

เอกสารประกอบการสอน
วิชา COS4311

Pattern and Software Design
รูปแบบและการออกแบบซอฟต์แวร์

โดย

ผศ.ดร.ภาวลัย ไกรพิรพรวน

July 2568

COS4311
Pattern and Software Design
รูปแบบและการออกแบบซอฟต์แวร์

1

References

- Alexander Shvets, “Dive Into Design Patterns”
- <https://refactoring.guru/design-patterns>
- <https://www.saladpuk.com/basic/solid>
- <https://www.saladpuk.com/beginner-1/design-patterns>
- <https://www.saladpuk.com/software-design/designpatterns>

เนื้อหา

- พื้นฐาน OOP
- รูปแบบการออกแบบเบื้องต้น
- หลักการในการออกแบบซอฟต์แวร์
 - SOLID
- รายการของรูปแบบการออกแบบ (**Catalog of Design Patterns**)
แบ่งตามวัตถุประสงค์ได้ 3 กลุ่ม
 - รูปแบบเชิงการสร้างอ็อบเจกต์ (**Creational Patterns**)
 - รูปแบบเชิงโครงสร้าง (**Structural Patterns**)
 - รูปแบบเชิงพฤติกรรม (**Behavioral Patterns**)

รูปแบบเชิงการสร้างอ้อมบเจกต์ (Creational Patterns)

สร้างอ้อมบเจกต์ที่มีความยืดหยุ่นและสามารถนำโค้ดกลับมาใช้ใหม่ได้

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

รูปแบบเชิงโครงสร้าง (Structural Patterns)

นำอ็อบเจกต์และคลาสต่างๆ มาประกอบกันเป็นโครงสร้างที่ใหญ่ขึ้น มีความยืดหยุ่นและมีประสิทธิภาพ

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

รูปแบบเชิงพฤติกรรม (Behavioral Patterns)

การติดต่อสื่อสารกันและการกำหนดความรับผิดชอบระหว่างอ็อบเจกต์

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

คะแนน

- ส่งโครงการครบ 22 pattern และอีกหนึ่งโครงการที่มีการใช้ 3 pattern รวมกัน จะได้ A
- ส่งโครงการ 21-22 pattern จะได้เกรด B+
- ส่งโครงการ 19-20 pattern จะได้เกรด B
- ส่งโครงการ 17-18 pattern จะได้เกรด C+
- ส่งโครงการ 15-16 pattern จะได้เกรด C
- ส่งโครงการ 13-14 pattern จะได้เกรด D+
- ส่งโครงการ 11-12 pattern จะได้เกรด D

Design Patterns

- Design patterns
 - เป็นแนวคิดวิธีการ ในการแก้ปัญหาการออกแบบซอฟต์แวร์
 - มีการจัดเตรียมแนวคิดในการแก้ปัญหาแต่ละปัญหาไว้ล่วงหน้า
- ผู้ริเริ่ม “Gang of Four” หรือ GoF ปี 1994
 - Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm
- หนังสือ Design Patterns: Elements of Reusable Object-Oriented Software

Pattern

ลำดับในการทำความเข้าใจการทำงานของแต่ละ **pattern**

- **Intent**

- ทำความเข้าใจปัญหาและวิธีแก้ปัญหาคร่าวๆ

- **Motivation**

- อธิบายปัญหาและวิธีแก้ปัญหาโดยใช้ **pattern**

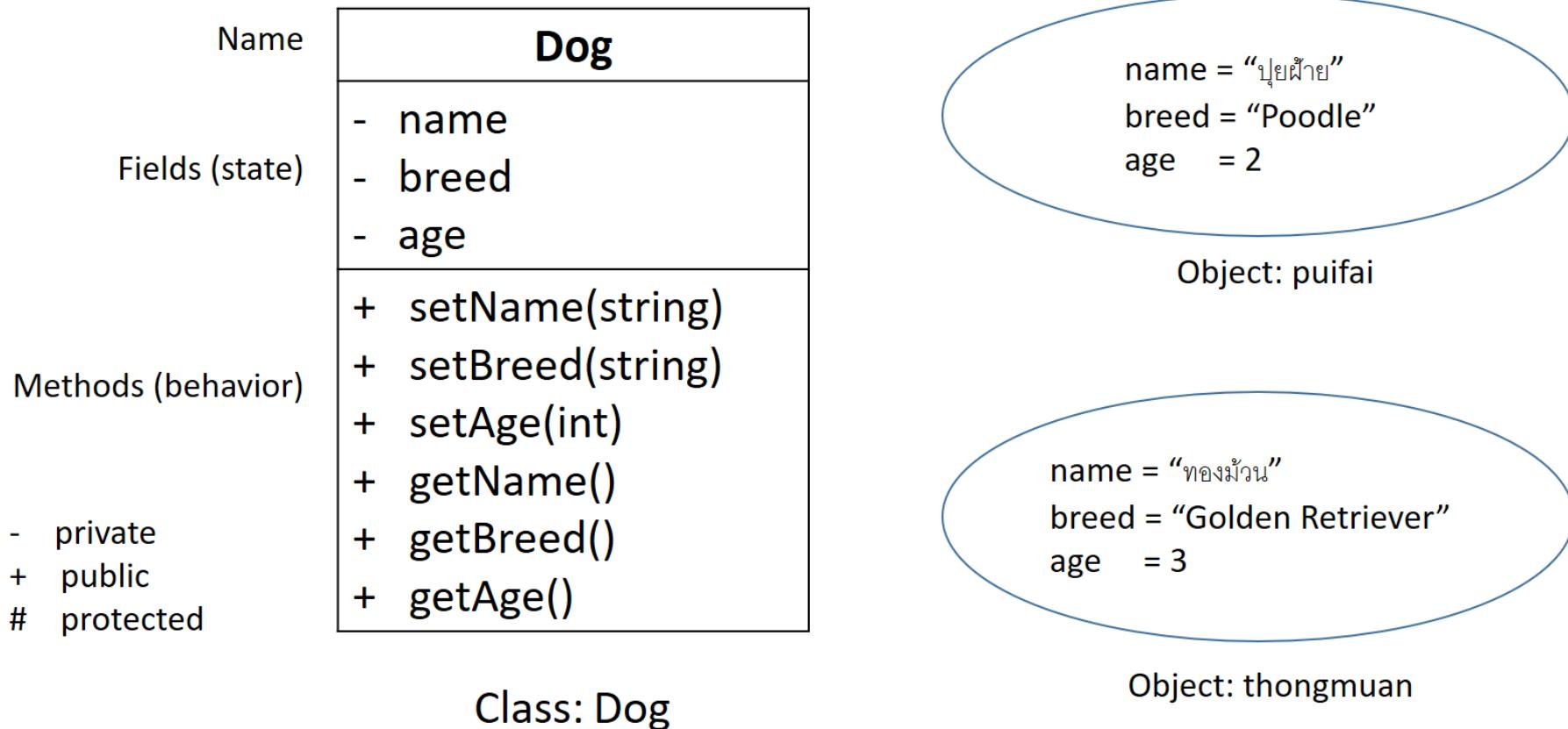
- **Structure**

- คลาสและความสัมพันธ์ระหว่างคลาส ที่ประกอบขึ้นเป็น **pattern**

- **Code example**

- ตัวอย่างโค้ดภาษาใดภาษาหนึ่งเพื่อให้ง่ายต่อการทำความเข้าใจ **pattern**

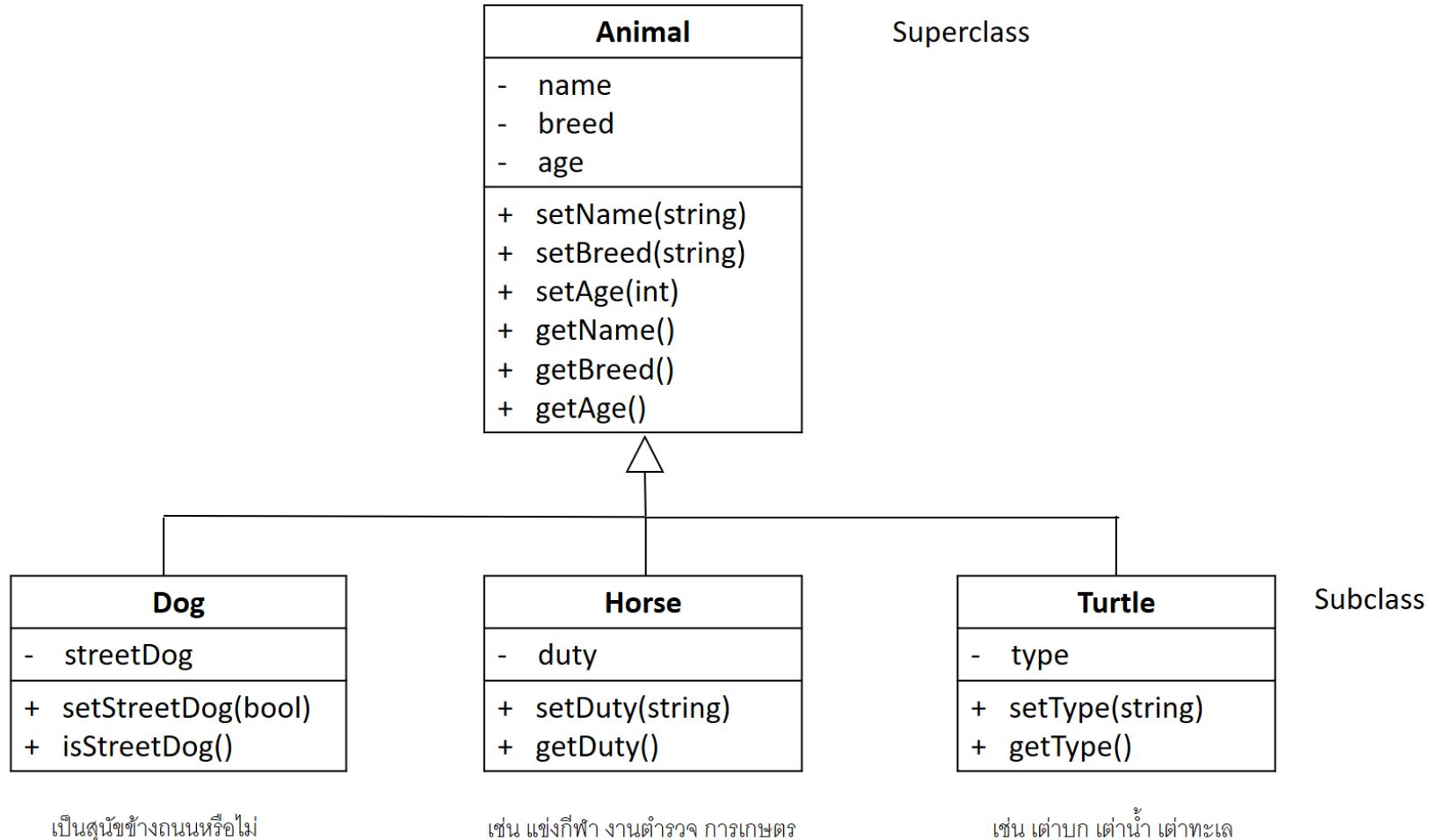
Object-Oriented Programming, OOP



คลาสเปรียบเสมือนพิมพ์เขียวที่นิยามโครงสร้างสำหรับอ็อบเจกต์

Class hierarchies

- Inheritance



หลักสำคัญของ OOP

- Abstraction
- Encapsulation
- Inheritance
- polymorphism

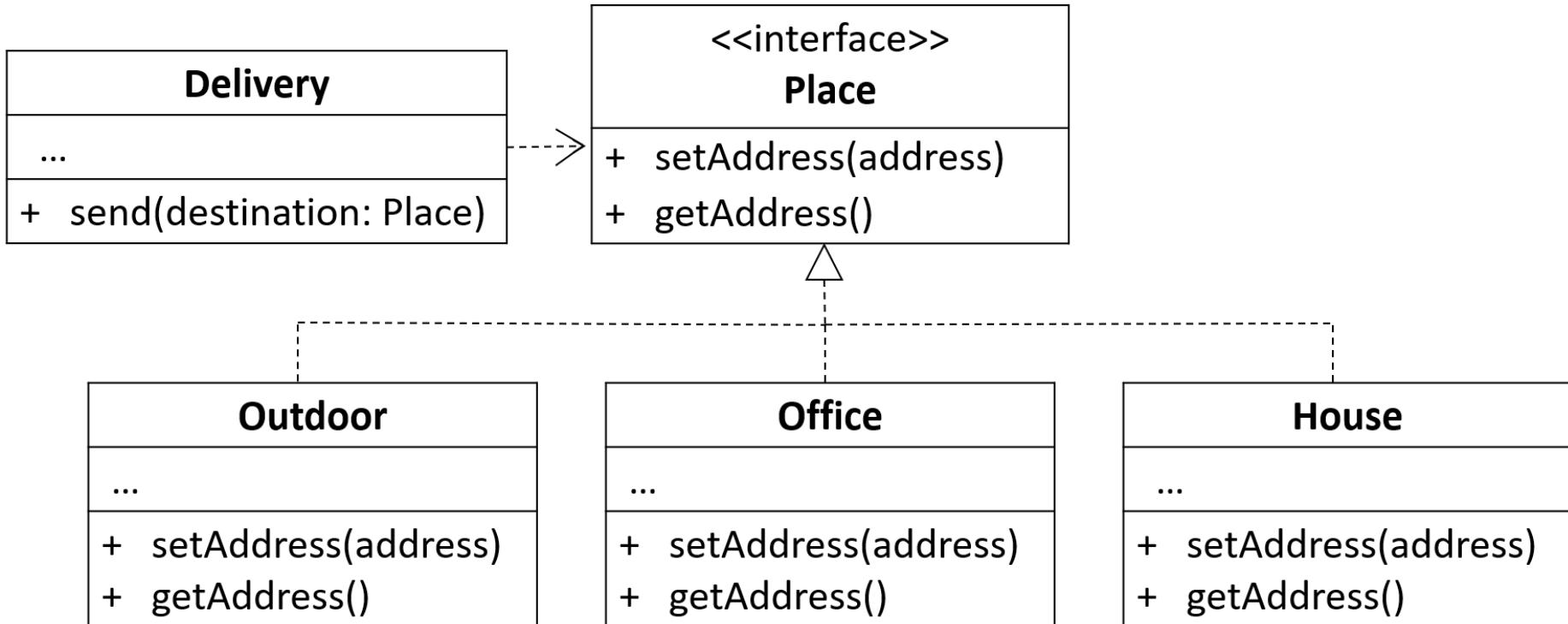
Abstraction

House
- owner
- address
- size
+ setOwner(string)
+ setAddress(string)
+ setSize(double)
+ getOwner()
+ getAddress()
+ getSize()

