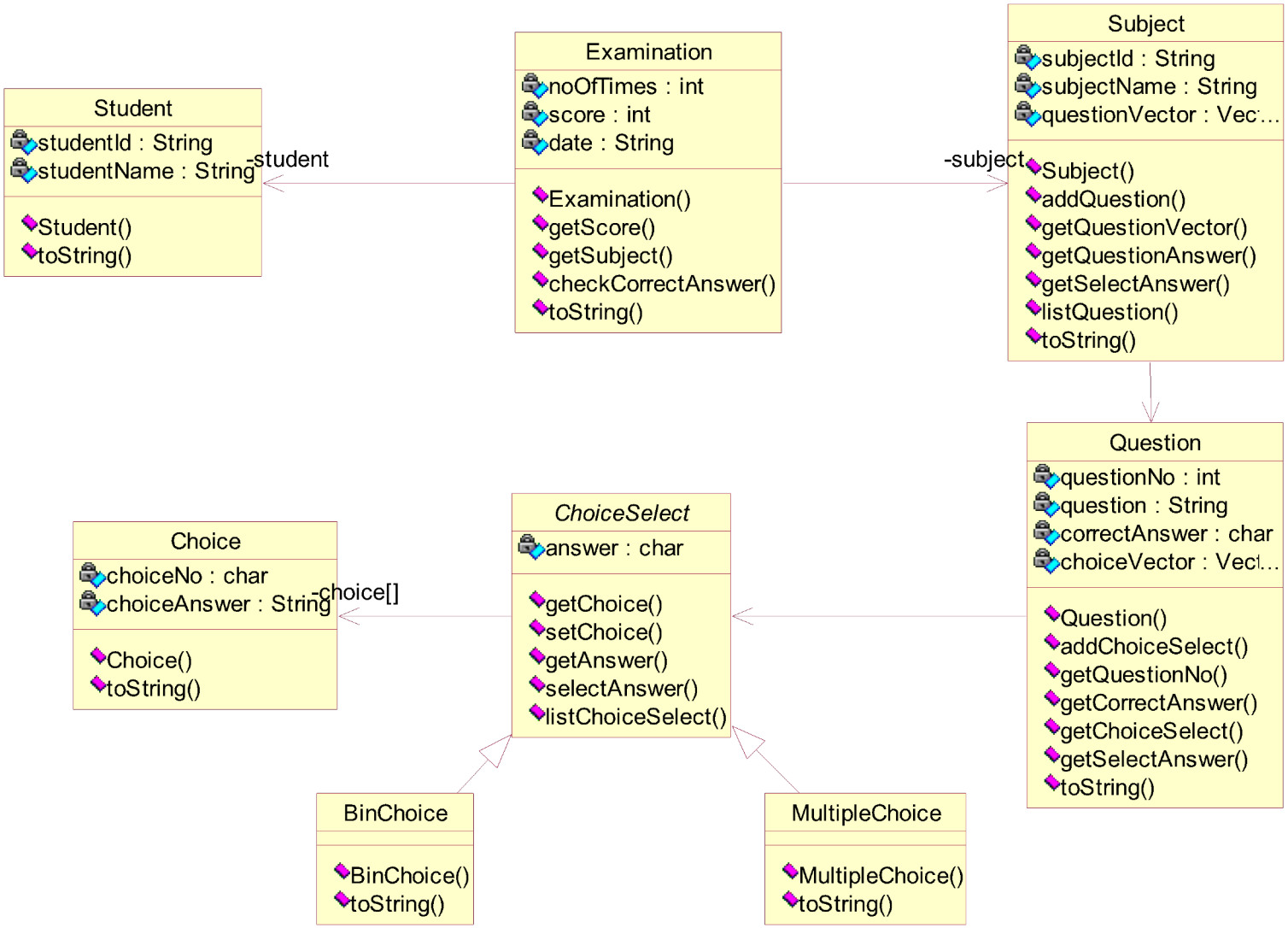
# Introduction to Design Patterns

By Assoc. Prof. Rangsit Sirirangsi

# OnlineExam

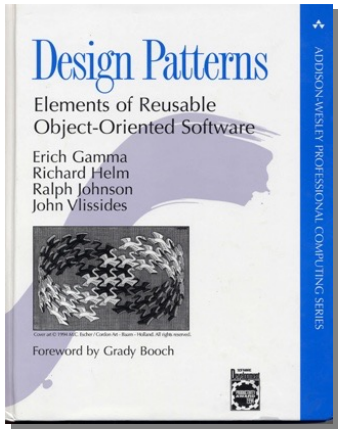

Student
studentId : String
studentName : Stri...
Student()
toString()
-student
Examination
noOfTimes : int
score : int
date : String
Examination()
toString()
-subject
Subject
subjectId : String
subjectName : String
questionVector : Vector
Subject()
addQuestion()
listQuestion()
toString()
Question
questionNo : int
question : String
correctAnswer : char
Question()
setChoice()
getCorrectAnswer()
listChoice()
toString()
-choice[]
Choice
choiceNo : char
choiceAnswer : Stri...
Choice()
toString()

# OnlineExam Modify

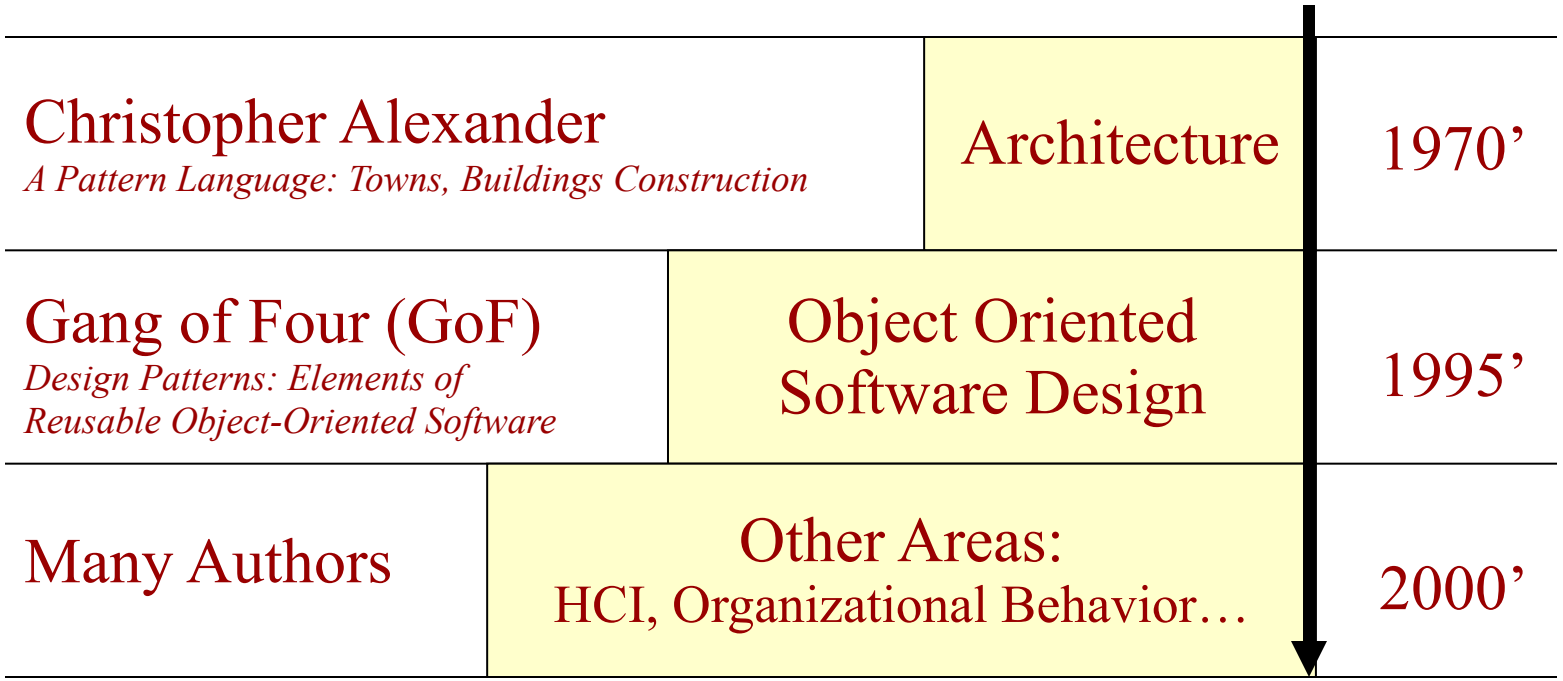


# Why Design Patterns ?


- นำแนวทางการแก้ไขปัญหาที่ทำงานอย่างได้ผลในอดีตกลับมาใช้ใหม่
- ช่วยในการออกแบบให้มีความยืดหยุ่นและสะดวกต่อการนำกลับไปใช้ใหม่
- ช่วยให้ Design Solutions ประสบความสำเร็จสำหรับระบบใหม่ที่นักพัฒนาสามารถทำงานได้อย่างแท้จริง
- สามารถระบุสถานการณ์ที่มีความเหมาะสมกับ Pattern ที่ใช้ได้
- ไม่ขึ้นอยู่กับภาษาโปรแกรมที่ใช้



# History of Design Patterns




- *"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution."*
- *Definition of a pattern: "A solution to a problem in a context."*