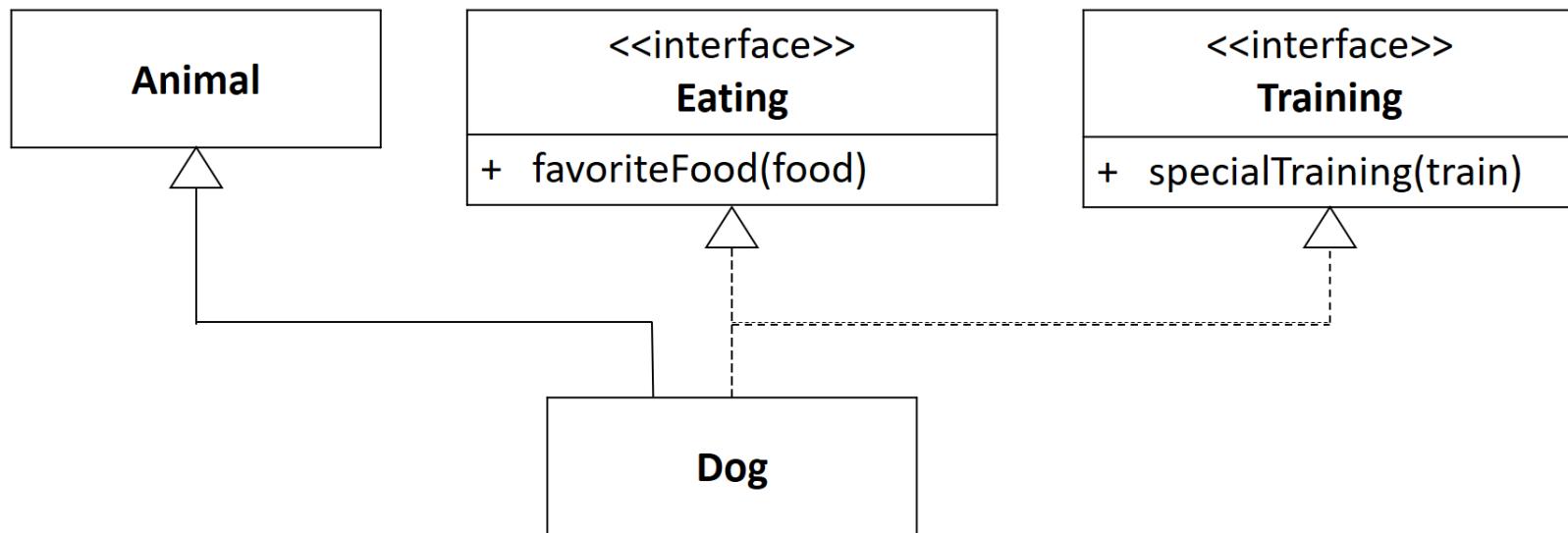
House
- bedroom
- restroom
- garage
+ setBedroom(int)
+ setRestroom(int)
+ setGarage(int)
+ getBedroom()
+ getRestroom()
+ getGarage()

Encapsulation

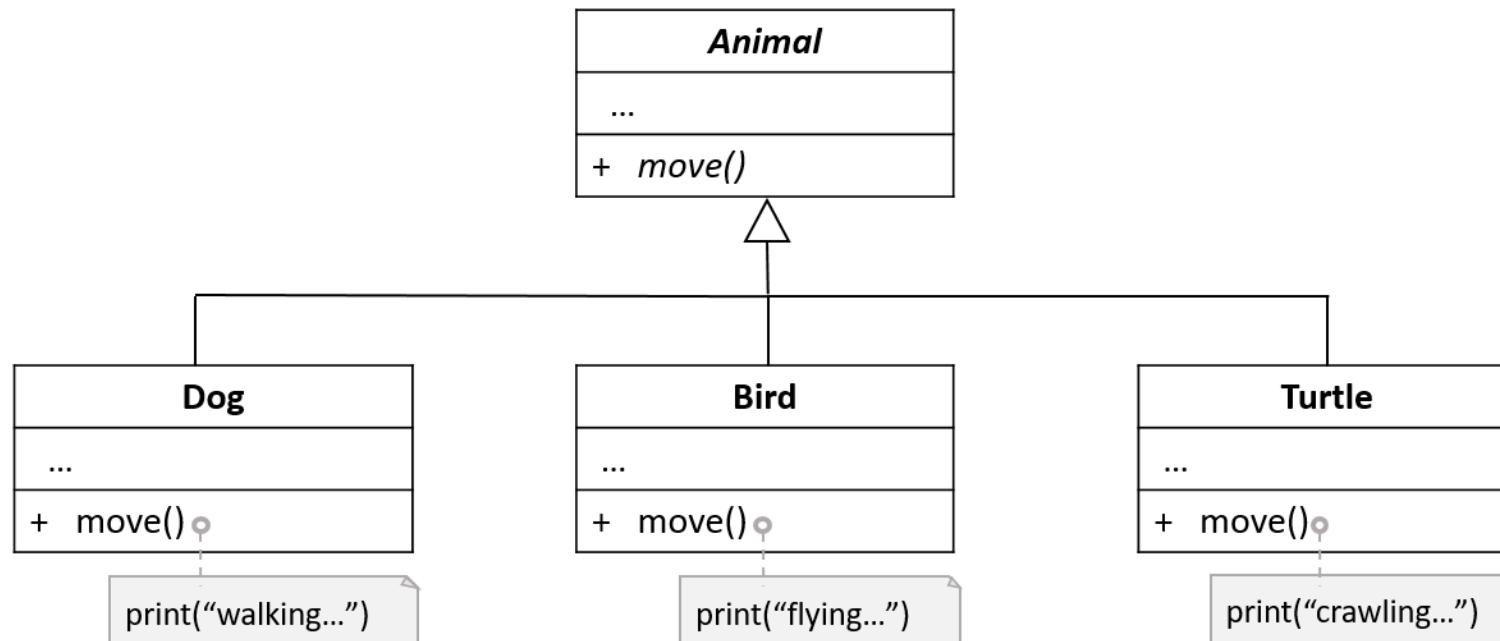
- private, protected
- interface, protocol
- abstract class/method



Inheritance



Polymorphism



```

#include<iostream>
#include <list>
using namespace std;
class Animal {
    string name;
public:
    Animal(string s) {
        name=s;
    }
    virtual ~Animal() {}
    virtual void move() = 0;
    string getName() {
        return name;
    }
};

class Dog: public Animal {
public:
    Dog(string s):Animal(s) {}
    ~Dog() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" walking..."<<endl;
    }
};

```

```

class Bird: public Animal {
public:
    Bird(string s):Animal(s) {}
    ~Bird() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" flying..."<<endl;
    }
};

class Turtle: public Animal {
public:
    Turtle(string s):Animal(s) {}
    ~Turtle() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" crawling..."<<endl;
    }
};

int main() {
    Animal *animal[3];
    animal[0]= new Dog("puifai");
    animal[1]= new Bird("bird bird");
    animal[2]= new Turtle("ninja turtle");

    for (int i=0; i<3; i++) {
        animal[i]->move();
    }
    for (int i=0; i<3; i++) {
        delete animal[i];
    }
}

```

puifai walking...
 bird bird flying...
 ninja turtle crawling...
 bye bye puifai
 bye bye bird bird
 bye bye ninja turtle

```
#include<iostream>
#include <list>
using namespace std;
class Animal {
    string name;
public:
    Animal(string s) {
        name=s;
    }
    virtual ~Animal() {}
    virtual void move() = 0;
    string getName() {
        return name;
    }
};

class Dog: public Animal {
public:
    Dog(string s):Animal(s) {}
    ~Dog() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" walking..."<<endl;
    }
};
```

```
class Bird: public Animal {
public:
    Bird(string s):Animal(s) {}
    ~Bird() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" flying..."<<endl;
    }
};

class Turtle: public Animal {
public:
    Turtle(string s):Animal(s) {}
    ~Turtle() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" crawling..."<<endl;
    }
};

int main() {
    list <Animal*> animal;
    animal.push_back(new Dog("puifai"));
    animal.push_back(new Bird("bird bird"));
    animal.push_back(new Turtle("ninja turtle"));

    for (Animal *a : animal) {
        a->move();
    }
    for (Animal *a : animal) {
        delete a;
    }
}
```

```
#include<iostream>
#include <list>
using namespace std;
class Animal {
    string name;
public:
    Animal(string s) {
        name=s;
    }
    virtual ~Animal() {}
    virtual void move() = 0;
    string getName() {
        return name;
    }
};

class Dog: public Animal {
public:
    Dog(string s):Animal(s) {}
    ~Dog() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" walking..."<<endl;
    }
};
```

```
class Bird: public Animal {
public:
    Bird(string s):Animal(s) {}
    ~Bird() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" flying..."<<endl;
    }
};

class Turtle: public Animal {
public:
    Turtle(string s):Animal(s) {}
    ~Turtle() {
        cout<<"bye bye "<<getName()<<endl;
    }
    void move() {
        cout<<getName()<<" crawling..."<<endl;
    }
};

int main() {
    list <Animal*> animal;
    animal.push_back(new Dog("puifai"));
    animal.push_back(new Bird("bird bird"));
    animal.push_back(new Turtle("ninja turtle"));

    for ( list<Animal*>::iterator it = animal.begin(); it != animal.end(); it++ ) {
        (*it)->move();
    }
    for ( list<Animal*>::iterator it = animal.begin(); it != animal.end(); it++ ) {
        delete (*it);
    }
}
```

```

using System;
using System.Collections.Generic;

abstract class Animal
{
    public string Name;
    public Animal(string n) => Name = n;
    ~Animal() => Console.WriteLine("Animal");
    public abstract void Move();
};

class Dog: Animal
{
    public Dog(string n) : base(n)
    { }
    ~Dog() => Console.WriteLine($"bye bye my dog {Name}");
    public override void Move() => Console.WriteLine($"{Name} walking...");
};

class Bird: Animal
{
    public Bird(string n) : base(n)
    { }
    ~Bird() => Console.WriteLine($"bye bye my bird {Name}");
    public override void Move() => Console.WriteLine($"{Name} flying...");
};

class Turtle: Animal
{
    public Turtle(string n) : base(n)
    { }
    ~Turtle() => Console.WriteLine($"bye bye my turtle {Name}");
    public override void Move() => Console.WriteLine($"{Name} crawling...");
};

```

```

class Program {

    static void Main(string[] args)
    {
        Console.WriteLine("c# Polymorphism");
        List<Animal> animals = new List<Animal>();

        animals.Add(new Dog("puifai"));
        animals.Add(new Bird("bird bird"));
        animals.Add(new Turtle("ninja turtle"));

        foreach (Animal a in animals)
        {
            a.Move();
        }
    }
}

```

```

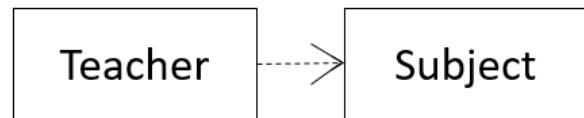
c# Polymorphism
puifai walking...
bird bird flying...
ninja turtle crawling...
bye bye my turtle ninja turtle
Animal
bye bye my dog puifai
Animal
bye bye my bird bird bird
Animal

```

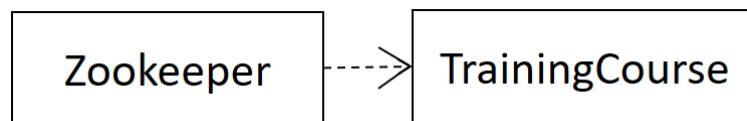
ความสัมพันธ์ระหว่างออบเจกต์

- **Dependency**

- ถ้ามีการเปลี่ยนแปลงเกิดขึ้นที่คลาสนึง อาจมีผลให้ต้องมีการเปลี่ยนแปลงอีกคลาสนึงด้วย
- ตัวอย่าง
 - อาจารย์ผู้สอน ขึ้นอยู่กับ วิชาที่เปิดสอน ถ้าเปลี่ยนวิชา ก็ต้องเปลี่ยนอาจารย์



- ถ้าเปลี่ยนหลักสูตรการอบรมสัตว์ เช่นจากลิงเป็นนกแก้ว ก็ต้องเปลี่ยน ผู้ดูแลสัตว์ที่เป็นคนอบรม

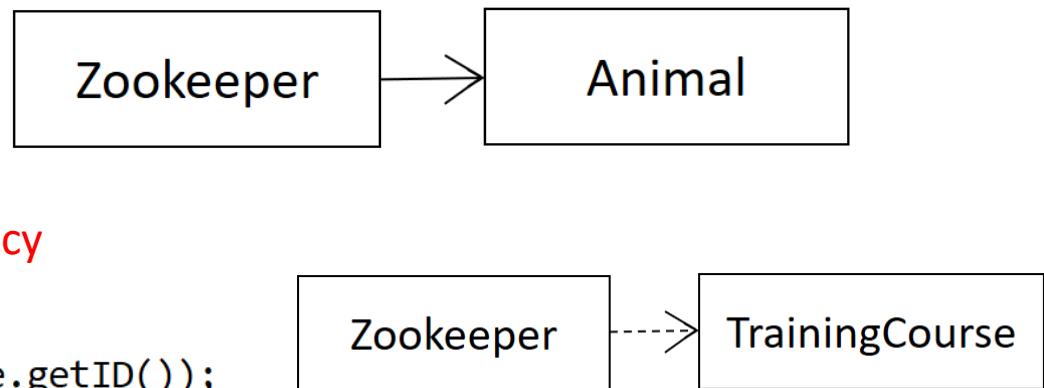


- สามารถทำ **dependency** ให้ลดลงได้โดยการใช้ **interface** หรือ **abstract class**

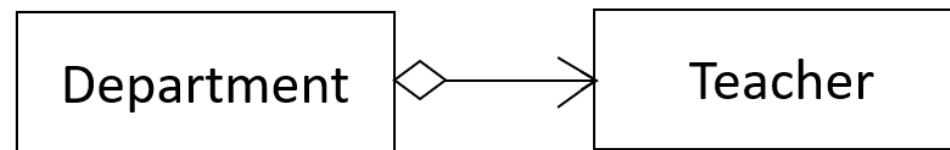
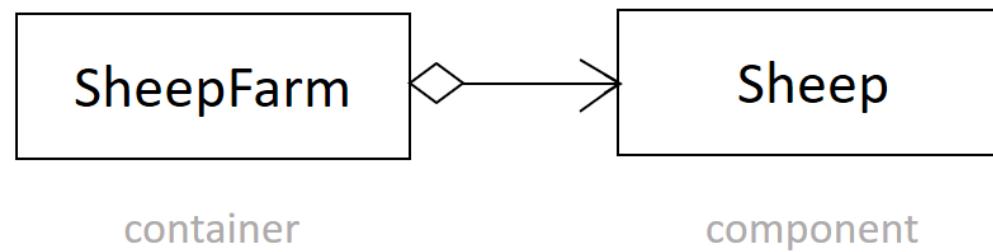
- Association