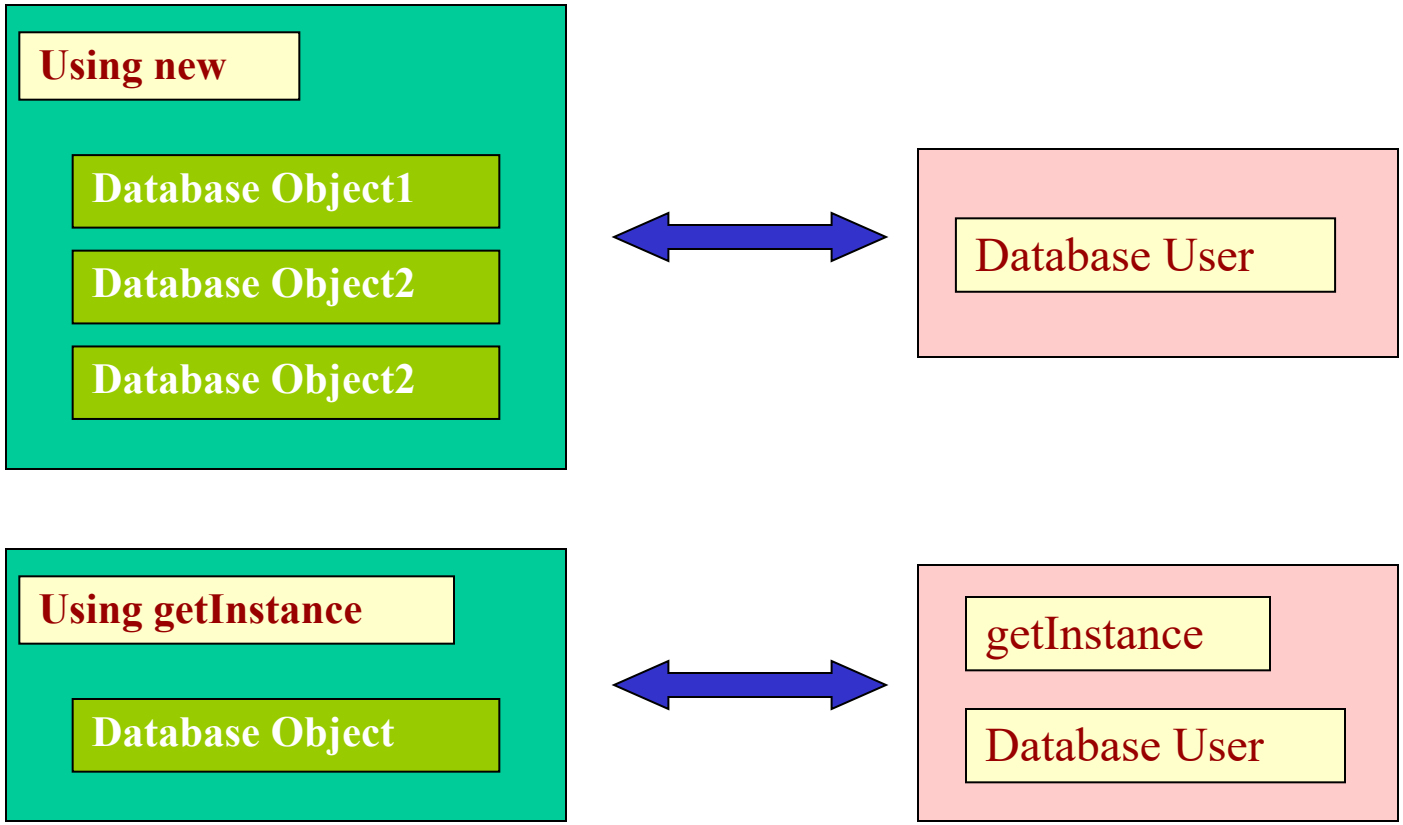
# Classification of patterns


- Design Patterns แบ่งออกเป็น 3 ประเภท ได้แก่
  - **Creational Design Patterns** เน้นไปที่การสร้างออปเจคและวิธีการที่อ้างอิงออปเจคนั้น ๆ เช่น Singleton, Abstract Factory, Factory Method…
  - **Structural Design Patterns** เกี่ยวข้องกับวิธีการที่คลาสและออปเจคต่าง ๆ มีการทำงานร่วมกัน เช่น Composite, Decorator, Façade, Adapter
  - **Behavioral Design Patterns** เน้นไปที่การรับและส่งแมสเสจระหว่างออปเจค รวมไปถึงการปฏิสัมพันธ์ระหว่างกัน เช่น Strategy, Command, Observer, Chain of Responsibilities…

# Singleton Pattern

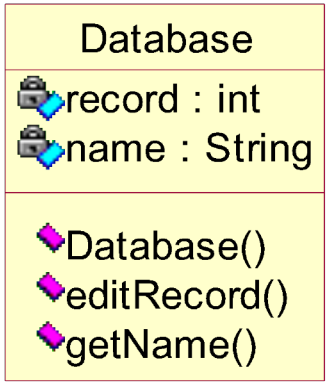

- เป็น Design Pattern ที่ใช้จำกัดจำนวนออปเจคที่ถูกสร้างในขณะที่โปรแกรมทำงาน ใช้ประโยชน์ในสถานการณ์ที่ทั้งระบบต้องมีออปเจคเพียงตัวเดียวเพื่อจะได้ไม่เกิดการทำงานซ้ำซ้อนกัน
- ตัวอย่างเช่น แม้จะมีหลาย ๆ ไฟล์ในการทำงาน แต่จะมีเพียงไฟล์ System เพียงไฟล์เดียวที่ใช้สำหรับเป็นศูนย์รวมการทำงานของไฟล์ทั้งหมด
- ใช้ในกรณีที่ต้องการให้ทุกๆ ส่วนของโปรแกรมใช้ออปเจค Singleton เพียงตัวเดียวร่วมกัน เพื่อให้ทุกส่วนทำงานสัมพันธ์กันนั่นเอง
- เพื่อให้แน่ใจว่าคลาสมีเพียงออปเจคเดียว และเป็นจุดศูนย์กลาง (Global Point) ในการเข้าถึงหรือใช้งานทั้งหมด
- Pattern แบบนี้ยังช่วยประหยัดพื้นที่เมมโมรี่ในการทำงานอีกด้วย

# Singleton Pattern Concepts



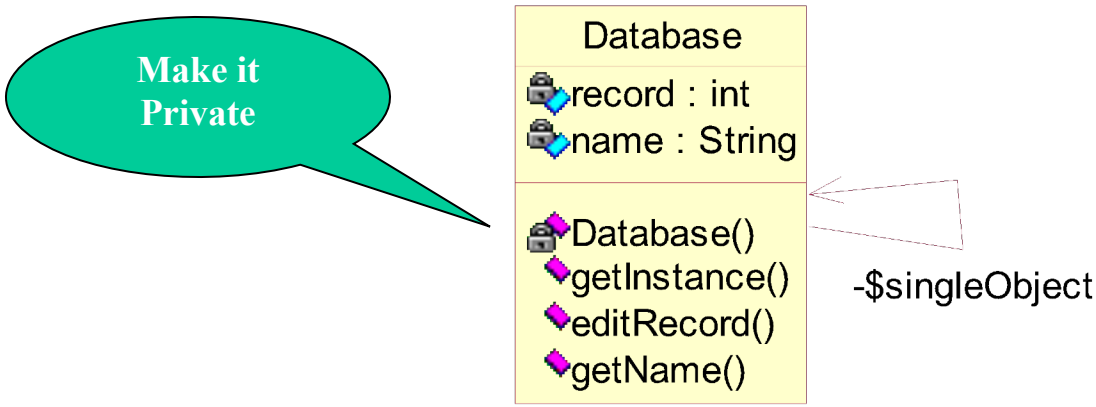
# Without Singleton

```
public class Database
{
  private int record;
  private String name;

  public Database(String n)
  {
    name = n;
    record = 0;
  }
  public void editRecord(String operation)
  {
    System.out.println("Performing a " + operation +
      " operation on record " + record +
      " in database " + name);
  }
  public String getName()
  {
    return name;
  }
}
```




```
Database one = new Database("One");

Database two = new Database("two");

Database three = new Database("three");
```

# Singleton: Structure



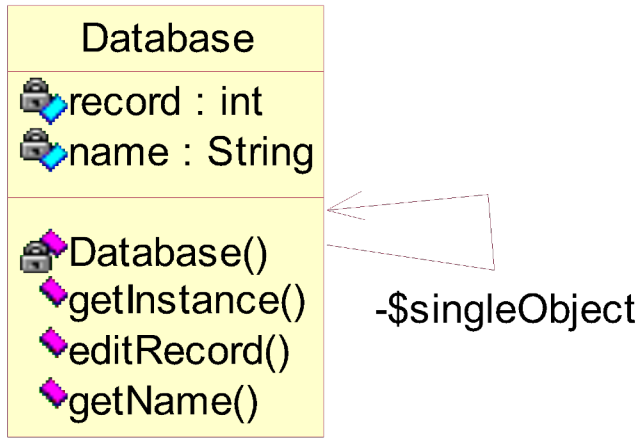

- ในทางปฏิบัติแล้วคลาสจะถูกปกปิดการเรียกใช้จากคลาสอื่น ๆ โดยการกำหนดให้ constructor เป็น private
- จากนั้นสร้างเมธอด getInstance() ซึ่งทำหน้าที่สร้างออปเจค แต่ต้องมีการตรวจสอบการสร้างออปเจคก่อน และคืนค่าออปเจคเดียวทุกครั้งที่มีการร้องขอ

# With Singleton

```
public class Database  {

  private static Database singleObject;
  private int record;
  private String name;

  private Database(String n)  {
    name = n;
    record = 0;
  }

  public static Database getInstance(String n)   {
      if (singleObject == null){
        singleObject = new Database(n);
      }
      return singleObject;
  }

  public void editRecord(String operation)  {
    System.out.println("Performing a " + operation + " on record " + record +
      " in database " + name);
  }
  public String getName()  {    return name;  }
}
```

Class variable

Make it Private



# With Singleton


```
public class Database  {

  private static Database singleObject;
  private int record;
  private String name;

  ……
  public static synchronized Database getInstance(String n)   {
     if (singleObject == null)  {
        singleObject = new Database(n);
     }
     return singleObject;
  }
……
}
```


```
public class RunSingleton {
   public static void main(String[] args) {
    Database db = Database.getInstance("Java");
    System.out.println(db.getName());
   }
}
```

- ข้อควรระวังการใช้ Pattern แบบนี้ในโปรแกรมแบบ multi -threading คือ หลายส่วนของโปรแกรมอาจจะพยายามเรียกเมธอดให้สร้างออปเจคเป็นครั้งแรกในเวลาเดียวกัน อาจทำให้มีการสร้างออปเจคขึ้นมามากกว่าหนึ่งได้
- ป้องกันให้เพียง 1 Thread ที่ทำงานได้โดยใช้ synchronized

# Singleton Pattern

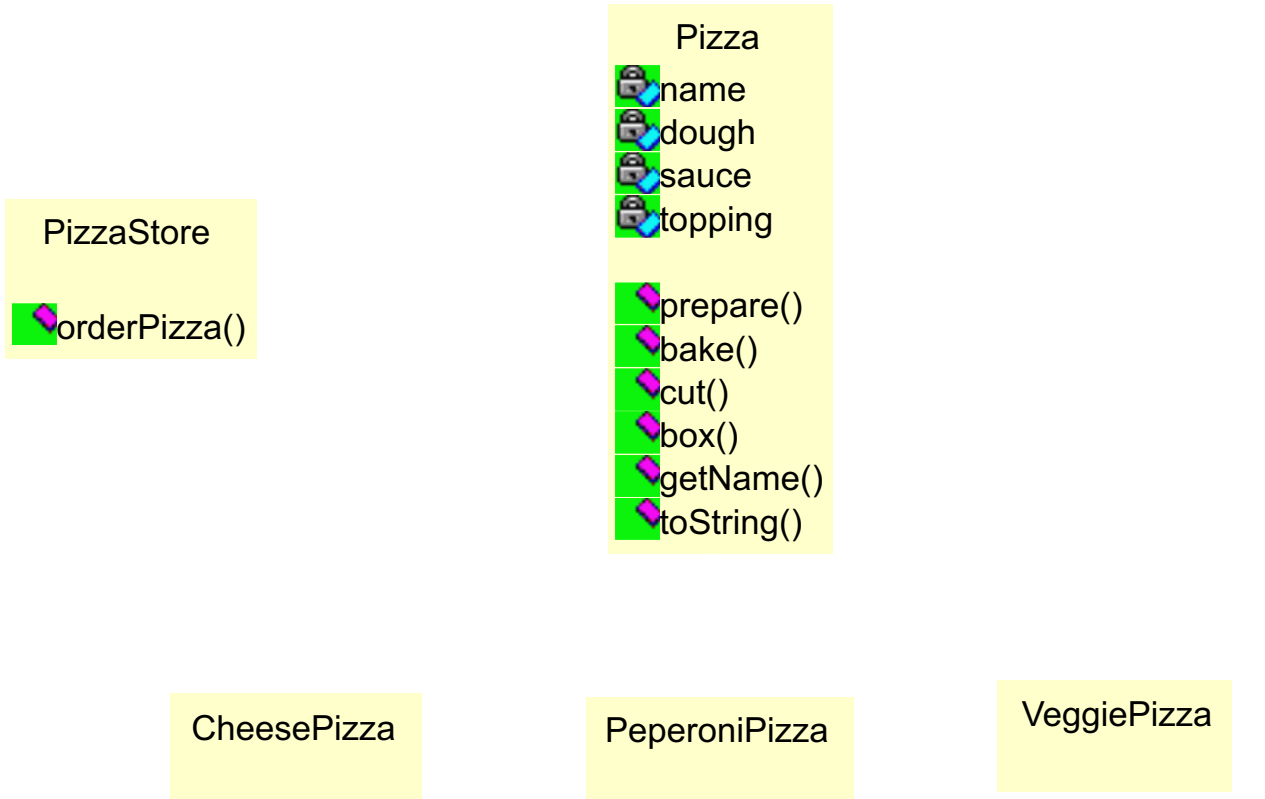

- ใช้ Singleton pattern เมื่อ
  - มีการสร้าง 1 ออปเจคจากคลาส และต้องสามารถเข้าถึงได้โดยไคลเอนต์จากจุดที่กำหนดไว้
  - ต้องการควบคุมการเข้าถึงการทำงานจากออปเจคหนึ่ง ๆ
    - เนื่องจากคลาสแบบ Singleton สร้างออปเจคเดี่ยว ดังนั้นจึงมีการควบคุมการเข้าถึงและใช้งานอย่างเข้มงวด