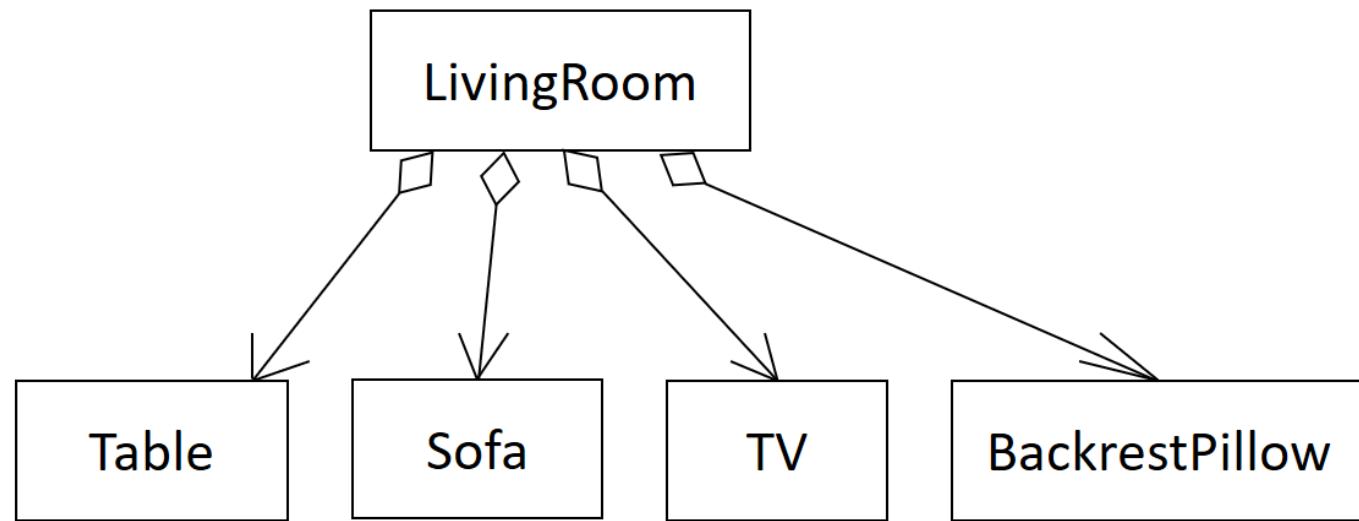
- การที่ออบเจกต์หนึ่งมีการใช้หรือมีการโต้ตอบกับอีกออบเจกต์หนึ่ง
- ความสัมพันธ์สามารถมีได้ทั้งสองทาง
- Association เป็นกรณีพิเศษของ Dependency
- Association จะมีลิงค์ถาวรระหว่างออบเจกต์ แต่ Dependency จะไม่มีการสร้างลิงค์ถาวร
- ตัวอย่าง Association

```
class Zookeeper {  
    string name;  
    Animal animal; // Association  
    public:  
        void train(TrainingCourse course) {  
            int previousLevel = animal.getLevel(course.getID());  
            animal.learn(course.level(previousLevel+1));  
        }  
};
```

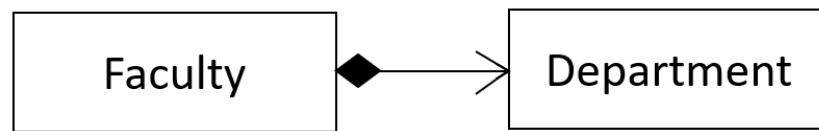
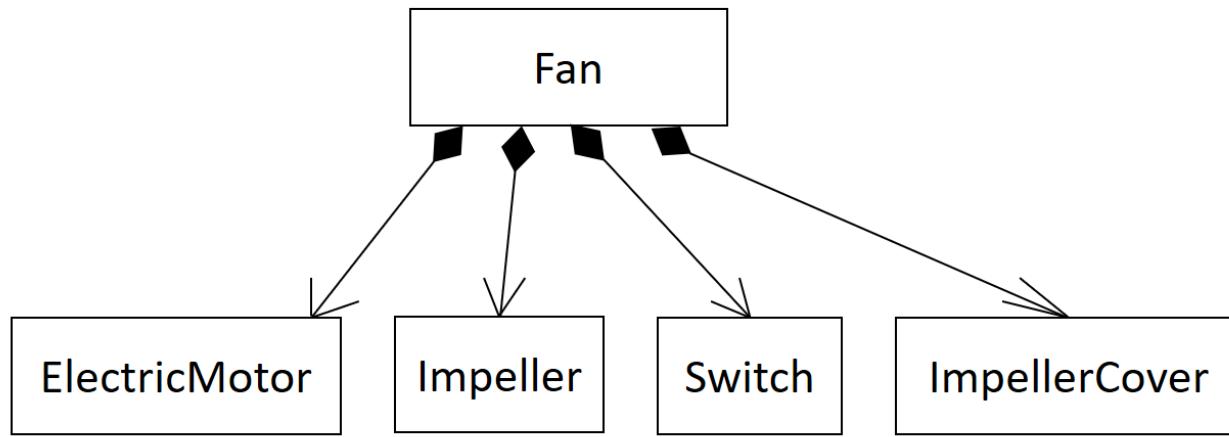


- Aggregation
 - one-to-many, many-to-many, whole-part relations
 - An object has a set of other objects
 - container, collection

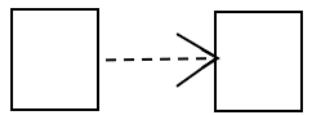




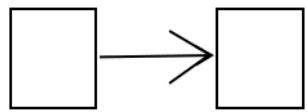
- Composition
 - One object is composed of one or more other objects



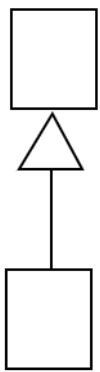
Choose composition over inheritance



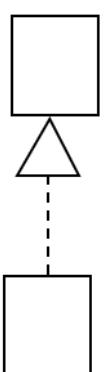
Dependency



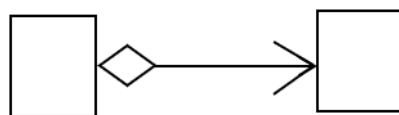
Association



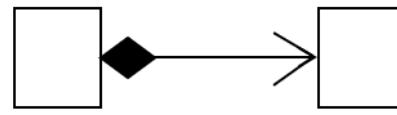
Inheritance



Implementation



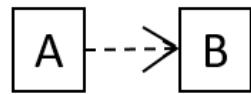
Aggregation



Composition

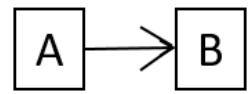
- Dependency

- ถ้า B เปลี่ยน A จะมีผลกระทบตามมาด้วย



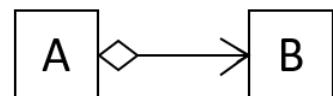
- Association

- A รู้จัก B , A ขึ้นกับ B



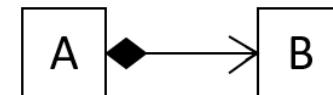
- Aggregation

- A รู้จัก B , A ประกอบด้วย B , A ขึ้นกับ B



- Composition

- A รู้จัก B , A ประกอบด้วย B , A ขึ้นกับ B , A จัดการช่วงชีวิตของ B

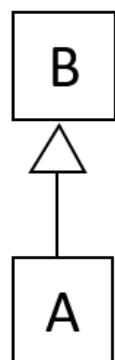


- Implementation

- A นิยามเมธอดที่ประกาศใน Interface B , A สามารถถือเป็น B , A ขึ้นกับ B



implementation



inheritance

- Inheritance

- A รับทรัพย์สินจาก B และ A สามารถขยายความสามารถเพิ่มได้
- A สามารถถือเป็น B , A ขึ้นกับ B

หลักการในการออกแบบซอฟต์แวร์

- ลักษณะของการออกแบบที่ดี
 - Code reuse
 - ใช้ pattern มากกว่า framework เนื่องจากขนาดเล็กกว่า และเป็นนามธรรม (abstract) มากกว่า
 - การ reuse มี 3 ระดับ
 - ตัวสุด
 - class
 - สูงสุด
 - Framework
 - กลาง
 - Pattern
 - Extensibility
 - สามารถเปลี่ยนแปลงได้ง่าย

หลักการในการออกแบบซอฟต์แวร์

- หลักการในการออกแบบ
 - Encapsulation
 - สิ่งที่มีการเปลี่ยนแปลงบ่อยๆ กับ สิ่งที่ไม่เคยมีการเปลี่ยนแปลง ให้แยกออกจากกัน
 - method level
 - class level
 - ใช้ Interface
 - ใช้ Composition แทนการใช้ Inheritance

Encapsulation

- method level

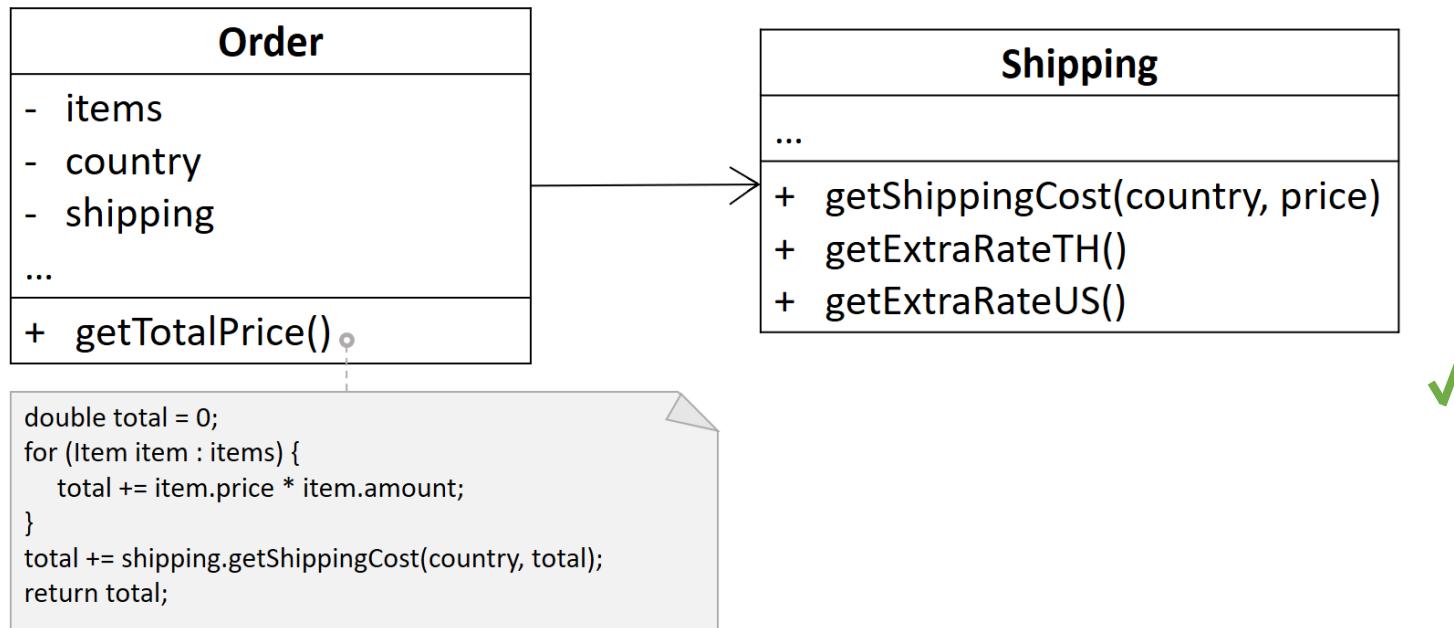
```
double getTotalPrice(list<Item> items) {  
    double total = 0;  
    for (Item item : items) {  
        total += item.price * item.amount;  
    }  
    if (total < 5000 && total >= 3500)  
        total += 180;  
    else if (total < 3500)  
        total += 360;  
    return total;  
}
```

```
double getTotalPrice(list<Item> items) {  
    double total = 0;  
    for (Item item : items) {  
        total += item.price * item.amount;  
    }  
    total += getShippingCost(total);  
    return total;  
}  
double getShippingCost(double price) {  
    if (price >= 5000)  
        return 0;  
    else if (price < 5000 && price >= 3500)  
        return 180;  
    else if (price < 3500)  
        return 360;  
}
```

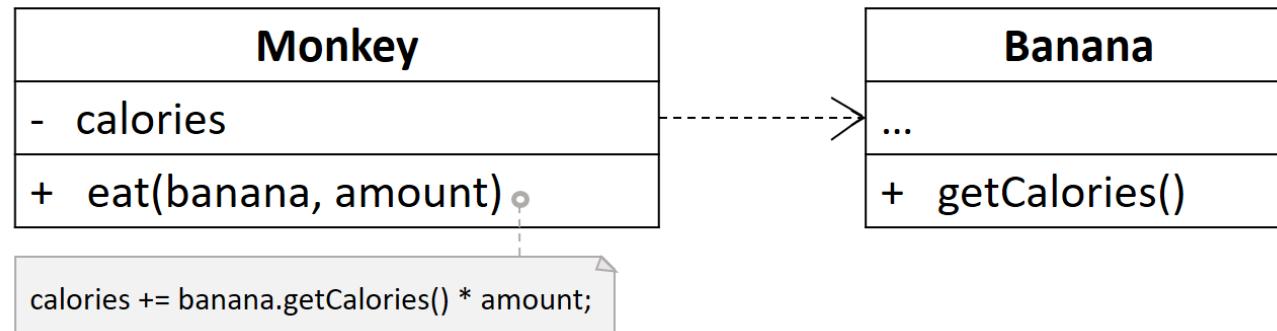
- class level

Order
- items
- country
...
+ getTotalPrice()
+ getShippingCost()

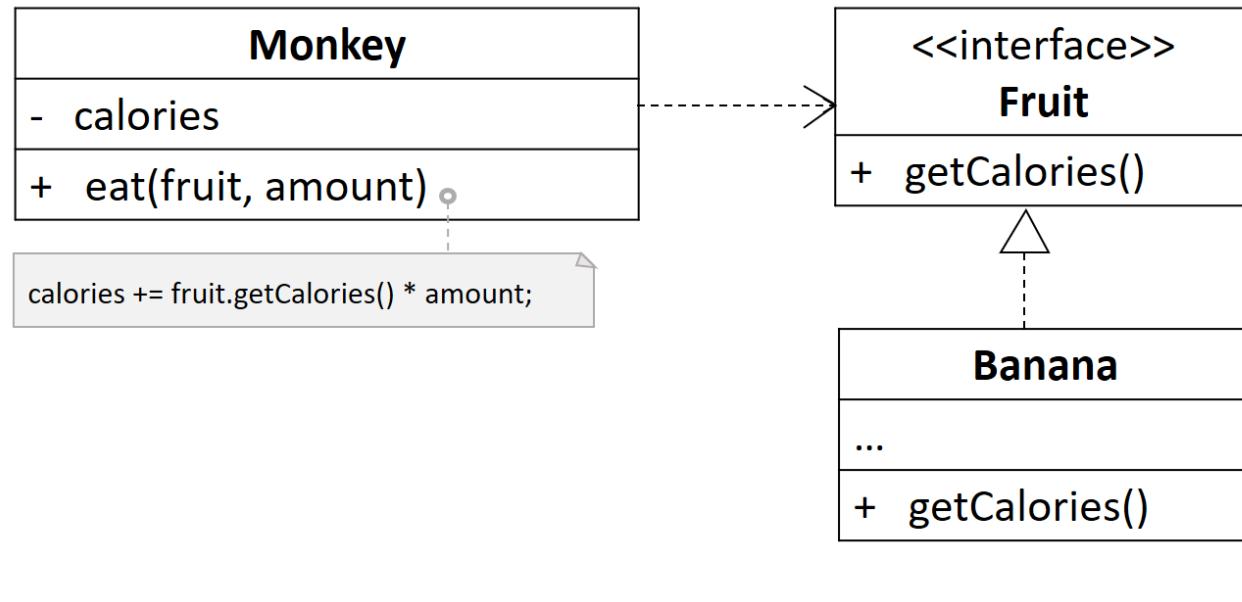
- class level

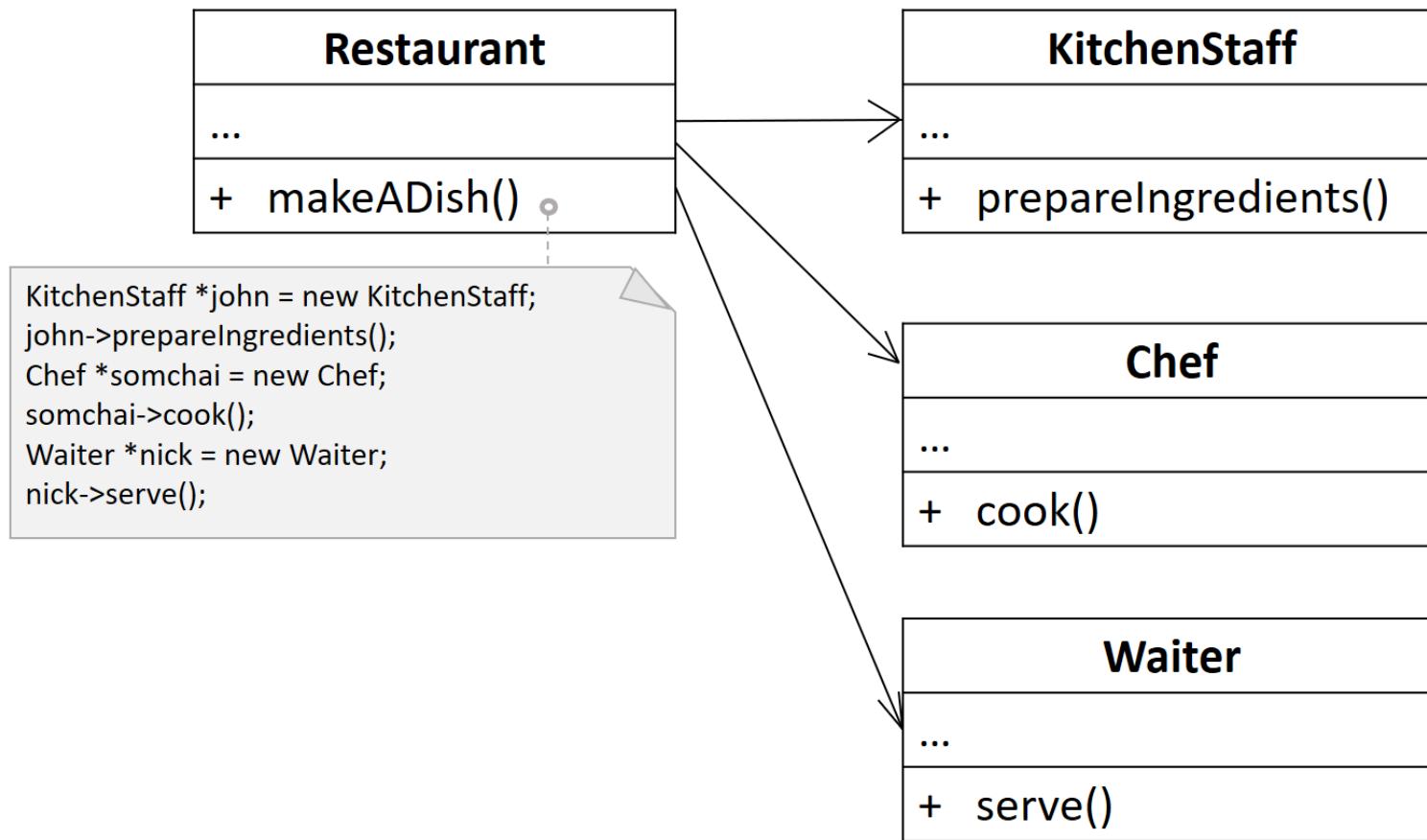


接口

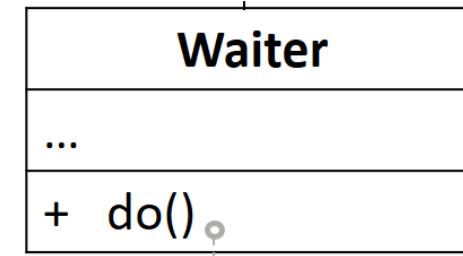
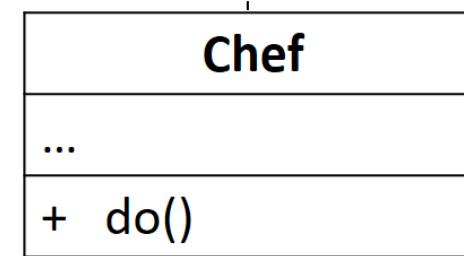
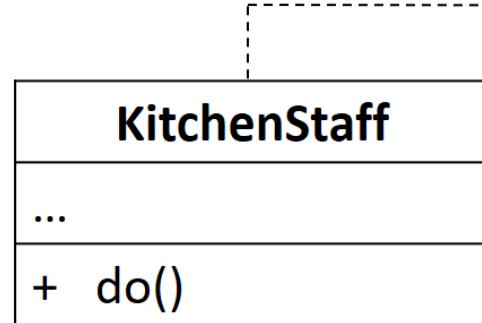
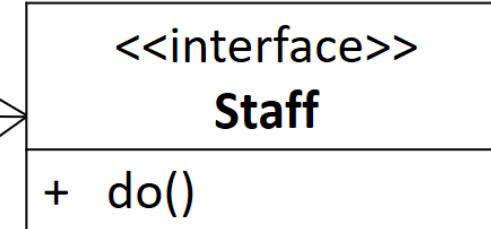
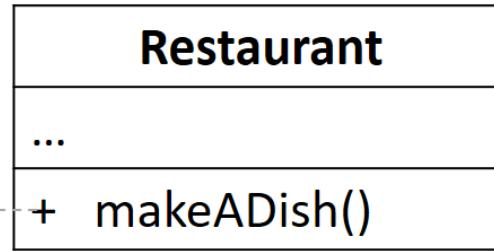


接口

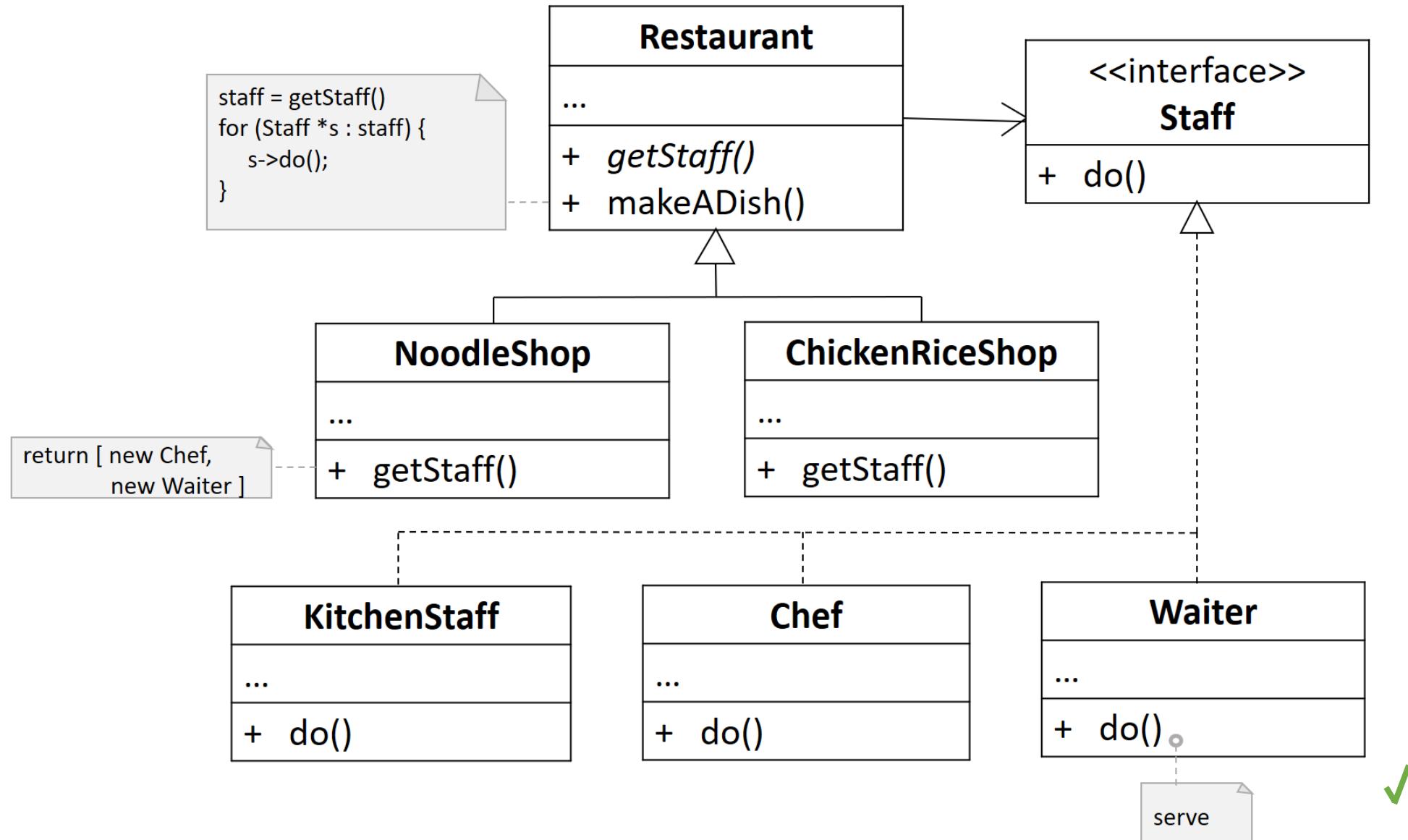




```
Staff *staff[3] = { new KitchenStaff,  
                   new Chef,  
                   new Waiter };  
  
for (int i=0; i<3; i++) {  
    staff[i]->do();  
}
```