# Factory Pattern


- เป็น Creational Design Pattern แบบหนึ่งที่มีจุดประสงค์ในการสร้างออปเจค โดยไม่มีการแสดงรายละเอียดของการสร้าง
- Factory Pattern ถูกกำหนดให้เป็นคลาสหรือ interface ที่รับผิดชอบในการสร้างออปเจค แต่คลาสสืบทอดตัดสินใจว่าคลาสใดที่ถูกสร้าง โดยพิจารณาจากข้อมูลที่ได้รับ
- Factory patterns สามารถนำไปใช้ได้ในกรณีดังต่อไปนี้ :
  - เมื่อคลาสไม่ทราบว่าคลาสใดที่ต้องสร้างออปเจค
  - ต้องการให้คลาสสืบทอดสร้างออปเจค
  - ในโปรแกรมภาษาผู้ใช้สามารถใช้ Factory Pattern เมื่อต้องการสร้างออปเจคจากคลาสสืบทอด โดยขึ้นกับข้อมูลที่จัดให้

# Factory Pattern

- ระบบรับการสั่งซื้อ pizza 3 ชนิด ได้แก่ Cheese, Peperoni, และ Veggie
- Pizzas เหล่านี้จะแตกต่างกันที่ name, sauce, dough ที่ใช้และ topping



# Factory Pattern

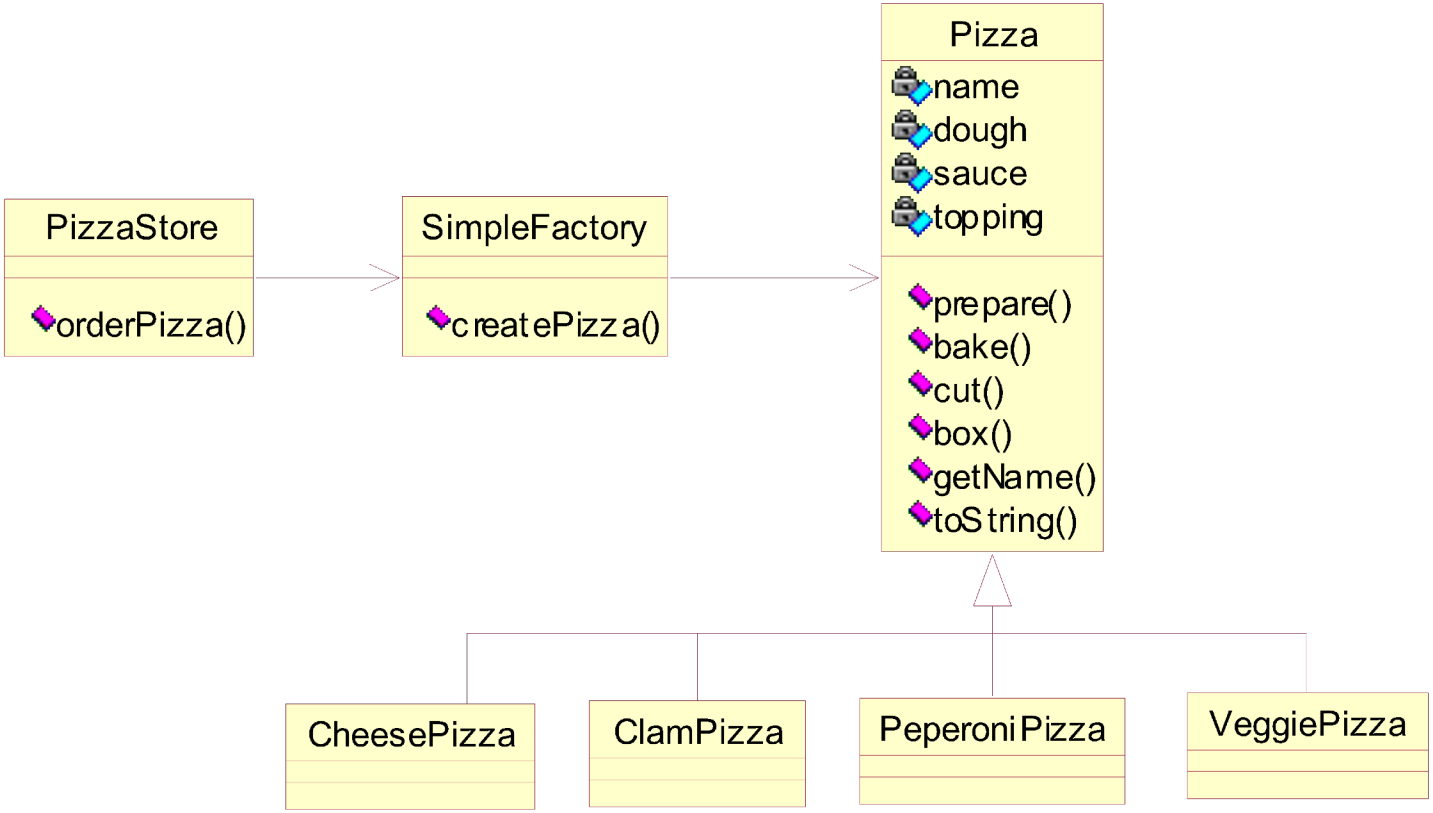

- ในกรณีนี้มีการสั่งซื้อพิซซาได้ 3 แบบ ได้แก่ : Cheese, Peperoni, และ Veggie
- ดังนั้นเพื่อให้ง่ายต่อการจัดการอาจรวบรวมการทำงานไว้ในเมธอดดังนี้

```
public Pizza orderPizza (String type)
{
    if (type.equals("Cheese"))
    {
      return new  CheesePizza();
    }
   else if (type.equals("Peperoni"))
   {
     return new PeperoniPizza();
    }
   else
   {
     return new VeggiePizza();
    }
}
```

# Factory Pattern


- ในกรณีที่ต้องการลบหรือเพิ่ม Pizza ชนิดใหม่เข้ามา การโปรแกรมในลักษณะดังกล่าวจำเป็นต้องมีการแก้ไขโค้ดค่อนข้างมาก
- ดังนั้นจึงอาจสร้างคลาส SimpleFactory เพื่อทำหน้าที่จัดการกับการเปลี่ยนแปลงของโค้ด ซึ่งจะส่งผลให้การออกแบบมีความยืดหยุ่นมากขึ้น
- การแก้ไขโค้ดอาจทำได้โดยการแยกโค้ดออกเป็นส่วน ๆ เช่น
  - ส่วนที่เป็น Client
  - ส่วนที่เป็น Factory ที่การตัดสินใจสร้างออปเจคขึ้นอยู่กับเงื่อนไขของข้อมูลที่กำหนด
  - ส่วนที่ใช้สำหรับการสร้าง Pizza ออปเจค

# Example: Revised System



# Factory Pattern


- การทำงานของ Pattern แบบนี้เริ่มต้นจากการสร้าง Pizza ให้อยู่ในรูปของคลาสแบบ abstract หรือ interface

```
public abstract class Pizza
{
  private String name;
  private String dough;
  private String sauce;
  private String topping;

  public void prepare()  {   }
  public void cut() {  }
  public void box() { }
  public String getNmae() { }
  public String toString()  { }
}
```

```
public class CheesePizza extends Pizza
{
  public CheesePizza()
  {
  }
  public String getName()
  {
    return “Cheese Pizza";
  }
}
```

- จากนั้นสร้างคลาสที่สืบทอดมาจากคลาส Pizza และ Override เมธอด getName() ที่คืนค่าชนิดของฐานข้อมูลที่ต้องการในรูป String

# Factory class


- คลาส SimpleFactory มีเมธอด createPizza() ที่ใช้สร้างออปเจค
- การตัดสินใจสร้างออปเจคขึ้นอยู่กับเงื่อนไขของข้อมูลที่กำหนดในตัวแปร type
- เมธอด createPizza() คืนค่า Pizza ในรูปของ interface หรือคลาสใดก็ตามที่อิมพลีเมนต์มาจาก interface นั้น ๆ

```
public class SimpleFactory
{
  private Pizza pizzaType;
  private String type;

  public SimpleFactory(Pizza p)  {
     this.pizzaType = p;
  }

  public String getName()     {
     return this.pizzaType.getName();
  }
}
```

# Factory Pattern


```
public abstract class Pizza
{
  private String name;
  private String sauce;
  private String topping;

  public void prepare()  {   }
  public void cut() { }
  public void box() { }
  public abstract String getName();

}
```

```
public class PeperoniPizza extends Pizza
{
  public PeperoniPizza()  { }
  public String getName()  {
    return “Peperoni Pizza";
  }
}
```

```
public class CheesePizza extends Pizza
{
  public CheesePizza()  { }
  public String getName()  {
    return “Cheese Pizza";
  }
}
```

```
public class VeggiePizza extends Pizza
{
  public VeggiePizza()  { }
  public String getName()  {
    return “Veggie Pizza";
  }
}
```

# Factory Pattern


```
public class SimpleFactory
{
  private Pizza pizzaType;
  private String type;

  public SimpleFactory(Pizza p)  {
    this.pizzaType = p;
  }

  public String getName()    {
     return this.pizzaType.getName();
  }
}
```

```
public class TestPizza
{
  public static void main(String args[])
  {
     SimpleFactory s = new SimpleFactory(new CheesePizza());
     System.out.println("You're ordering " + s.getName());
  }
}
```

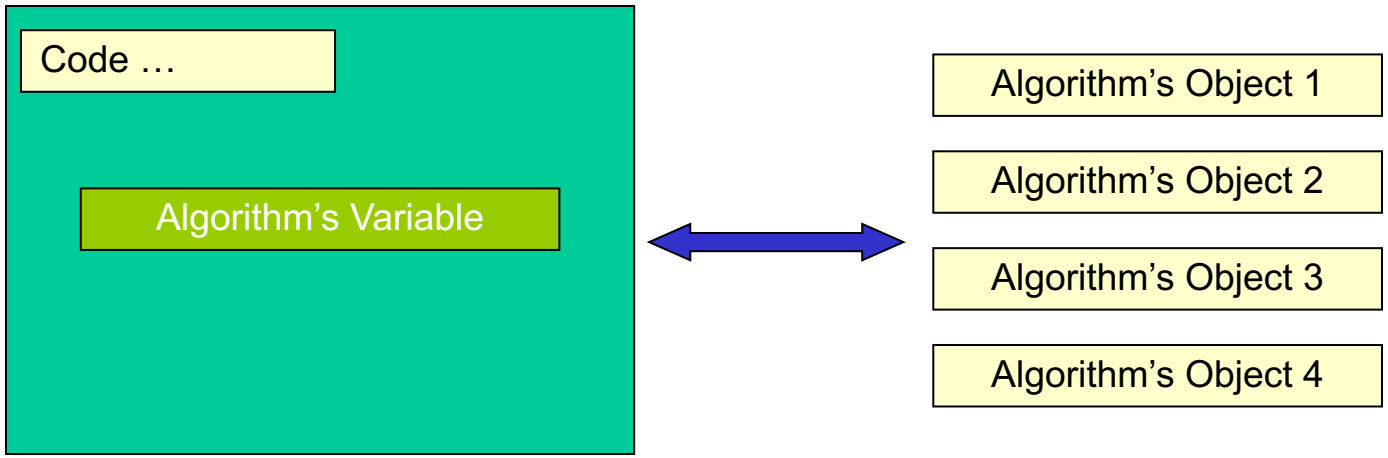
# Points about the Factory Pattern:


- factory คืนค่าออปเจคหนึ่งจากหลาย ๆ คลาสที่มี SuperClass เดียวกัน
- SuperClass สามารถอยู่ในรูปของคลาสแบบ abstract หรือ interface
- โดยปกติแล้วไคลเอนต์แจ้งข้อมูลที่ต้องการไปยัง factory จากนั้น factory ทำหน้าที่ตัดสินใจว่า Subclass ใดที่มีการสร้างออปเจค และคืนค่ากลับไปยังไคลเอนต์


- Client: คลาส TestPizza
- Factory Method: createPizza()
- Pizza: ออปเจคของคลาสที่อิมพลีเมนต์จาก interface

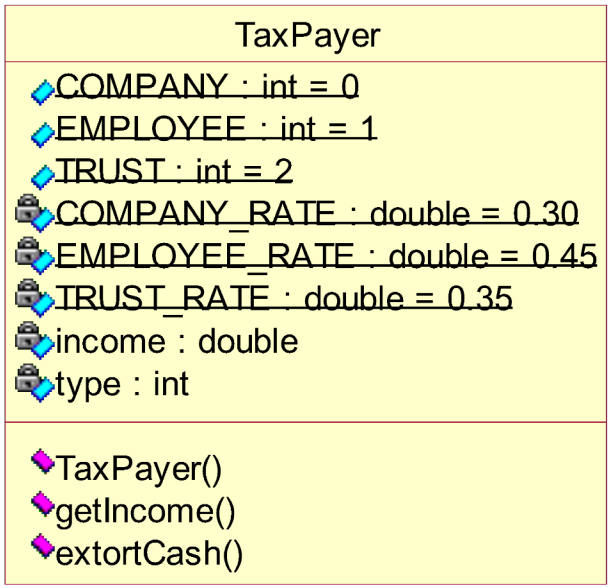
# Strategy: Applicability

- Strategy Pattern ใช้ในการแก้ปัญหาในกรณีที่มีการรวบรวมอัลกอลิทึมส์ต่าง ๆ ไว้เป็นกลุ่มเดียวกัน และสามารถใช้ทดแทนกันได้ โดยเปลี่ยนแปลงได้ตามการเรียกใช้ของไคลเอนต์



# Before Strategy Pattern

- ตัวอย่างเช่น คลาสสำหรับการคำนวณค่า tax สำหรับรายการต่าง ๆ ของคลาส TaxPayer โดยค่า tax ถูกกำหนดไว้ในรูปของ **int** ภายในคลาส
- ดังนั้นผู้ใช้ต้องตัดสินใจเลือกค่าคงที่ในรูปของ tax ที่เหมาะสมในการคำนวณเอง



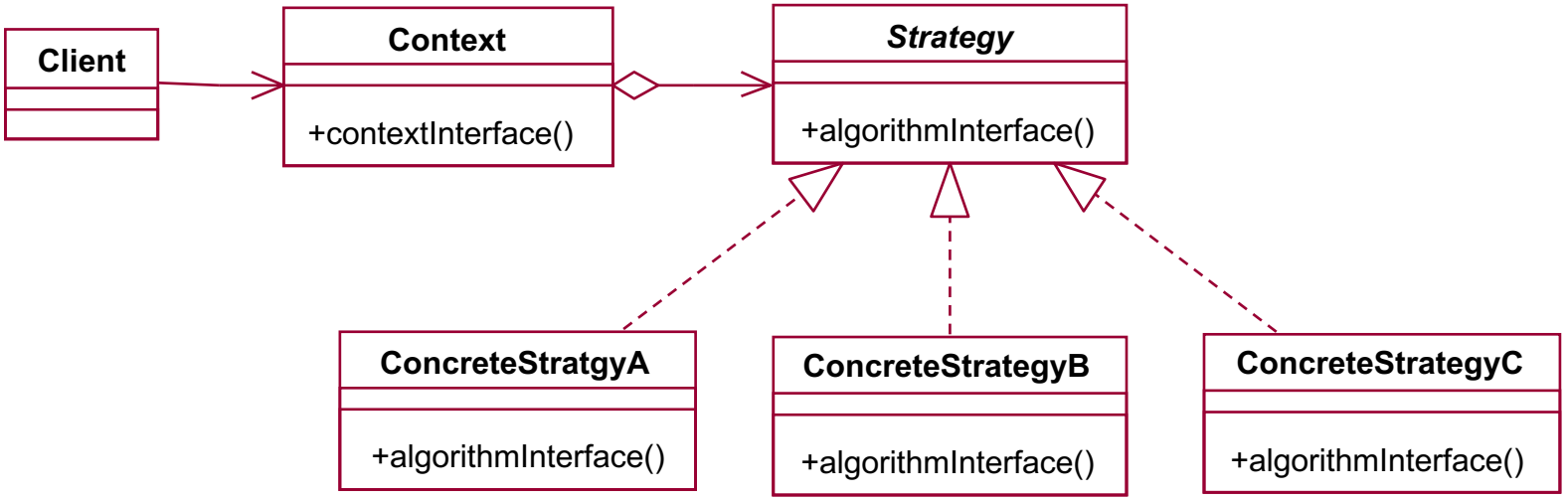

```
public class Run {

        public static void main(String[] args) {
            TaxPayer one = new TaxPayer
                        (TaxPayer.EMPLOYEE, 50000);
            TaxPayer two = new TaxPayer
                    (TaxPayer.COMPANY, 100000);
            TaxPayer three = new TaxPayer
                     (TaxPayer.TRUST, 30000);
            System.out.println(one.payTax());
            System.out.println(two.payTax());
            System.out.println(three.payTax());
    }
}
```

# Before Strategy Pattern


```
public class TaxPayer {
        public static final int COMPANY = 0;
        public static final int EMPLOYEE = 1;
        public static final int TRUST = 2;
        private static final double COMPANY_RATE = 0.30;
        private static final double EMPLOYEE_RATE = 0.45;
        private static final double TRUST_RATE = 0.35;
        private double income;
        private final int type;

        public TaxPayer(int type, double income) {
         this.type = type;
         this.income = income;
        }

        public double getIncome() {    return income;  }
        public double payTax() {
         switch (type) {
            case COMPANY:  return income * COMPANY_RATE;
            case EMPLOYEE: return income * EMPLOYEE_RATE;
            case TRUST:    return income * TRUST_RATE;
            default:       throw new IllegalArgumentException();
          }
    }
}
```

# Strategy Pattern (Step 2)

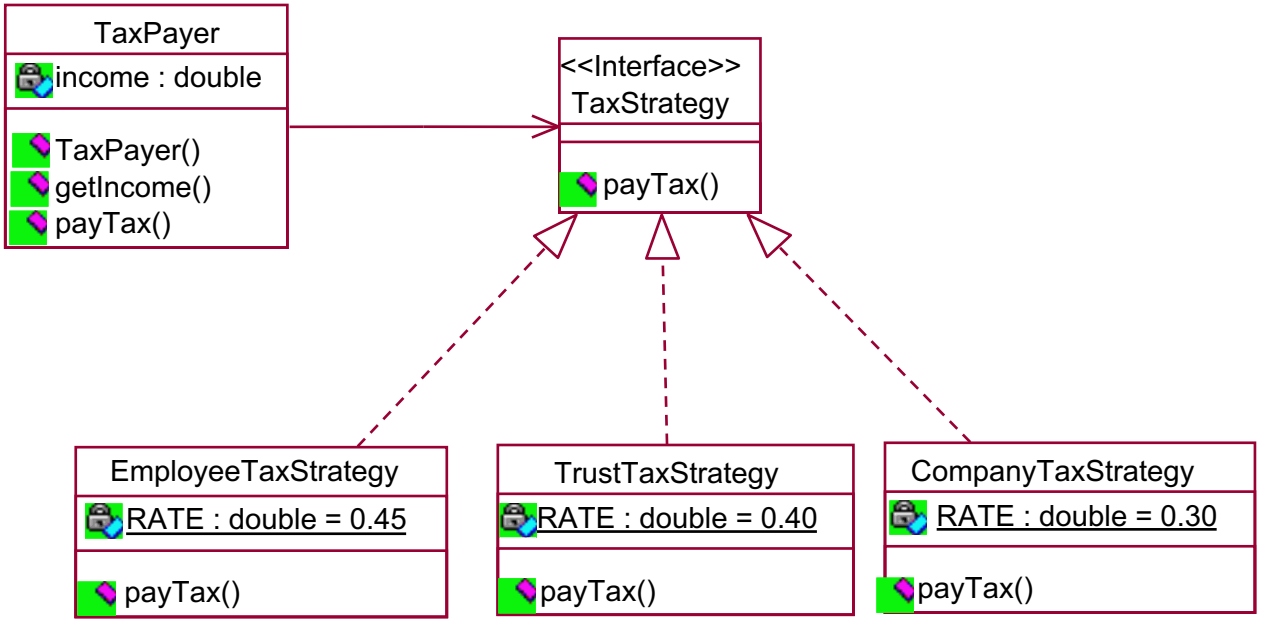
- คลาส TaxPayer ที่ทำงานได้ตามต้องการสำหรับการคำนวณง่าย ๆ แต่หากมีการเปลี่ยนแปลงเกิดขึ้นในการคำนวณ การแก้ไขจะต้องทำที่คลาสโดยตรง
- นั่นคือในกรณีที่มีคำสั่ง **switch** statements ภายในคลาส ผู้ใช้ต้องอัพเดททุกครั้งที่เพิ่ม tax payer ใหม่เสมอ
- วิธีการที่ดีกว่าคือการใช้ Strategy pattern



# Strategy Pattern

- การแปลง **switch** statement ให้อยู่ในรูป Strategy Pattern โดยใช้ TaxStrategy interface แทน จากนั้นแยกการโค้ดสำหรับการคำนวณที่แตกต่างกันออกไป

```
public interface TaxStrategy {
    public double payTax (double income);
}
```



# Strategy Pattern

```
public class CompanyTaxStrategy implements TaxStrategy {
      private static final double RATE = 0.30;
      public double payTax(double income) {
        return income * RATE ;
      }
}

public class EmployeeTaxStrategy implements TaxStrategy {
      private static final double RATE = 0.45;
      public double payTax(double income) {
        return income * RATE;
          }
}

public class TrustTaxStrategy implements TaxStrategy {
      private static final double RATE = 0.40;
      public double payTax(double income) {
        return income * RATE;
      }
}
```

# Strategy Pattern


- ดังนั้นคลาส TaxPayer จะประกอบไปด้วย TaxStrategy ออปเจคแทน **int** type จาก Before Strategy Pattern
- จากนั้นคุณสมบัติแบบ Polymorphism จะถูกนำมาใช้เงื่อนไขต่าง ๆ ดังนี้

```
public class TaxPayer {
    private TaxStrategy strategy;
    private double income;

    public TaxPayer(TaxStrategy strategy, double income) {
        this.strategy = strategy;
        this.income = income;
    }
    public double getIncome() {
        return income;
    }
    public double payTax() {
        return strategy.payTax(income);
    }
}
```

# Strategy Pattern


```
public class Run {
      public static void main(String[] args) {
         TaxPayer one = new TaxPayer(new EmployeeTaxStrategy(), 50000);
         TaxPayer two = new TaxPayer(new CompanyTaxStrategy(), 100000);
         TaxPayer three = new TaxPayer(new TrustTaxStrategy(), 30000);
         System.out.println(one.payTax());
         System.out.println(two.payTax());
         System.out.println(three.payTax());
      }
}
```