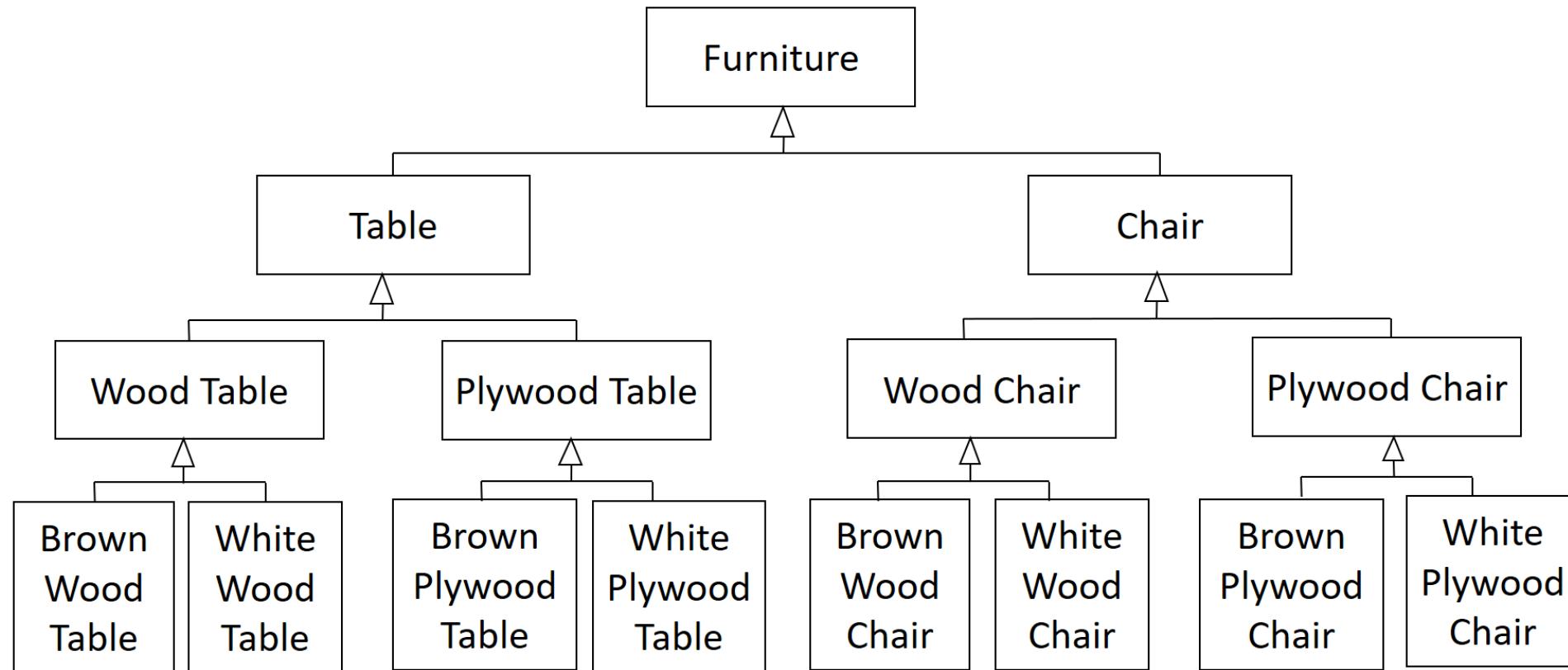
serve

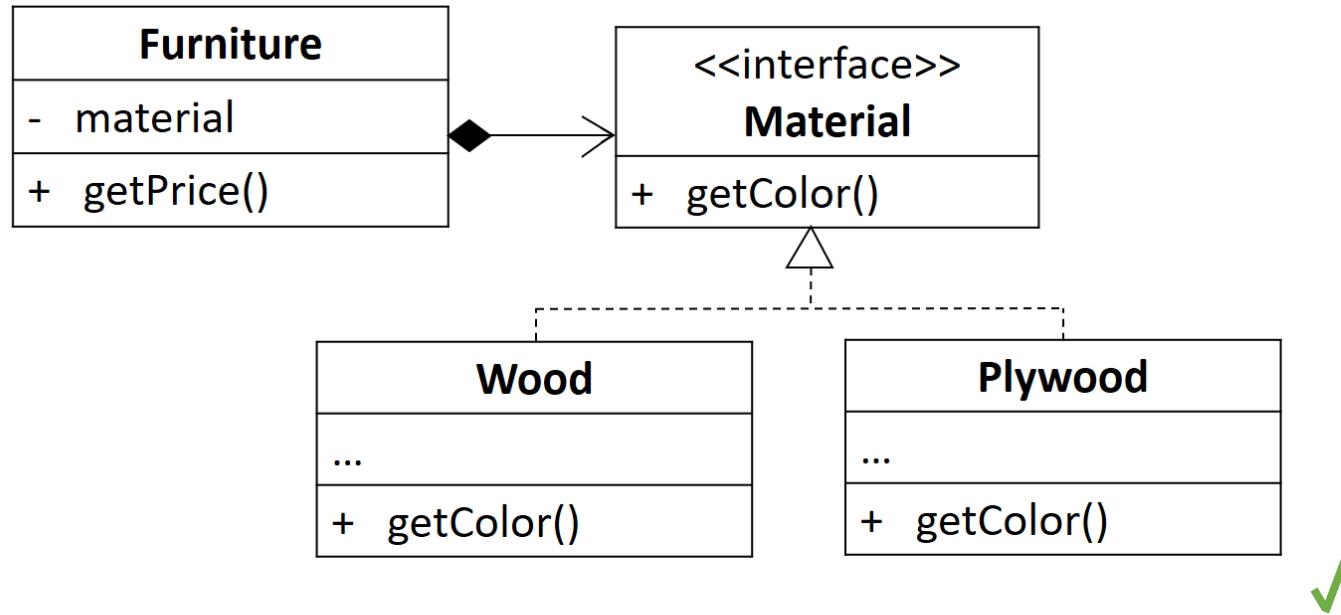


Factory Method Pattern

ใช้ Composition แทนการใช้ Inheritance

- ข้อเสียของ Inheritance
 - คลาสลูกต้องสร้าง **abstract method** ของคลาสมแม่ทุกเมธอด ถึงแม้ว่าจะไม่ได้ใช้ก็ตาม
 - โควต้าที่ต้องมีพัฒนาร่วมสอดคล้องกับคลาสมแม่
 - ขัดต่อหลักการซ่อนสารสนเทศของคลาสมแม่
 - ถ้ามีการเปลี่ยนแปลงที่คลาสมแม่ อาจมีผลกระทบต่อกลุ่มลูก
 - กรณีที่ Inheritance มีมากกว่า 1 มิติ อาจต้องสร้างคลาสใหม่ที่เกิดจากการรวมกันของคลาสเดิมอีกจำนวนมาก
- Composition
 - Has-a relationship เช่น รถยนต์ มี เครื่องยนต์
 - Aggregation (ไม่ได้ควบคุม lifecycle) เช่น รถยนต์ มี คนขับ





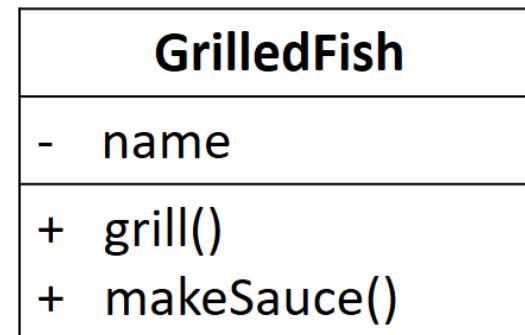
Strategy Pattern

SOLID Principles

- Robert Martin, “Agile Software Development, Principles, Patterns, and Practices”
- SOLID
 - Single Responsibility Principle
 - Open/Closed Principle
 - Liskov Substitution Principle
 - Interface Segregation Principle
 - Dependency Inversion Principle
- การใช้ SOLID อาจสร้างความเสียหายมากกว่าการไม่ได้ใช้
 - โค้ดซับซ้อนมากขึ้น
 - แต่ถ้าใช้อย่างเหมาะสมก็จะดีมาก

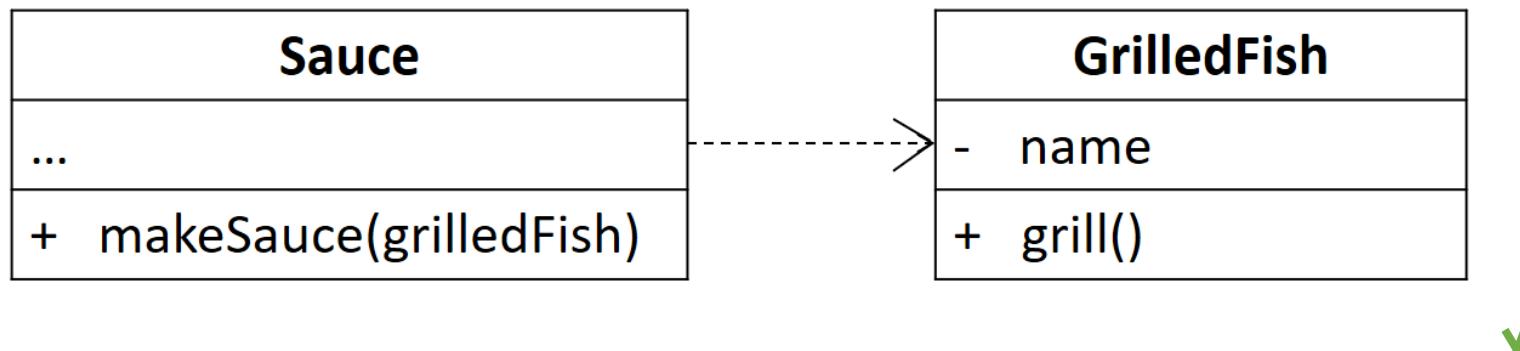
Single Responsibility Principle

- ความมีแค่เหตุผลเดียวในการเปลี่ยนแปลงคลาส



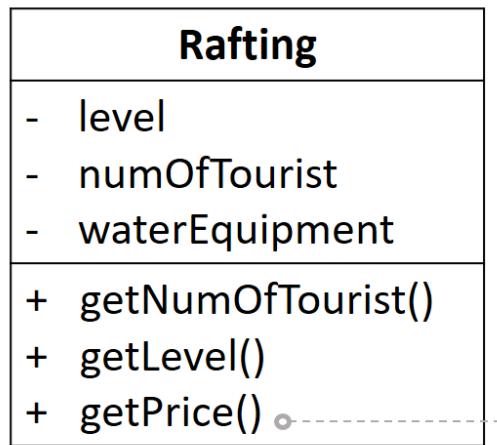
Single Responsibility Principle

- ควรมีแค่เหตุผลเดียวในการเปลี่ยนแปลงคลาส



Open/Closed Principle

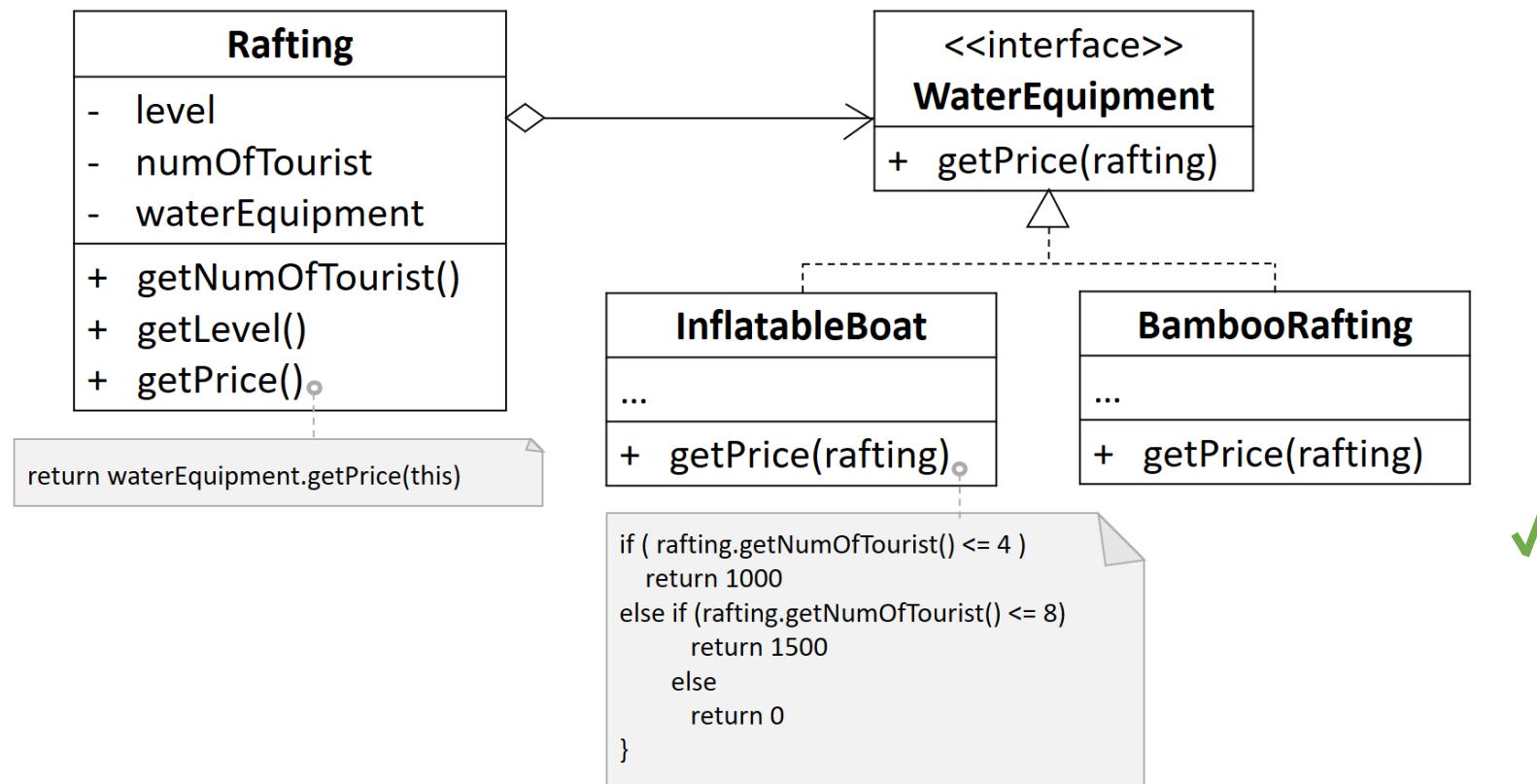
- คลาสควรจะเปิด (**open**) ให้สามารถเพิ่มขยายได้ แต่ก็ควรจะปิด (**close**) เพื่อไม่ให้มีการแก้ไข



```
If ( waterEquipment == "ແພີ້ນໆ" ) {  
    if ( getLevel() >= 1 && getLevel() <= 3 ) {  
        if ( getNumOfTourist() <= 2 )  
            return 400  
        else  
            return 600  
    } else return 0  
}  
else if ( waterEquipment == "ເຈືອຍາງ" ) {  
    if ( getNumOfTourist() <= 4 )  
        return 1000  
    else if (getNumberOfTourist() <= 8)  
        return 1500  
    else  
        return 0  
}
```

Open/Closed Principle

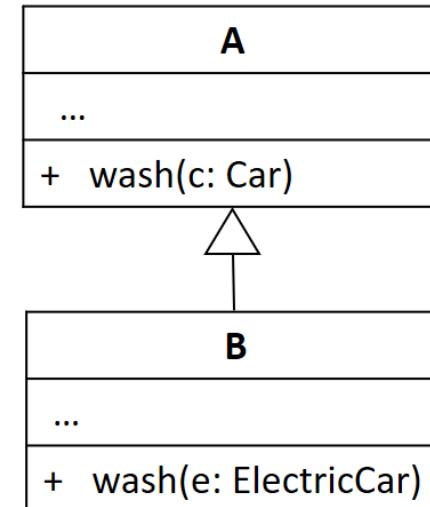
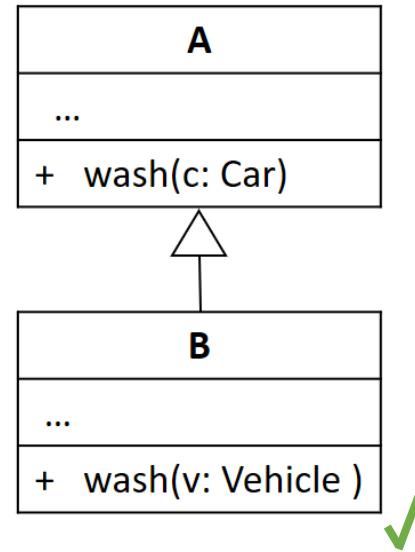
- คลาสควรจะเปิด (open) ให้สามารถเพิ่มขยายได้ แต่ก็ควรจะปิด (close) เพื่อไม่ให้มีการแก้ไข



Strategy Pattern

Liskov Substitution Principle

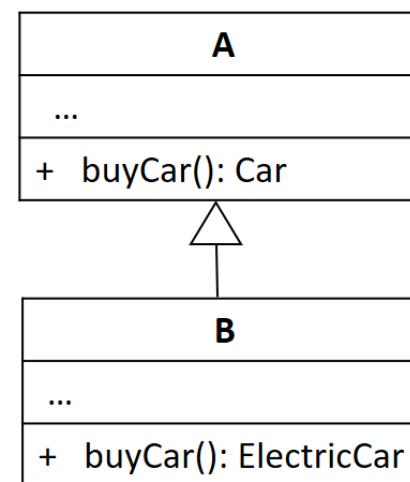
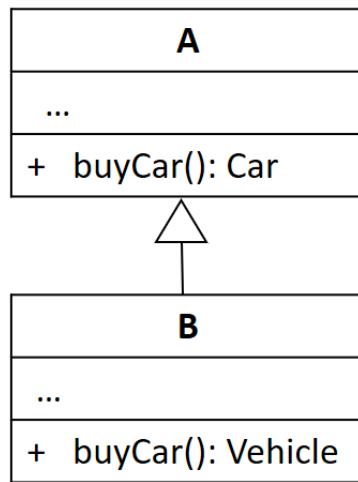
- อีกบเจกต์ของคลาสลูกต้องสามารถแทนที่อีกบเจกต์ของคลาสแม่ได้
- หลักการการแทนที่
 - ชนิดของพารามิเตอร์ของเมธอดของคลาสลูกควรจะต้อง **match** หรือ **more abstract** มากกว่าชนิดของพารามิเตอร์ของเมธอดของคลาสแม่



Liskov Substitution Principle

- หลักการการแทนที่

2. ชนิดของการวีเทิร์นของเมธอดของคลาสลูกควรจะต้อง **match** หรือเป็น **subtype** ของชนิดของการวีเทิร์นของเมธอดของคลาสแม่

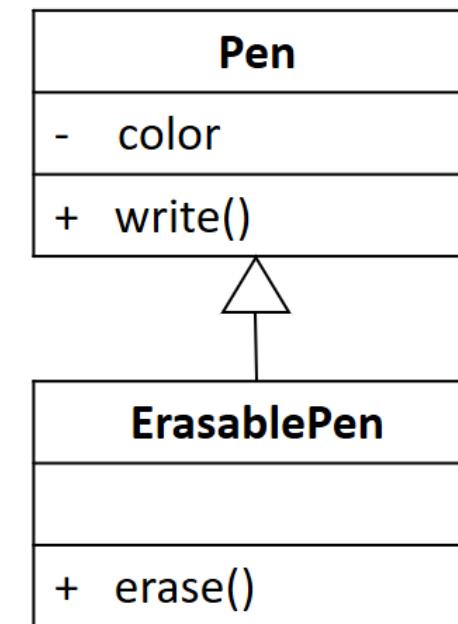
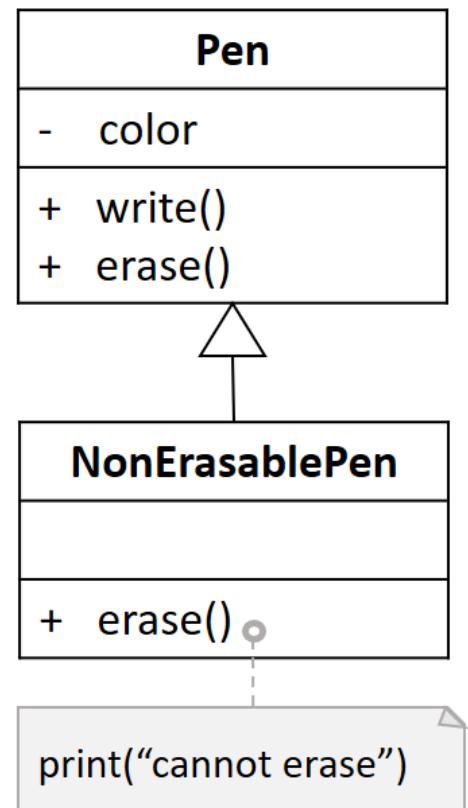


Liskov Substitution Principle

- หลักการการแทนที่
 - เมธอดของคลาสลูกไม่ควรโยนชนิดของความผิดปกติ (**exception**) ที่ซึ่งเมธอดของคลาสแม่ไม่ได้โยน
 - คลาสลูกไม่ควรทำ **pre-condition** ให้เข้มงวดมากขึ้นกว่าคลาสแม่
 - เช่น พารามิเตอร์ของเมธอดของคลาสแม่เป็น **int** ถ้าคลาสลูกทำการอโวร์โอดแล้วกำหนดให้เป็นค่าบวกเท่านั้น โดยที่ถ้าเป็นค่าลบจะโยนความผิดปกติอกรมา กรณีนี้จะเกิดปัญหา...
 - คลาสลูกไม่ควรทำ **post-condition** ให้อ่อนกว่าคลาสแม่
 - เช่น เมธอดของคลาสแม่มีการทำงานกับฐานข้อมูล และปิดฐานข้อมูลทุกฐานก่อนการรีเทิร์น ถ้าคลาสลูกเปลี่ยนเป็นไม่ปิด เพื่อจะได้ใช้งานฐานข้อมูลได้อีกเรื่อยๆ โดยไม่ต้องสร้างการเชื่อมต่อใหม่ กรณีนี้จะเกิดปัญหา...
 - คุณลักษณะของคลาสแม่ควรเป็นอย่างไร ก็ควรจะเป็นแบบนั้น
 - อาจใช้วิธีการเพิ่มพิลด์และเมธอดในคลาสลูก เพื่อที่จะได้ไม่ทำให้กระทบกับสมาชิกที่มีอยู่แล้วในคลาสแม่
 - คลาสลูกไม่ควรไปเปลี่ยนค่าของพิลด์ที่เป็น **private** ของคลาสแม่

Liskov Substitution Principle

```
foreach (p in pens)  
    if (!p instanceof NonErasablePen)  
        p.erase()
```



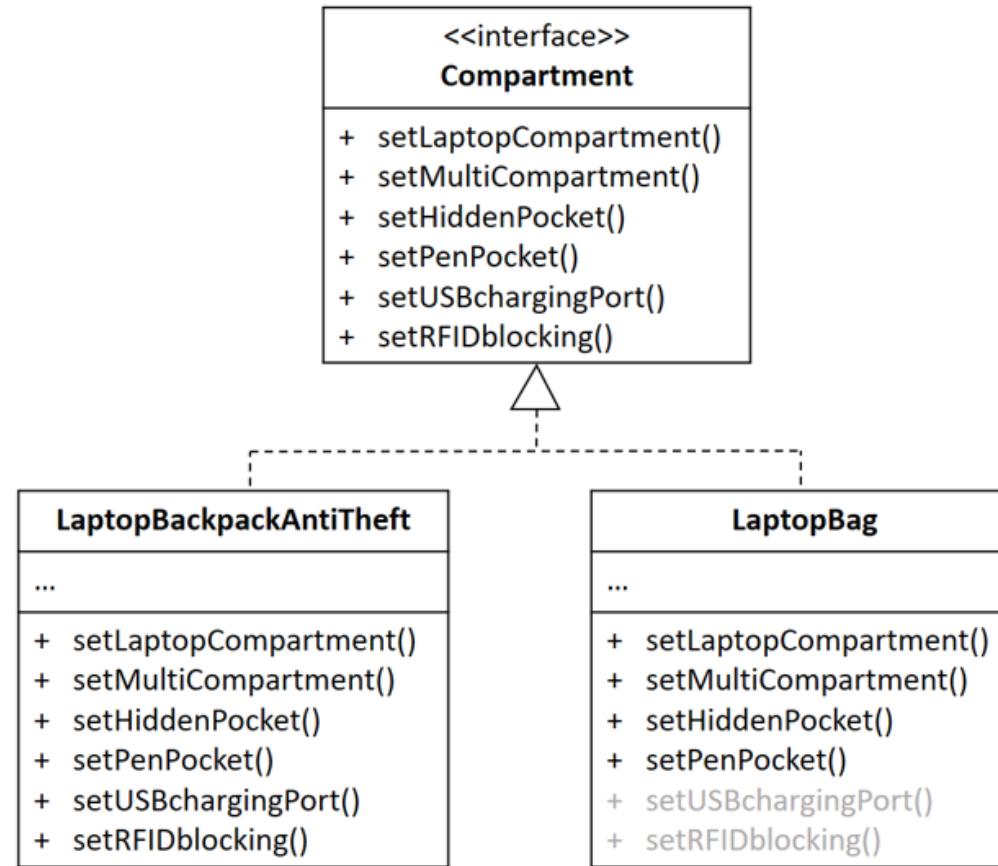
```
foreach (p in ErasablePen)  
    p.erase()
```



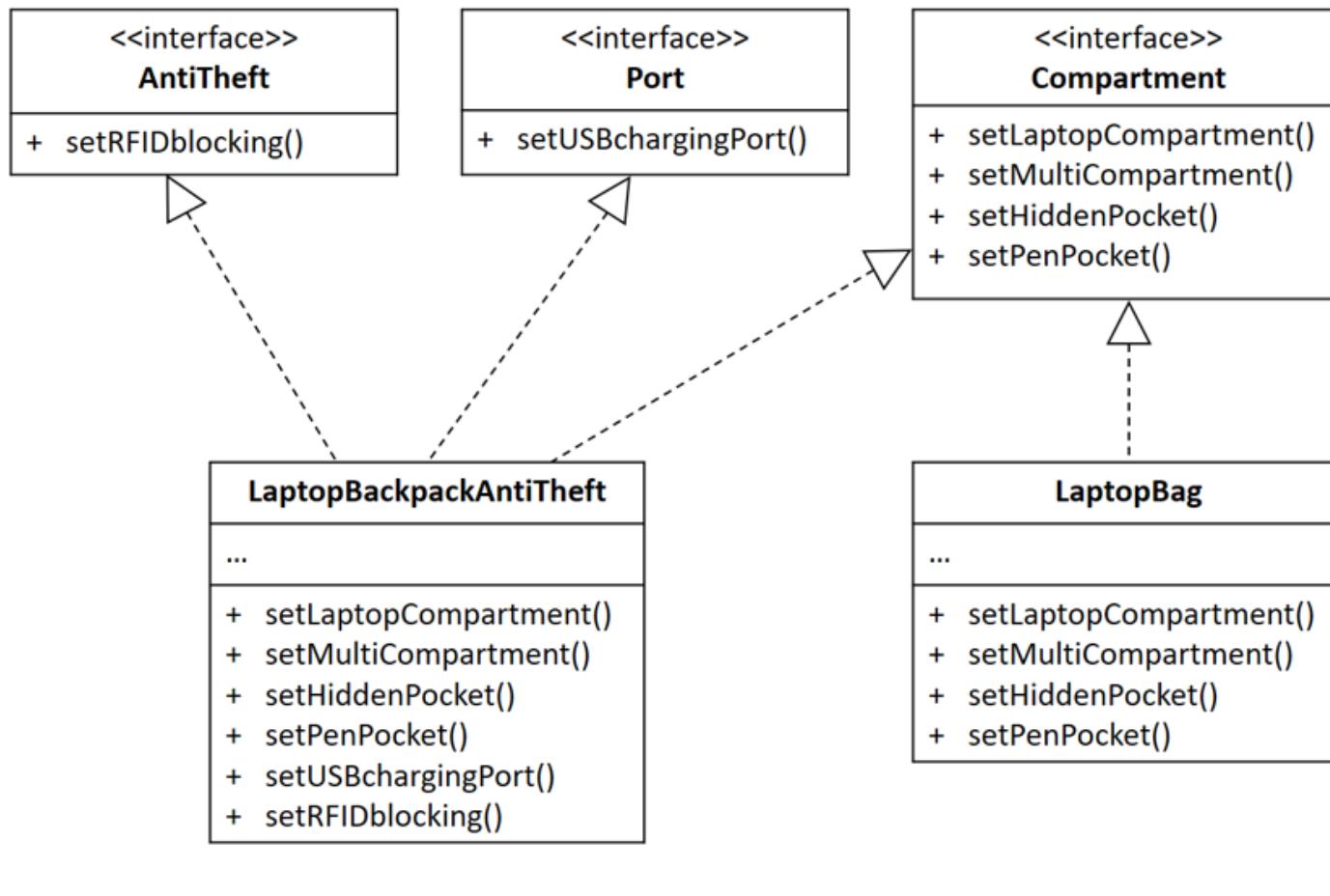
Interface Segregation Principle

- โค้ดไม่ควรที่จะต้องสร้างเมธอดที่ตนเองไม่ได้ใช้
- หลีกเลี่ยง “fat” interface

Interface Segregation Principle



Interface Segregation Principle



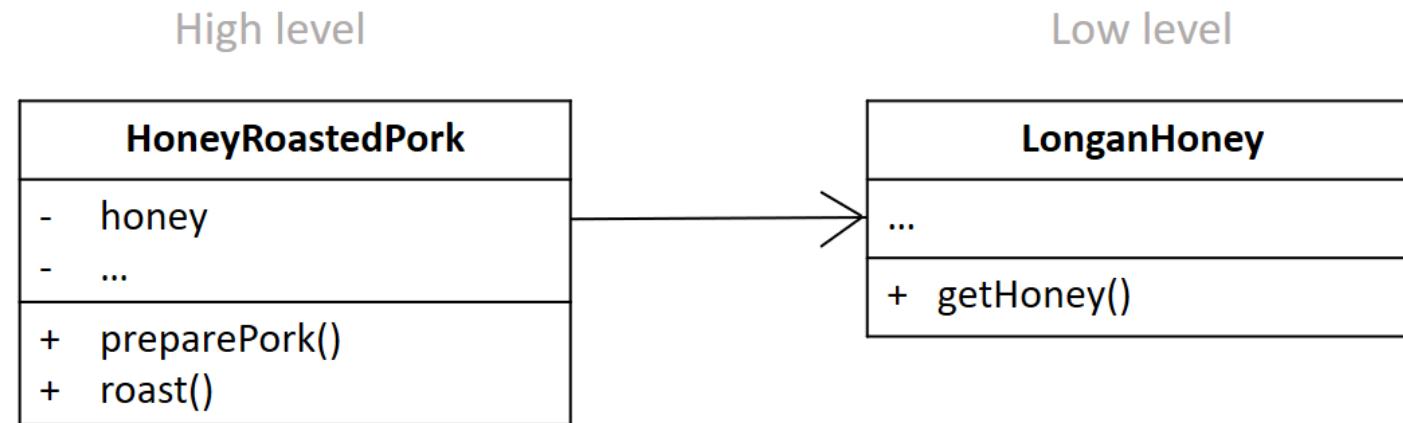
Dependency Inversion Principle

- คลาสมี 2 ระดับ
 - Low level class
 - ทำงานพื้นฐานต่างๆ เช่น การรับส่งข้อมูลผ่านเครือข่าย การเชื่อมต่อฐานข้อมูล เป็นต้น
 - High level class
 - ทำงานในระดับ business logic
 - ควบคุมสั่งการให้คลาสระดับล่างทำงานต่างๆ
- คลาสที่อยู่ระดับสูงกว่า (**high level class**) ไม่ควรขึ้นกับคลาสที่อยู่ในระดับต่ำกว่า (**low level class**)
- คลาสทั้งสองระดับควรขึ้นอยู่กับ **Abstraction**
- **Abstraction** ไม่ควรมีการกำหนดรายละเอียด