# Strategy: Consequences

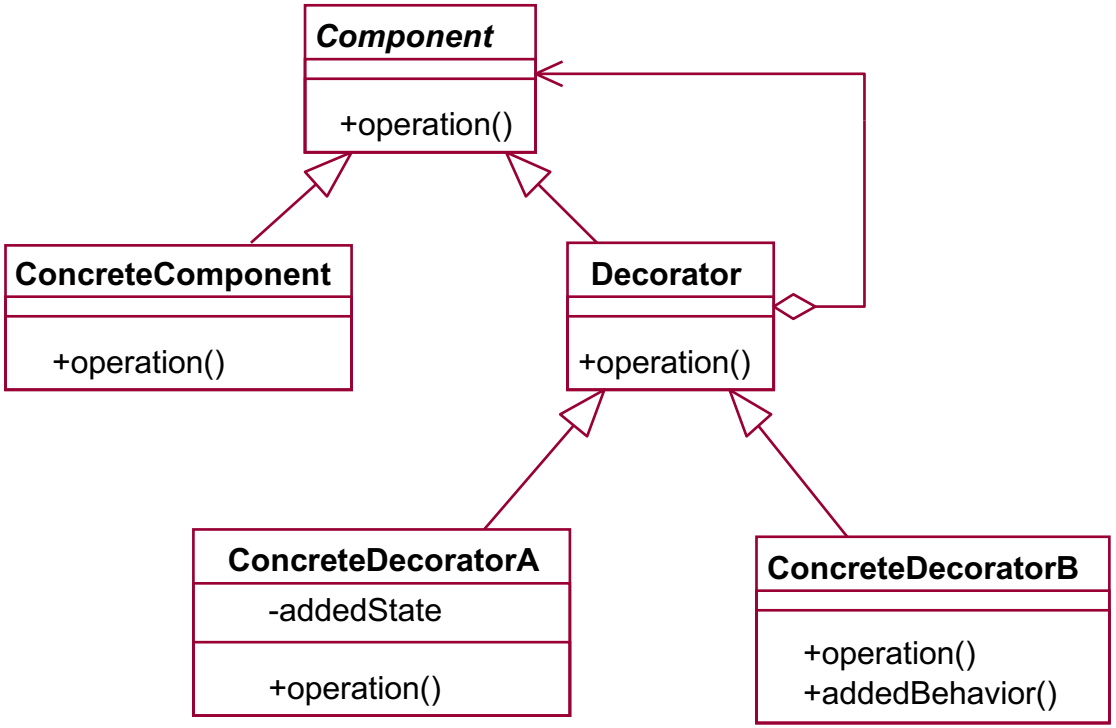

- ใช้ Strategy pattern เมื่อ:
  - คลาสกำหนดพฤติกรรมไว้หลายแบบ แต่แตกต่างกันเฉพาะพฤติกรรมบางส่วน
  - ใช้เงื่อนไขต่าง ๆ ในการกำหนดการทำงานตามพฤติกรรมที่กำหนด
  - การแก้ไขทำได้โดยย้ายเงื่อนไขเหล่านั้นให้ไปอยู่ในคลาสที่เหมาะสม (Strategy class)
- ข้อดี
  - เป็นทางเลือกสำหรับคลาสสืบทอดเพื่อให้สามารถทำงานได้กับพฤติกรรมหรืออัลกอลิทึมส์ที่แตกต่างกัน
  - โค้ดฝั่งไคลเอนต์ปราศจากชุดคำสั่งแบบเงื่อนไขต่าง ๆ เป็นจำนวนมาก

# Decorator Pattern


- Decorator Patter ใช้สำหรับ Modify หรือเพิ่มฟังก์ชันการทำงานของออปเจคแบบไดนามิคโดยไม่มีการแก้ไขโค้ด
- ใช้ในกรณีที่ออปเจคหนึ่ง ๆ มีพฤติกรรมการที่กำหนดไว้แล้ว แต่ต้องการเพิ่มพฤติกรรมใหม่เข้าไปเพื่อให้เหมาะสมกับสถานการณ์ที่เปลี่ยนแปลงไป โดยใช้กลไกที่เรียกว่า Composition แทนการสร้างออปเจคใหม่จากคลาสสืบทอด
- แพตเทิร์นแบบนี้บางครั้งอาจถูกเรียกว่า Wrapper ซึ่งยอมให้ออปเจคหนึ่ง ๆ สามารถห่อหุ้มออปเจคอื่นไว้ภายในได้
- ผู้ใช้สามารถเรียกใช้ออปเจคได้โดยการเพิ่มหรือเรียกใช้หลาย ๆ Wrapper ซ้ำแล้วซ้ำอีกได้ โดยไม่ต้องมีการแก้ไขโค้ดทุกครั้งที่ในสถานการณ์ที่เปลี่ยนแปลงไป

# Decorator Patterns

- Decorator Pattern ประกอบด้วยออปเจคซ้อนกัน ส่วนที่อยู่ภายนอกเรียกว่า Decorator ออปเจค และภายในเรียกว่า C1omponent ออปเจค
- โดย Decorator ส่งการร้องขอไปยัง Component เป็นหลัก



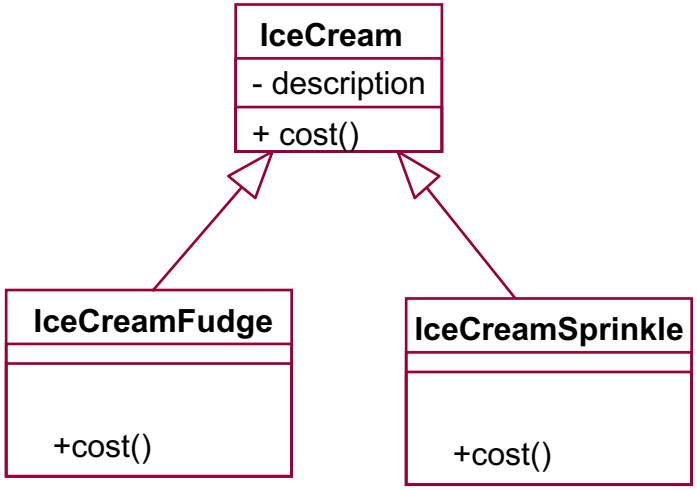
# Decorator Patterns

- ตัวอย่างเช่น ไอศกรีมต่าง ๆ เช่น Chocolate ที่ใส่ส่วนประกอบต่างกัน เช่น Sprinkle, Fudge , Whip Cream เป็นต้น ส่งผลให้ราคาแตกต่างกันตามไปด้วย
- หากสร้างคลาสขึ้นมาเพื่อรองรับไอศครีมที่มี “ส่วนผสม” ต่างกัน อาจจำเป็นต้องสร้างคลาสขึ้นมาเป็นจำนวนมาก




# Problem : Decorator Patterns

- ต้องการคำนวณค่า IceCream ดังนั้นจึงจำเป็นต้องสร้างเมธอด cost()



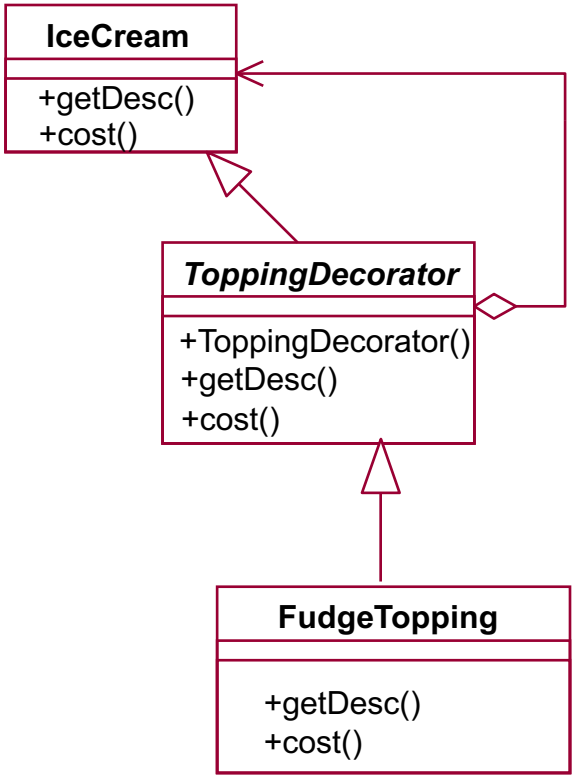

```
class IceCream
{
    private String description;
    public String getDescription() {
    }
    public double Cost() {
      return 30;
   }
}

class IceCreamFudge extends IceCream
{
      public double Cost()
      {
        return 33;
      }
}

class IceCreamSprinkle extends IceCream
{
      public double Cost()
      {
        return 40;
      }
}
```

# Decorator Patterns

- ในทางปฏิบัติอาจใช้การสร้างคลาส Chocolate เพียงคลาสเดียว และเพิ่มส่วนผสมต่าง ๆ เข้าไปในคลาส เพื่อให้การทำงานสามารถทำได้ง่ายขึ้น
- ในกรณีที่สร้างคลาสขึ้นใหม่ เช่น Vanilla สามารถใช้ส่วนผสมร่วมกับคลาส Chocolate ที่มีอยู่แล้วได้
  - การทำงานในลักษณะนี้จำเป็นต้องสร้าง interface ขึ้นมาเพื่อใช้กับคลาส Chocolate และ Vanilla สำหรับการกำหนดส่วนผสมต่าง ๆ ไว้ใช้ร่วมกัน

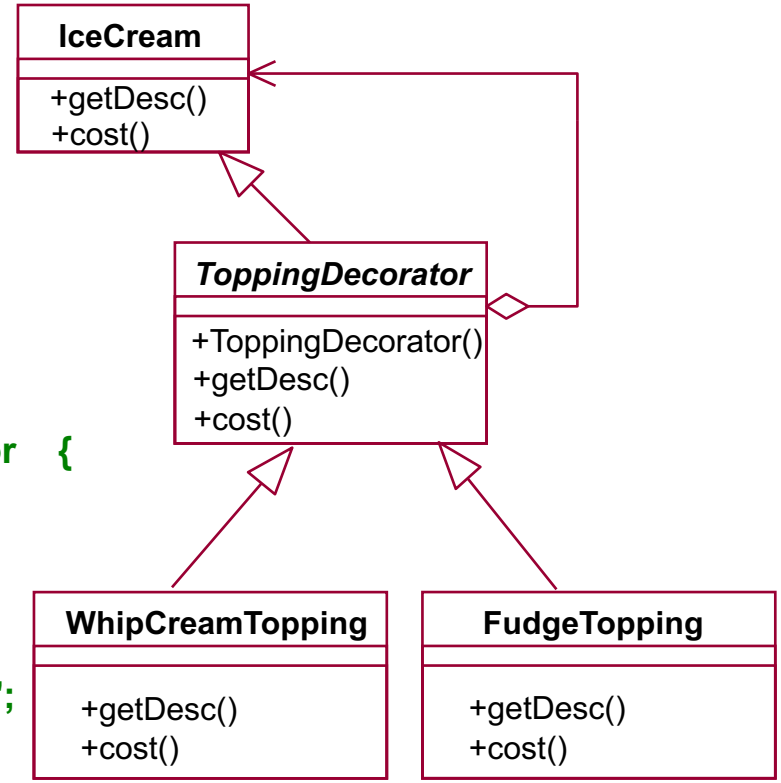
# Decorator Patterns

```
interface IceCream   {
    String getDesc();
    double getCost();
 }

abstract class ToppingDecorator implements IceCream    {
    protected  IceCream dec;
    public ToppingDecorator(IceCream dec)    {
          this.dec = dec;
    }
    public abstract String getDesc();
    public abstract double getCost();
}

class FudgeTopping extends ToppingDecorator        {
   public FudgeTopping(IceCream dec)  {
          super(dec);
   }
  public String getDesc()    {
    return dec.getDesc() + " Fudge Topping";
 }
  public double getCost()  {
          return dec.getCost() + 0.4;
  }
}
```