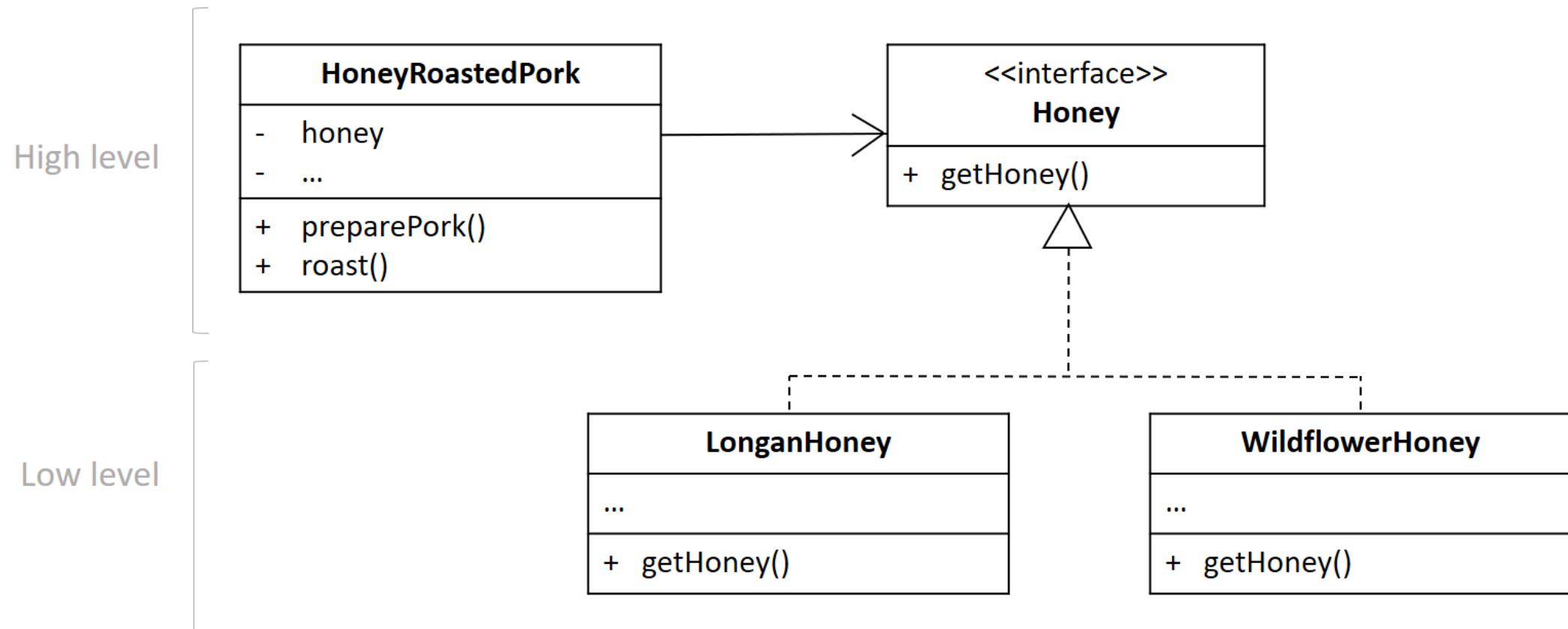
Dependency Inversion Principle

- เดิม: ออกแบบคลาสระดับต่ำกว่าก่อนแล้วค่อยออกแบบคลาสระดับสูงกว่า กรณีนี้คลาสที่อยู่ระดับสูงกว่า จะขึ้นอยู่กับคลาสที่อยู่ระดับต่ำกว่า ✗
- ข้อแนะนำ:
 - กำหนด **interface** ให้กับพังก์ชันในระดับ **low level** ที่มีคลาสระดับ **high level** ขึ้นอยู่กับมัน
 - เช่น คลาสระดับ **high level** ควรเรียกใช้เมธอด `readFile(name)` แทนที่จะเรียกใช้ `open(file)`, `read()`, `close(file)`
 - คลาสระดับ **high level** จะขึ้นกับ **interface** นั้น
 - คลาสในระดับ **low level** หลังจาก **implement** อินเตอร์เฟส ก็จะเปลี่ยนมาขึ้นกับคลาสในระดับ **high level** แทน

Dependency Inversion Principle



Dependency Inversion Principle



Catalog of Design Patterns

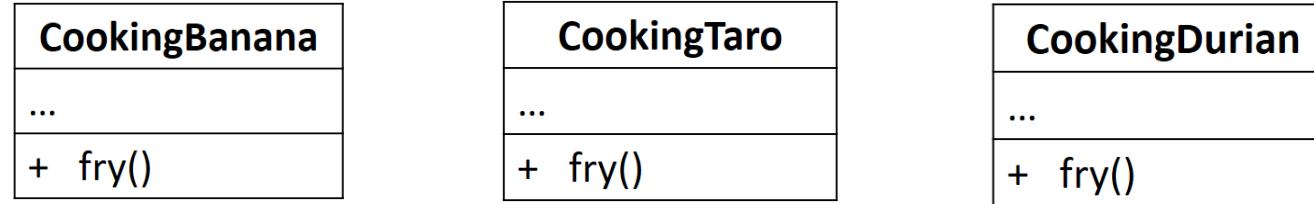
Creational Design Patterns

Factory Method

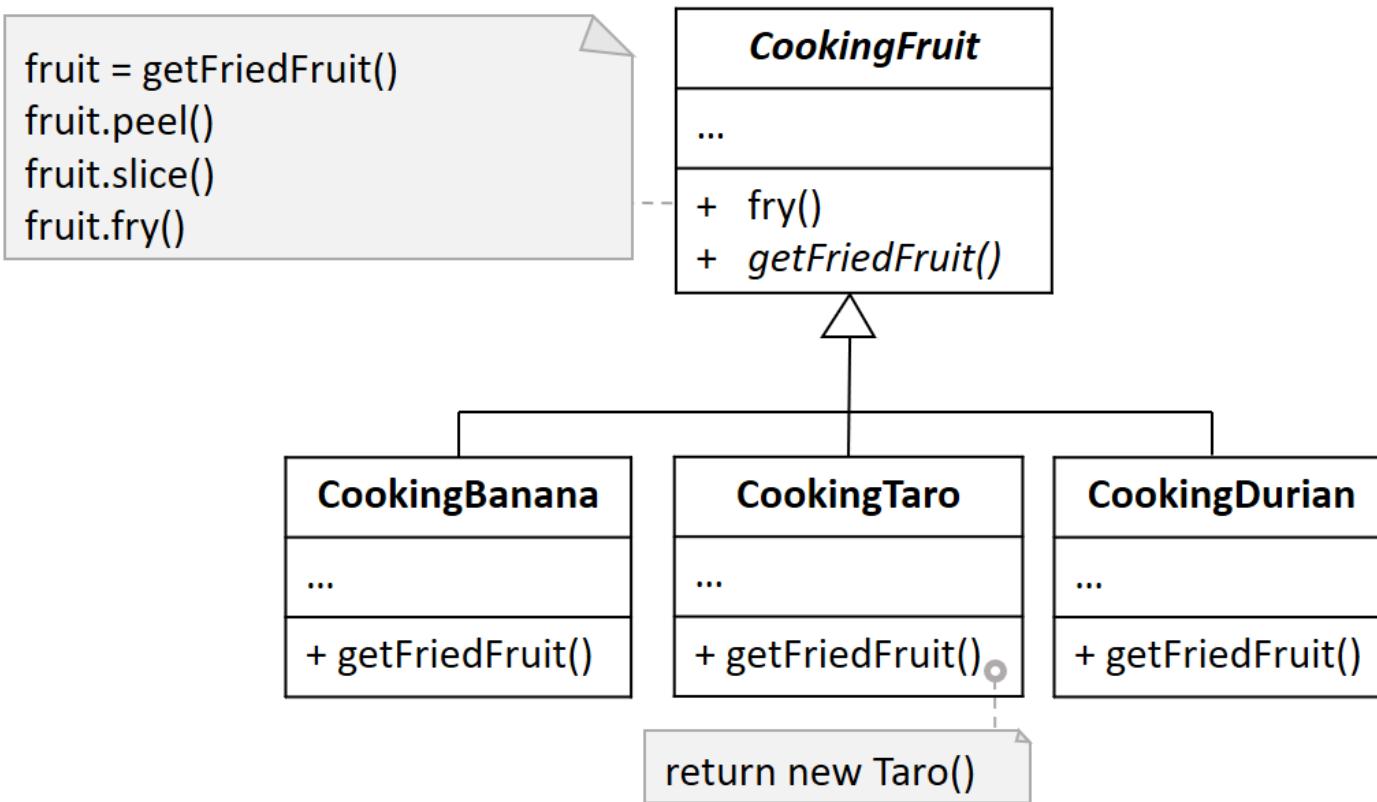
Factory Method หรือ Virtual Constructor

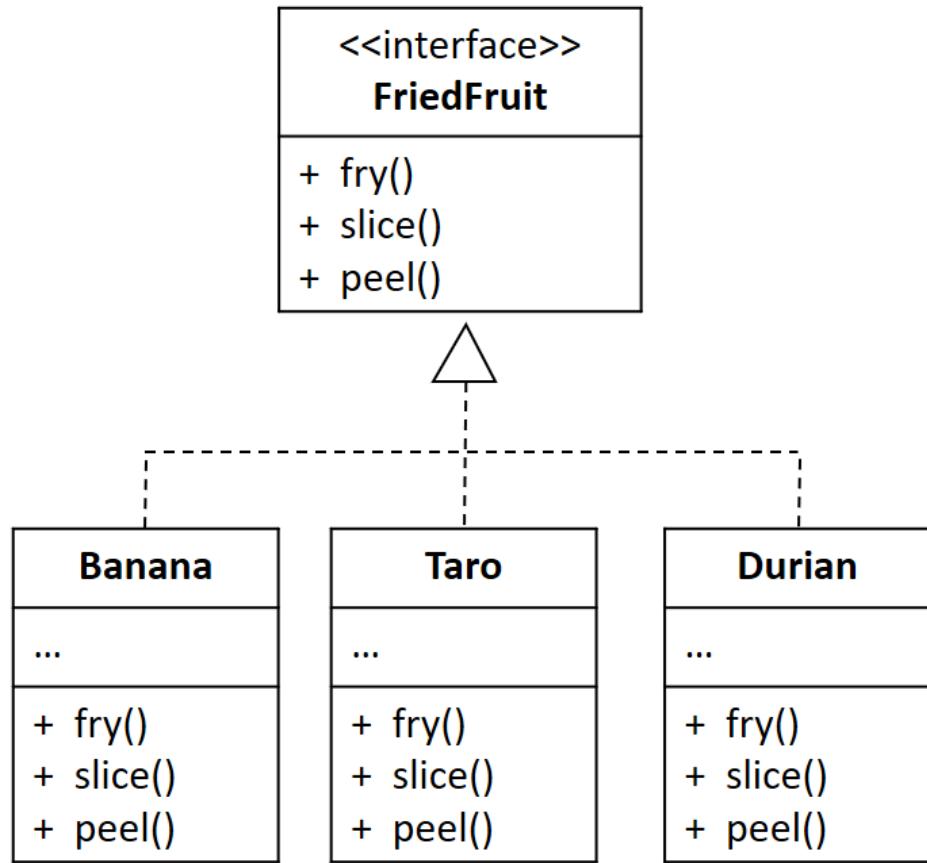
- จัดเตรียมเมธอดสำหรับการสร้างอ็อบเจกต์ในคลาสแม่ และยอมให้คลาสลูกสามารถเปลี่ยนชนิดของอ็อบเจกต์ที่จะถูกสร้างได้