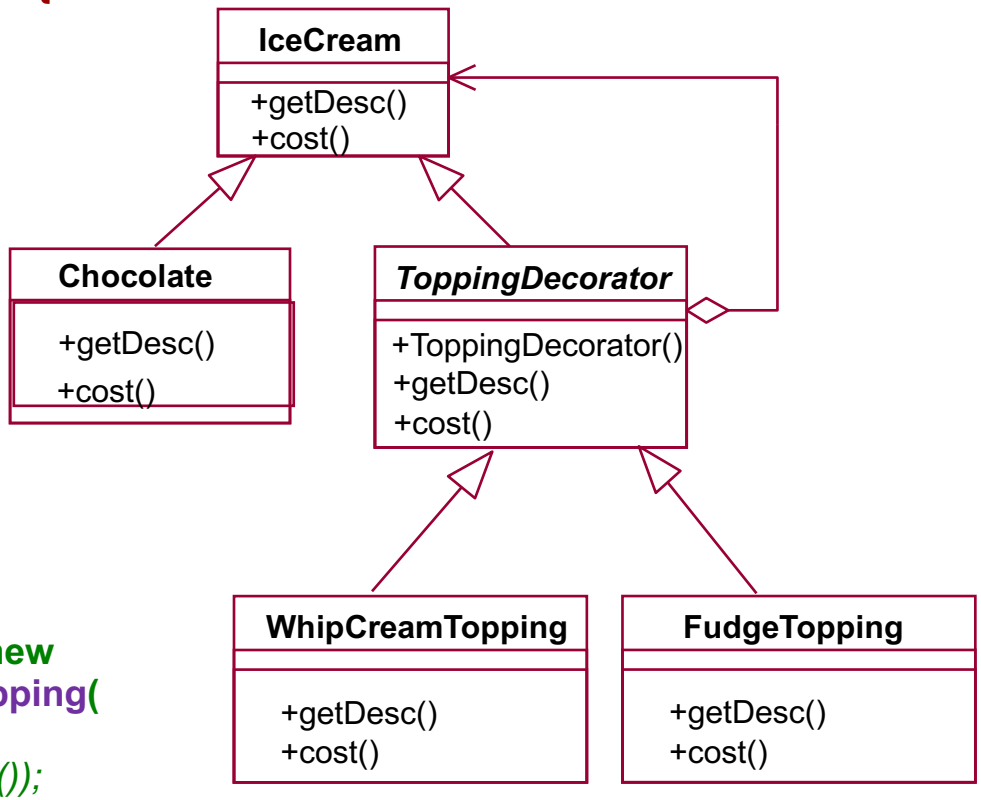
# Decorator Patterns

```
class SprinkleTopping extends ToppingDecorator    {
     public SprinkleTopping(IceCream dec)  {
           super(dec);
      }
     public String getDesc()    {
          return dec.getDesc() +  " Sprinkle Topping";
     }
     public double getCost()  {
          return dec.getCost() + 0.2;
   }
}
class WhipCreamTopping extends ToppingDecorator    {
     public WhipCreamTopping(IceCream dec){
           super(dec);
      }
     public String getDesc()   {
         return dec.getDesc() +  " WhipCream Topping";
     }
     public double getCost()  {
         return dec.getCost() + 0.3;
    }
}
```



# Decorator Pattern

```
class Chocolate implements IceCream   {
   public String getDesc()   {
         return "Chocolate Ice Cream";
       }
    public double getCost()  {
         return 1.5;
     }
}
```




```
public class RunDecorator {

public static void main(String[] args) {

 IceCream ice = new SprinkleTopping(new
     FudgeTopping(new WhipCreamTopping(
           new Vanilla())));
            System.out.println(ice.getDesc());
            System.out.println(ice.getCost());

   }
}
```

# Advantages

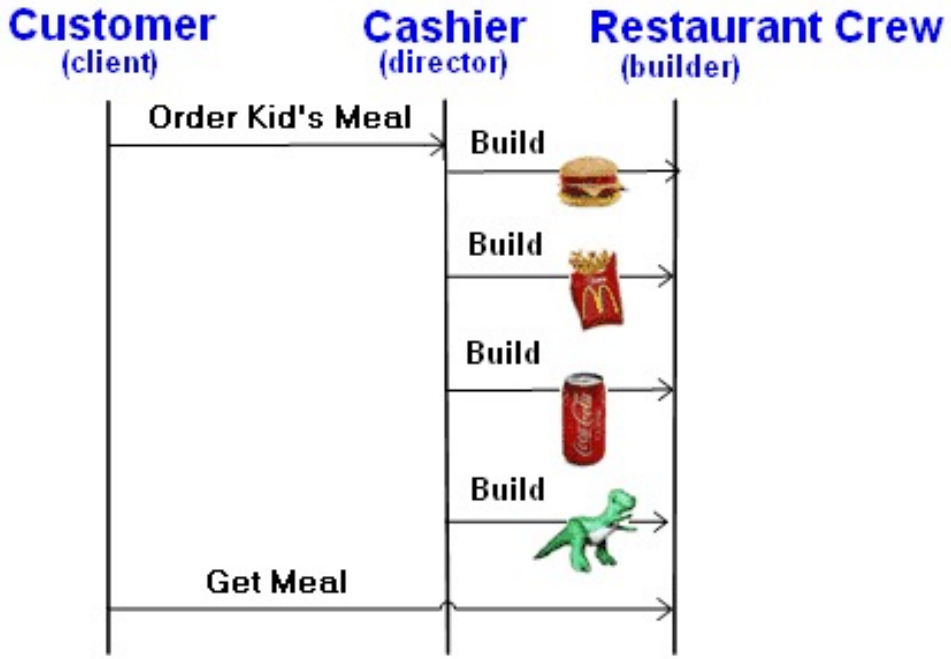

- ใช้ Decorator Pattern เมื่อต้องการความยืดหยุ่นมากกว่าคลาสสืบทอด
  - สามารถเพิ่มหรือลดการทำงานต่าง ๆ ได้ในขณะรันไทม์ โดยไม่มีผลกระทบต่อออปเจคอื่น ๆ
  - ป้องกันปัญหาที่เกิดจากคลาสมีภาระหน้าที่มากเกินไปในลำดับชั้นด้านบน ซึ่งแก้ไขได้โดยการสร้างคลาสง่าย ๆ และเพิ่มฟังก์ชันการทำงานเข้าไปในขณะที่มีการทำงานเพิ่มขึ้นทีละชั้น
  - แก้ปัญหาโดยใช้คลาสสืบทอดอาจไม่เหมาะสม มีความเป็นไปได้ที่คลาสสืบทอดจะมีจำนวนมากเกินกว่าจะควบคุมได้

# Builder Pattern

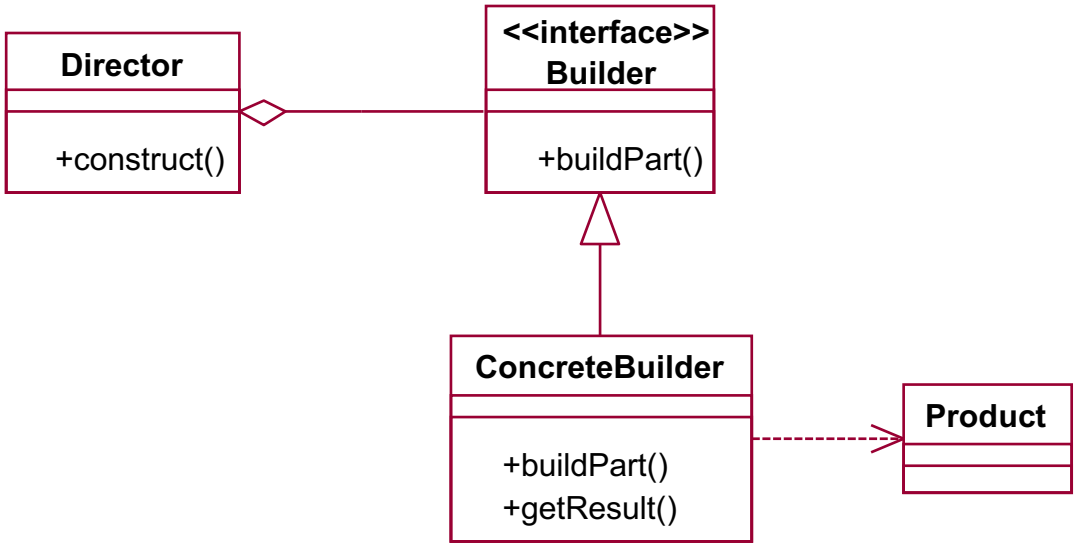

- Builder แพตเทิร์นใช้ในการสร้างออปเจคที่มีความซับซ้อน หรือออปเจคประกอบไปด้วยออปเจคอื่น ๆ
- โดยใช้กระบวนการสร้างที่เหมือนกันสามารถใช้ในการสร้างออปเจคที่มีการนำเสนอที่แตกต่างกันได้
- เพื่อป้องกันไม่ให้ไคลเอนต์ทราบรายละเอียดของการสร้างออปเจค ดังนั้นไคลเอนต์ระบุเฉพาะชนิดและรายละเอียดของออปเจคที่ต้องการสร้างเท่านั้น
- การทำงานในลักษณะนี้ถือเป็นการแยกกระบวนการสร้างออปเจคที่ซับซ้อนออกจากการนำเสนอ
  - “Separate Complex Object Construction from its Presentation.”

# Main Concepts

- ใช้กระบวนการสร้างแบบเดียวกันสำหรับการนำเสนอที่แตกต่างกัน
- เช่น อาหารจานด่วนประกอบด้วย Burger, Fries, Coke, และ Toy
- แม้อาหารมีได้หลายชนิดแต่กระบวนการสร้างจะเหมือนกัน นั่นคือเมื่อลูกค้าสั่งอาหารสำหรับเด็กหรือผู้ใหญ่กระบวนการจะเหมือนกันนั่นเอง แต่รายละเอียดต่างกัน

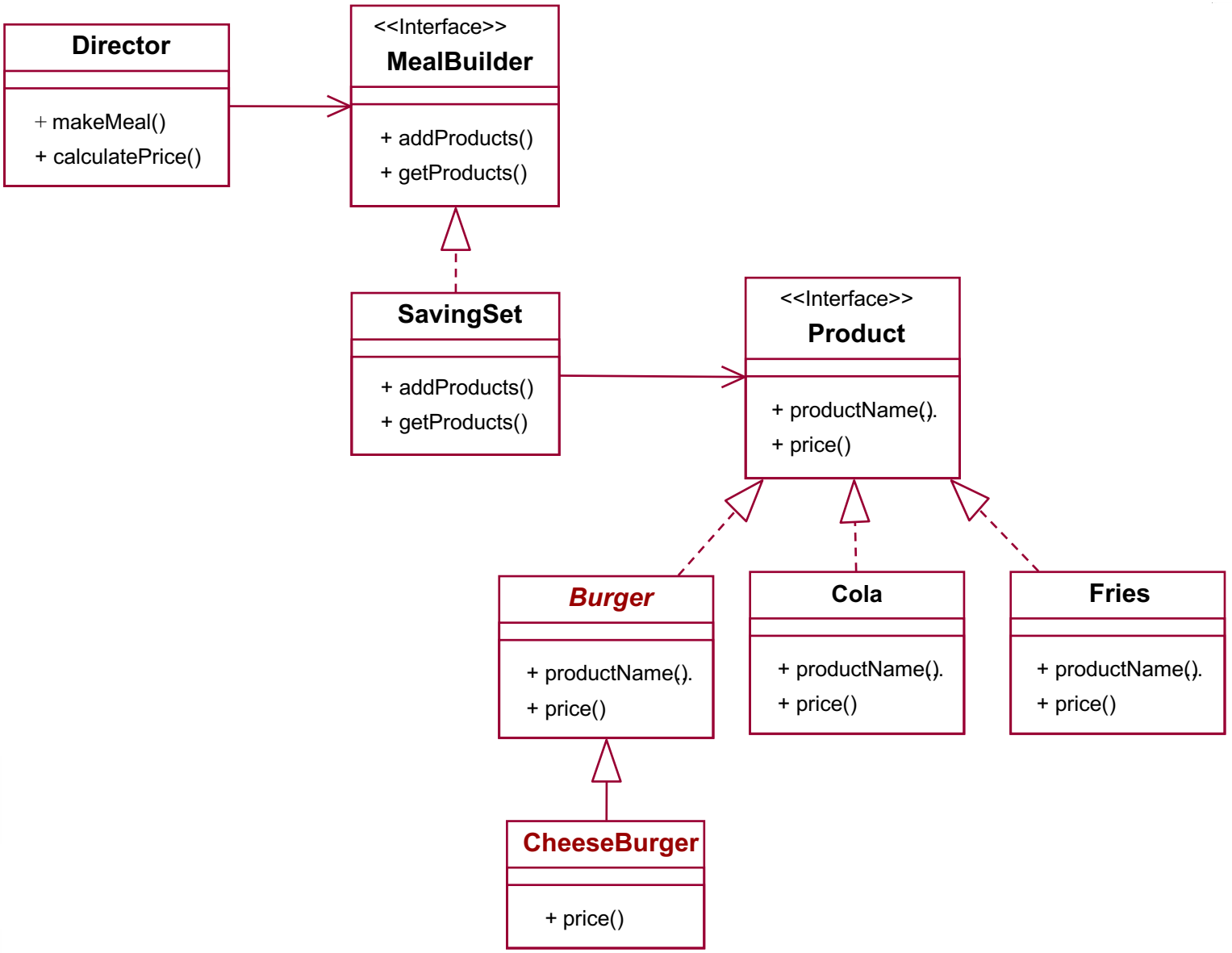


# Builder Pattern Concepts



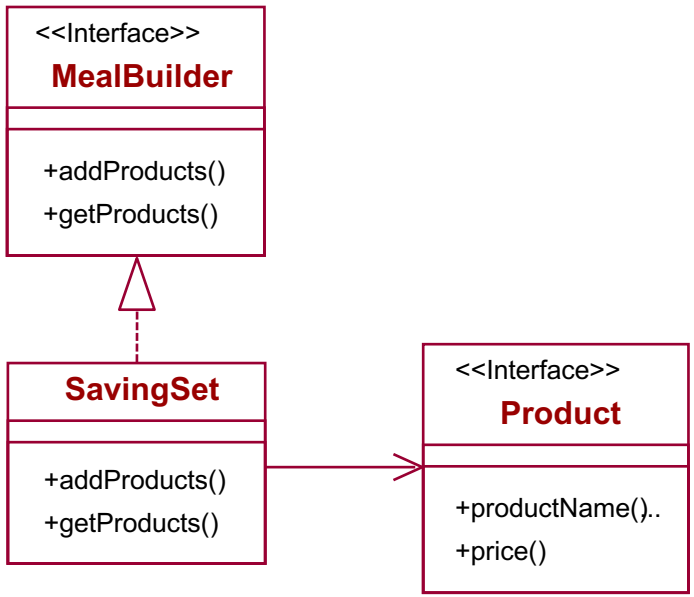

- Director ผ่านค่าไปยัง Builder เพื่อระบุส่วนประกอบใดของ Product ที่ต้องการสร้าง
- Builder เป็น interface ที่กำหนดพฤติกรรมสำหรับการสร้างส่วนประกอบต่าง ๆ ไว้
- ConcreteBuilder ทำหน้าที่สร้างและประกอบส่วนต่าง ๆ ของ Product เข้าด้วยกัน
- Product ประกอบไปด้วยส่วนประกอบต่าง ๆ ที่ถูกสร้างขึ้นแยกจากกัน

# Collaborations



# Builder Pattern

```
interface MealBuilder {
        public abstract void addProducts();
        public abstract Product[] getProducts();
}

class SavingSet implements MealBuilder {

        private Product[] product = new Product[3];
        public void addProducts() {
          product[0] = new CheeseBurger();
          product[1] = new Fries();
          product[2] = new Cola();
        }

        public Product[] getProducts() {
            return product;
        }
}
```



# Builder Pattern


```
class ComboSet implements MealBuilder {
      private Product[] product = new Product[4];
      public void addProducts() {
        product[0] = new CheeseBurger();
        product[1] = new Fries();
        product[2] = new Cola();
        product[3] = new Toy();
      }
      public Product[] getProducts() {
        return product;
      }
}
```

```
class Director    {
  private MealBuilder meal;
  public void makeMeal(MealBuilder obj)
   {
        meal = obj;
         meal.addProducts();
  }
  public int calculatePrice() {
     int totalPrice = 0;
     Product[] p = meal.getProducts();
         for ( int i = 0; i < p.length; i++) {
       totalPrice += p[i].price();
              System.out.println(" "+
              p[i].productName()+" : "+ p[i].price());
        }
          return totalPrice;
       }
}
```

```
interface Product {
        public String productName();
        public int price();
}
abstract class Burger implements Product {
public String productName() {
        return "Burger";
 }
public abstract int price();
}

class CheeseBurger extends Burger {
public int price() {    return 39;   }
}
```

```
class Fries implements Product {
     public String productName() {    return "Fries";   }
     public int price() {
        return 25;
    }
}
class Cola implements Product {
      public String productName() {    return "Cola“;   }
      public int price() {
          return 15;
   }
}
class Toy implements Product {
      public String productName() {   return "Toy“;     }
      public int price() {
            return 55;
     }
}
```

```
class Run {
     public static void main (String[] args) {
        Director meal = new Director();
        meal.makeMeal(new ComboSet());
        System.out.println("Total Meal price is $"+ meal.calculatePrice() + " only ");
   }
}
```

# Why Use Builder?

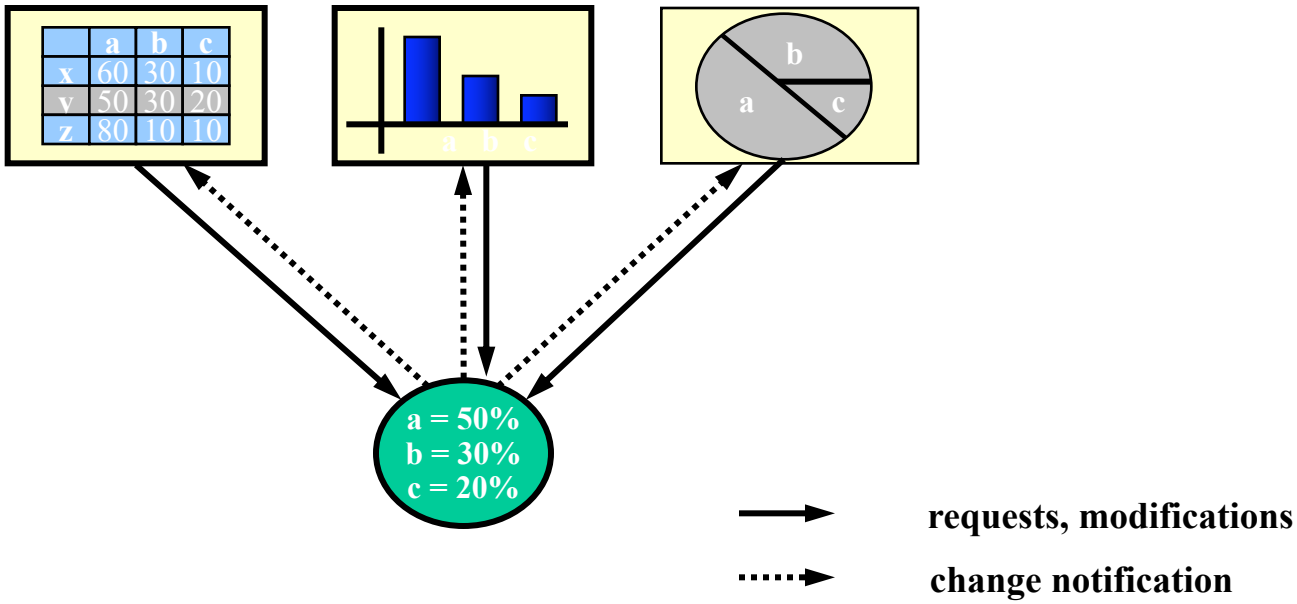

- ความเหมาะสมในการใช้งาน:
  - ส่วนที่เป็นอัลกอลิทึมส์สำหรับการสร้างออปเจคที่ซับซ้อนควรเป็นอิสระจากวิธีการและส่วนที่นำมาประกอบกันเป็นออปเจคที่ต้องการ
  - กระบวนการสร้างจะต้องยอมให้มีการนำเสนอที่แตกต่างกันสำหรับออปเจคที่ถูกสร้างขึ้น
  - แพตเทิร์นแบบนี้เป็นประโยชน์เนื่องจากยอมให้โปรแกรมมีความยืดหยุ่นในการสร้างออปเจคต่าง ๆ ที่สามารถลบหรือเพิ่มได้อย่างเป็นอิสระ

# Observer Pattern


- เป็น Design Pattern ที่ยอมให้หลาย ๆ ออปเจคซึ่งทำหน้าที่เป็น observer และถูกแจ้งให้ทราบเมื่อออปเจคที่เป็น subject มีการเปลี่ยนแปลงเกิดขึ้น
- โดยแต่ละ observer จะถูกติดตั้ง (registers) เข้ากับ subject และเมื่อมีการเปลี่ยนแปลงเกิดขึ้น subject จะแจ้งให้ออปเจคที่เป็น observers ทราบในเวลาเดียวกัน
- โดยปกติเป็นการทำงานที่มีความสัมพันธ์ระหว่างออปเจค แบบ dependency โดยมีจำนวน “one-to-many”
- เมื่อออปเจคหนึ่งมีการเปลี่ยนแปลงสถานะ (state) จะมีการแจ้งไปยังทุก ๆ ออปเจคที่เกี่ยวข้องให้ทราบและมีการอัพเดทโดยอัตโนมัติ

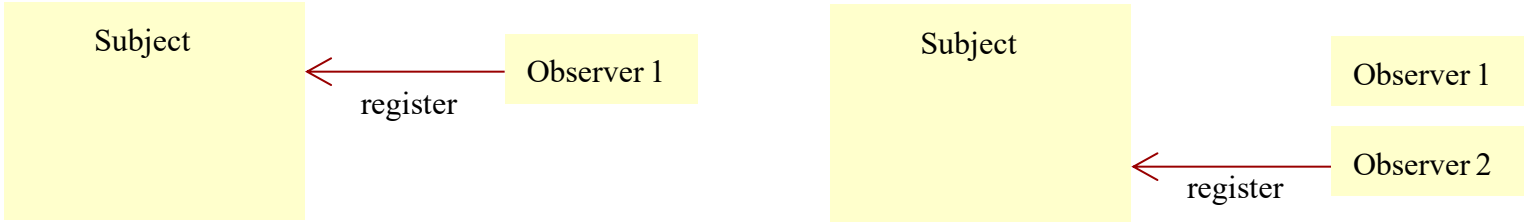
# Subject & Observer

- Subject : เป็นออปเจคการเปลี่ยนแปลงค่าสถานะมีผลต่อออปเจคที่เกี่ยวข้อง
- Observer: เป็นออปเจคที่ขึ้นอยู่กับ subject และมีการอัพเดทค่าตามสถานะของ subject

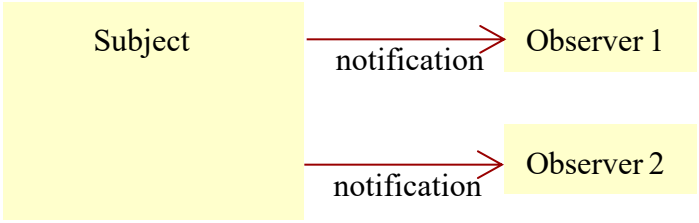


# Observer Pattern - Working

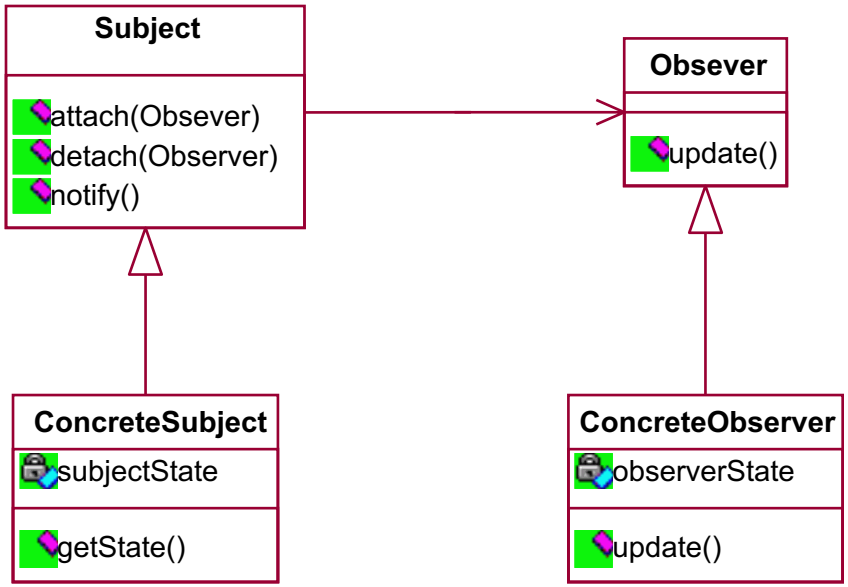
- Observers ทั้งหมดจะถูกติดตั้ง “register” เพื่อรับการแจ้งให้ทราบถึงความเปลี่ยนแปลงที่เกิดขึ้นกับ Subject




- เมื่อมีเหตุการณ์ที่ก่อให้เกิดความเปลี่ยนแปลงแก่ Subject ทุก ๆ Observers จะถูก“notified’ ทันที



# Observer Pattern - Key Players




- Subject : มีเมธอดที่ใช้ติดตั้งและถอดถอน observer
- Observer : เป็น interface ทำหน้าที่แจ้งให้ทราบถึงความเปลี่ยนแปลงที่เกิดขึ้นกับออปเจค
- ConcreteSubject: เก็บค่า “state” และ notification เมื่อสถานะมีการเปลี่ยนแปลง
- ConcreteObserver: เป็นคลาสที่อิมพลีเมนต์มาจาก Observer interface

# Creating an observer interface


- ขั้นตอนแรกเป็นการกำหนด interface หรือ abstract class เพื่อใช้สำหรับ observer
- Observer interface ถูกกำหนดขึ้นเพื่อให้รับทราบการเปลี่ยนสถานะผ่านเมธอด update() ซึ่งในกรณีนี้เป็นการผ่านค่าการทำงานของ database (เช่น “edit”, “delete”, “create” เป็นต้น) และ record ที่มีการเปลี่ยนแปลง

```
public interface Observer
{
    public void update(String operation, String record);
}
```

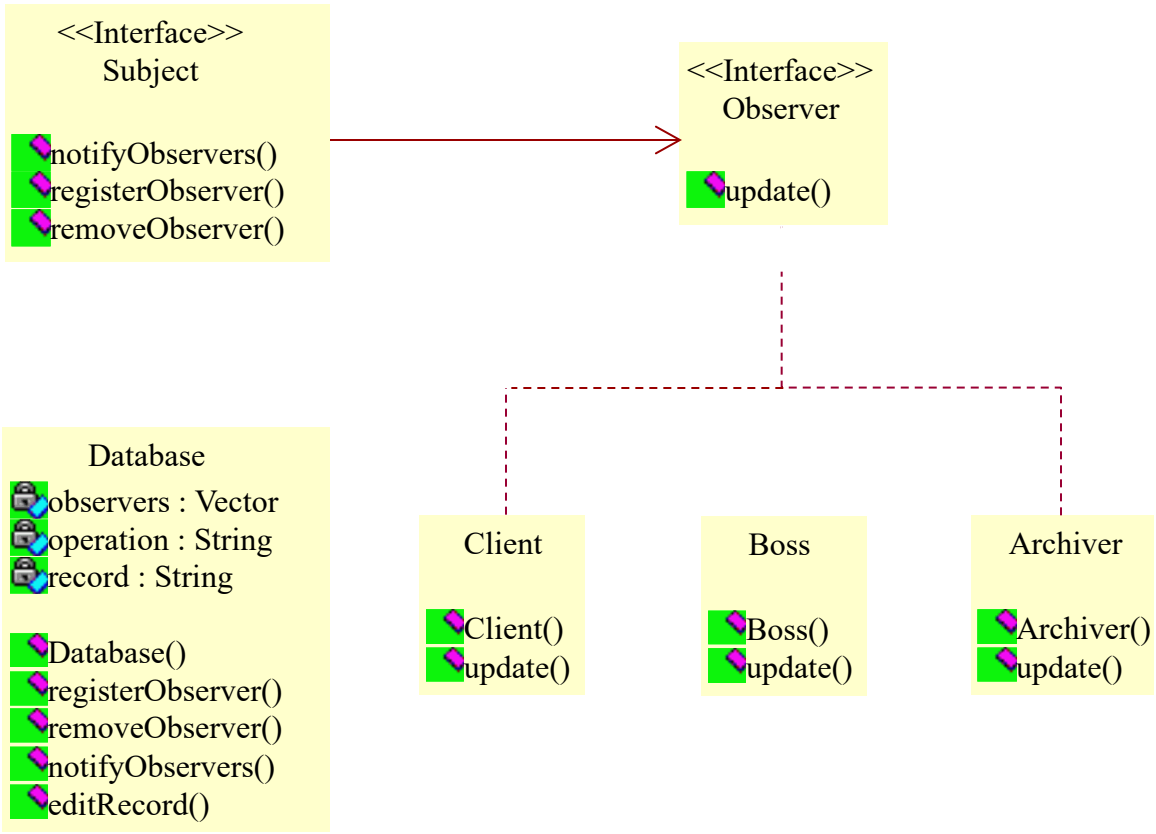
- เมื่อ observer มีการเรียกใช้เมธอด update() จะสามารถผ่านค่าการทำงานและเรคคอร์ดที่ต้องการได้

# Creating a subject

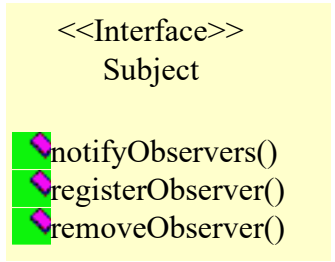

- โดยปกติแล้ว Observer Pattern จะประกอบไปด้วย Subject ที่ใช้สำหรับเก็บค่าสถานะ และใช้สำหรับแจ้งให้ observer ต่าง ๆ ทราบถึงการเปลี่ยนแปลงที่เกิดขึ้น
- ดังนั้น Subject จึงถูกกำหนดขึ้นในรูปของ interface ที่ประกอบไปด้วยเมธอดที่ใช้สำหรับการติดตั้ง Observer เข้าสู่ Subject ดังนี้

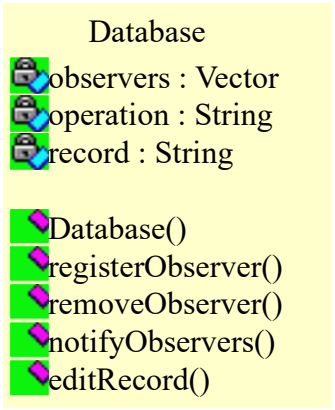
```
public interface Subject
{
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

# Observer Pattern - UML



# Creating Concrete Subject



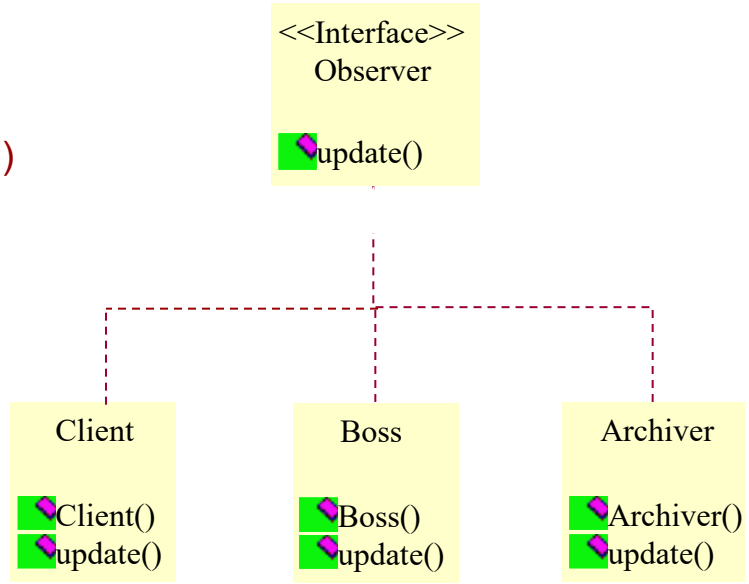

```
import java.util.*;

public class Database implements Subject   {
        private Vector observers;
        private String operation;
        private String record;

        public Database() {   observers = new Vector();   }

public void registerObserver(Observer o) {  observers.add(o);  }
public void removeObserver(Observer o) { observers.remove(o); }

public void notifyObservers()  {
        for (int index = 0; index < observers.size(); index++) {
          Observer observer = (Observer)observers.get(index);
          observer.update(operation, record);
        }
}
public void editRecord(String operation, String record) {
        this.operation = operation;
        this.record = record;
        notifyObservers();
        }
}
```

# Creating Concrete observers

```
public class Archiver implements Observer  {
    public Archiver()  {
}
   public void update(String operation, String
   record)    {
    System.out.println("The archiver says a " +
      operation +  " operation was performed
      on " + record);
   }
}
```

```
public class Client implements Observer  {
   public Client()  {
  }
  public void update(String operation, String
       record)   {
       System.out.println("The client says a " +
            operation + " operation was
             performed on " + record);
       }
}
```

```
public class Boss implements Observer  {
    public Boss()  { }
    public void update(String operation, String record)
    {
       System.out.println("The boss says a " +
        operation +  " operation was performed on " +
            record);
      }
}
```



# Testing the Database observers


- Observer pattern ช่วยให้มีความยืดหยุ่นมากกว่า hard coding ที่ทุกสิ่งทุกอย่างอยู่ภายใน 1 บล็อก และยอมให้ผู้ใช้สามารถติดตั้งหรือถอดถอน observers ในขณะรันไทม์ได้

```
public class TestObserver
{
public static void main(String args[])
{
      Database database = new Database();
      Archiver archiver = new Archiver();
      Client client = new Client();
      Boss boss = new Boss();
      database.registerObserver(archiver);
      database.registerObserver(client);
      database.registerObserver(boss);
      database.editRecord("delete", "record 1");
  }
}
```

```
The boss says a delete operation was performed on record 1
The client says a delete operation was performed on record 1
The archiver says a delete operation was performed on record 1
```

# Observer Pattern


- Observer Pattern เหมาะสมที่จะใช้ในสถานการณ์ดังนี้:
  - เมื่อ abstraction มีสองแบบที่ไม่ขึ้นแก่กัน ให้แยกออปเจคดังกล่าวออกจากกัน เพื่อให้สามารถใช้งานได้โดยไม่ขึ้นแก่กัน
  - เช่น Subjects และ Observers จะมีลักษณะเป็น loosely coupled
  - การเปลี่ยนแปลงที่เกิดขึ้นกับ Subject หรือ Observer จะไม่มีผลกระทบระหว่างกัน
  - Subjects และ Observers สามารถนำกลับไปใช้ใหม่ได้โดยไม่ขึ้นแก่กัน
  - นอกจากนั้น Subject รู้จักเฉพาะ Observer interface และไม่ใช้โค้ดของการพัฒนา