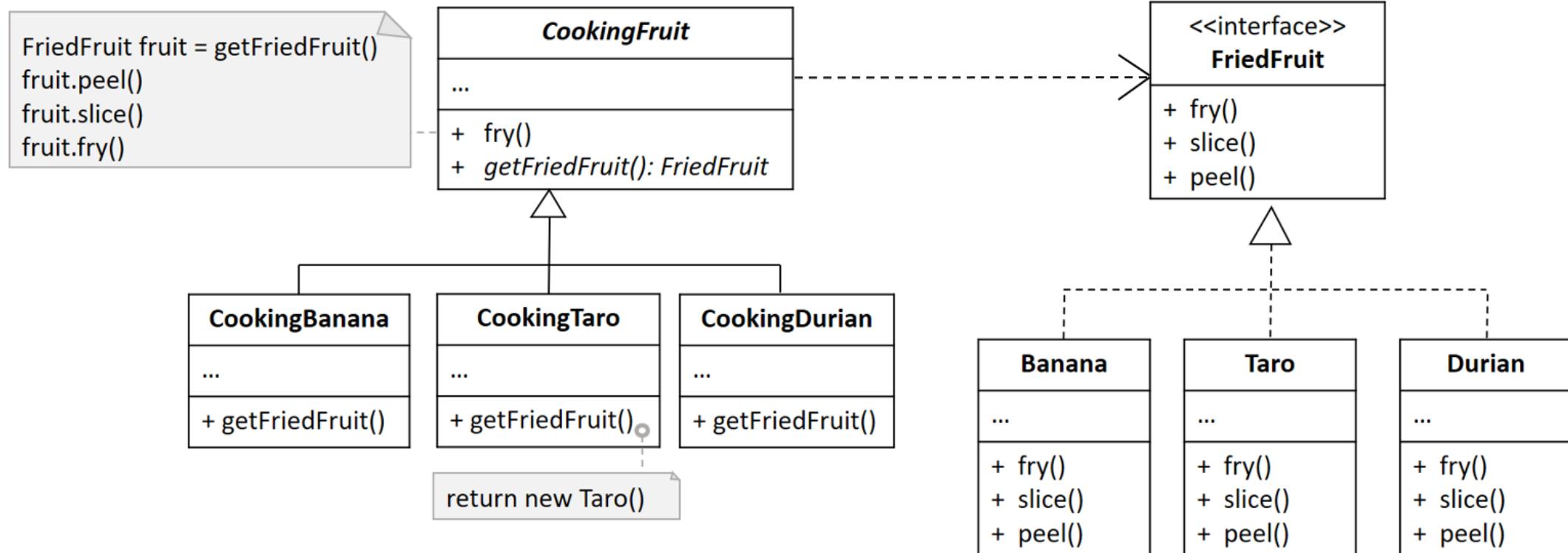
- ตัวอย่าง



- เดิมขายกล้วยทอด ขายดี๊ไปขายເຝືອทอด ทุเรียนทอดตามมา
- ให้เรียกใช้ **factory method** ในการสร้างอ็อบเจกต์ แทนที่จะสร้างอ็อบเจกต์โดยตรง
- ใน **factory method**, การสร้างยังคงใช้ **new** เหมือนเดิม หรือ จะรีเทิร์นอ็อบเจกต์ที่มีอยู่แล้วก็ได้
- เราเรียกอ็อบเจกต์ที่รีเทิร์นจาก **factory method** ว่า **product**
- โค้ดที่เรียกใช้ **factory method** เราเรียกว่า **client code**







```
#include <iostream>
using namespace std;

class FriedFruit {
public:
    virtual ~FriedFruit() {}
    virtual string fry() = 0;
    virtual string slice() = 0;
    virtual string peel() = 0;
};

class Banana: public FriedFruit {
public:
    string fry() { return "frying bananas"; }
    string slice() { return "slicing bananas"; }
    string peel() { return "peeling bananas"; }
};

class Taro: public FriedFruit {
public:
    string fry() { return "frying taro"; }
    string slice() { return "slicing taro"; }
    string peel() { return "peeling taro"; }
};

class Durian: public FriedFruit {
public:
    string fry() { return "frying durian"; }
    string slice() { return "slicing durian"; }
    string peel() { return "peeling durian"; }
};
```

```
class CookingFruit {
public:
    virtual ~CookingFruit() {};
    virtual FriedFruit* getFriedFruit() = 0;
    string fry() {
        FriedFruit* fruit = this->getFriedFruit();
        string result = "Cooking Fruit...\n";
        result+=fruit->peel() + "\n";
        result+=fruit->slice() + "\n";
        result+=fruit->fry() + "\n";
        delete fruit;
        return result;
    }
};

class CookingBanana: public CookingFruit {
public:
    FriedFruit* getFriedFruit() {
        return new Banana();
    }
};

class CookingTaro: public CookingFruit {
public:
    FriedFruit* getFriedFruit() {
        return new Taro();
    }
};
```

```

class CookingDurian: public CookingFruit {
public:
    FriedFruit* getFriedFruit() {
        return new Durian();
    }
};

void client(CookingFruit& cook) {
    cout << cook.fry() << endl;
}

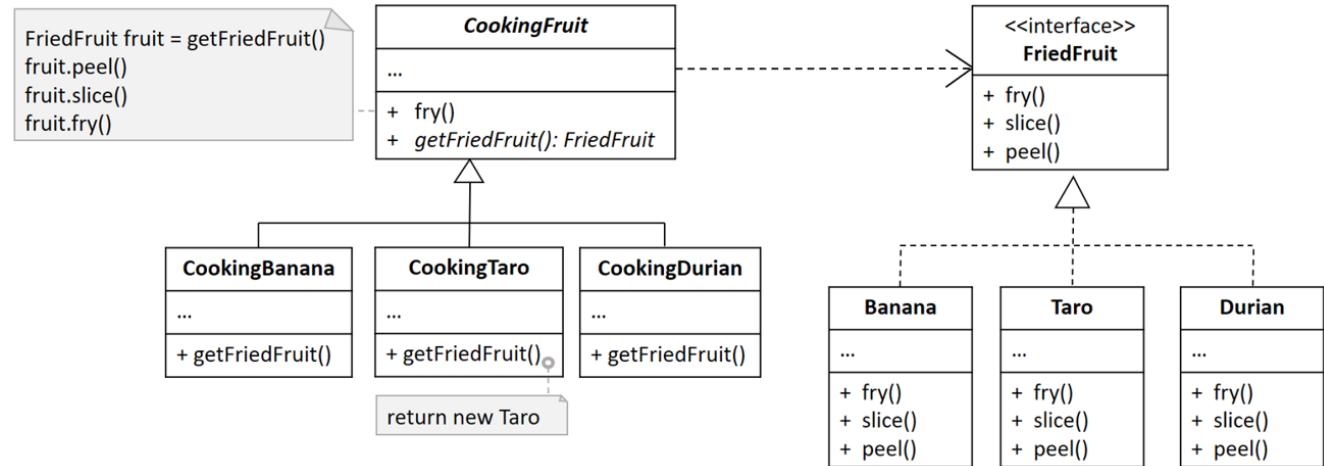
int main() {
    CookingFruit* cook = new CookingBanana;
    client(*cook);
    delete cook;

    cook = new CookingTaro;
    client(*cook);
    delete cook;

    cook = new CookingDurian;
    client(*cook);
    delete cook;

    return 0;
}

```

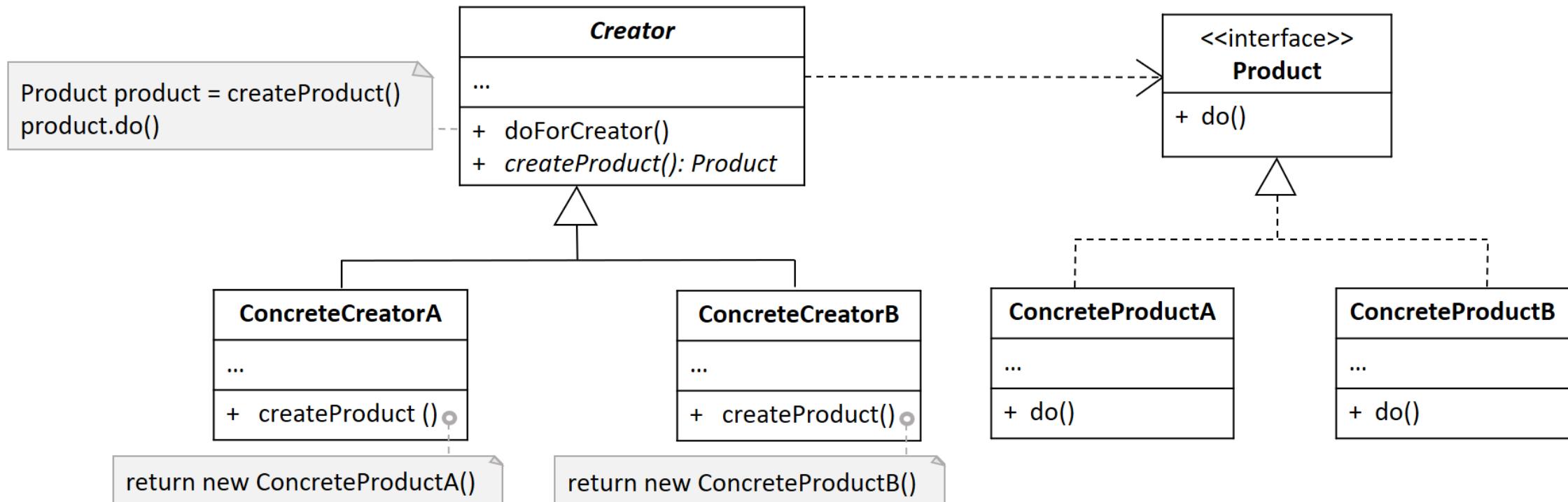


Cooking Fruit...
 peeling bananas
 slicing bananas
 frying bananas

Cooking Fruit...
 peeling taro
 slicing taro
 frying taro

Cooking Fruit...
 peeling durian
 slicing durian
 frying durian

โครงสร้าง



SafariPark
...
+ feedingShow()

Aquarium
...
+ feedingShow()

```
animals = getAnimal()
for each (Animal a in animals) {
    a.eat()
}
```

AnimalPark

...
+ feedingShow()
+ getAnimal()

SafariPark

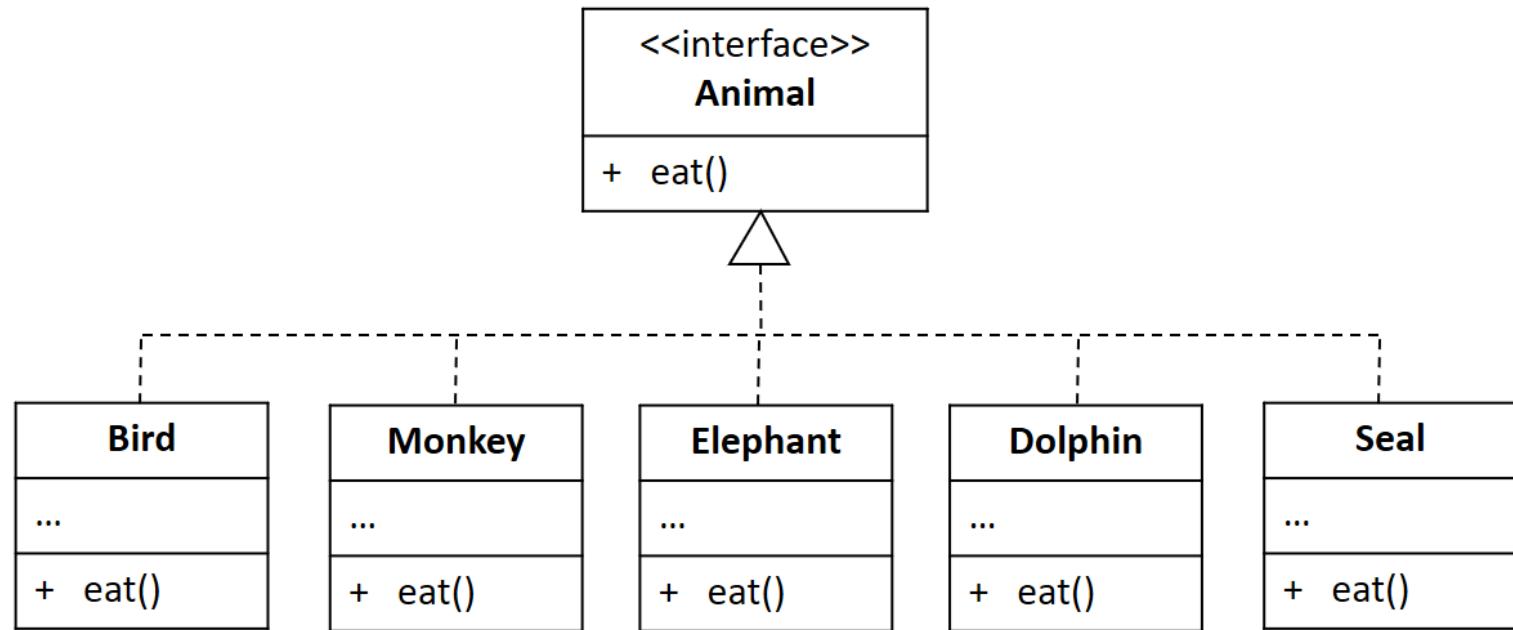
...
+ getAnimal() ●

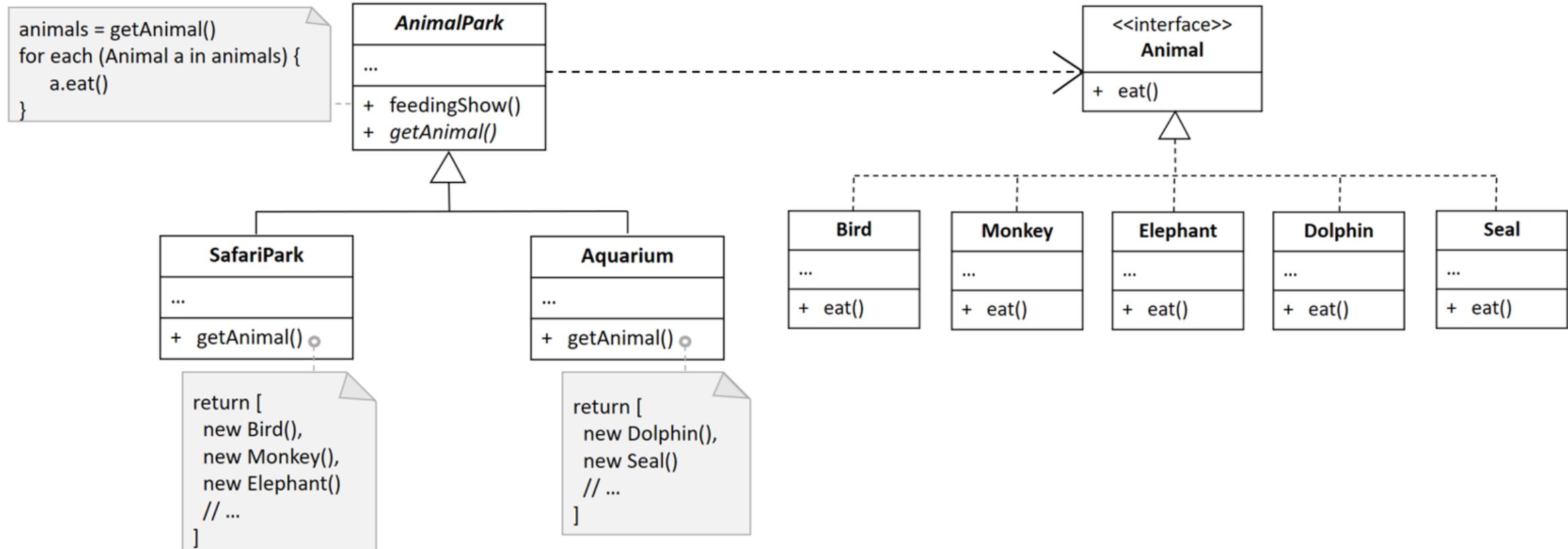
```
return [
    new Bird(),
    new Monkey(),
    new Elephant()
    // ...
]
```

Aquarium

...
+ getAnimal() ●

```
return [
    new Dolphin(),
    new Seal()
    // ...
]
```





```
#include <iostream>
using namespace std;

class Animal {
public:
    virtual ~Animal() {}
    virtual string eat() = 0;
};

class Bird: public Animal {
public:
    string eat() { return "Feeding birds"; }
};

class Dolphin: public Animal {
public:
    string eat() { return "Feeding dolphins"; }
};

class Monkey: public Animal {
public:
    string eat() { return "Feeding monkeys"; }
};

class Seal: public Animal {
public:
    string eat() { return "Feeding seals"; }
};
```

```
class Elephant: public Animal {
public:
    string eat() { return "Feeding elephants"; }
};

class AnimalPark {
int num;
public:
    virtual ~AnimalPark() {};
    virtual Animal** getAnimal() = 0;
    void setNum(int n) { num = n; }
    int getNum() { return num; }

    string feedingShow() {
        Animal** animal = this->getAnimal();

        string result = "";
        for (int i=0; i<getNum(); i++)
            result += animal[i]->eat() + "\n";

        for (int i=0; i<getNum(); i++)
            delete animal[i];
        delete animal;

        return result;
    }
};
```

```
class SafariPark: public AnimalPark {
public:
    Animal** getAnimal() {
        setNum(3);
        Animal** a = new Animal*[getNum()];
        a[0] = new Bird;
        a[1] = new Monkey;
        a[2] = new Elephant;
        return a;
    }
};

class Aquarium: public AnimalPark {
public:
    Animal** getAnimal() {
        setNum(2);
        Animal** a = new Animal*[getNum()];
        a[0] = new Dolphin;
        a[1] = new Seal;
        return a;
    }
};

void client(AnimalPark& park) {
    cout<<park.feedingShow()<<endl;
}
```

```
int main() {
    AnimalPark* park = new SafariPark;
    cout<<"Safari Park"<<endl;
    client(*park);
    delete park;

    cout<<"Aquarium"<<endl;
    park = new Aquarium;
    client(*park);
    delete park;

    return 0;
}
```

```
Safari Park
Feeding birds
Feeding monkeys
Feeding elephants

Aquarium
Feeding dolphins
Feeding seals
```

```
#include <iostream>
#include <list>
using namespace std;

class Animal {
public:
    virtual ~Animal() {}
    virtual string eat() = 0;
};

class Bird: public Animal {
public:
    string eat() { return "Feeding birds"; }
};

class Dolphin: public Animal {
public:
    string eat() { return "Feeding dolphins"; }
};

class Monkey: public Animal {
public:
    string eat() { return "Feeding monkeys"; }
};

class Seal: public Animal {
public:
    string eat() { return "Feeding seals"; }
};

class Elephant: public Animal {
public:
    string eat() { return "Feeding elephants"; }
};

class AnimalPark {
public:
    virtual ~AnimalPark() {};
    virtual list <Animal*> getAnimal() = 0;

    string feedingShow() {
        list <Animal*> animal = this->getAnimal();
        string result = "";
        for (Animal *a : animal) {
            result += a->eat() + "\n";
        }
        for (Animal *a : animal) {
            delete a;
        }
        return result;
    }
};
```

```
class SafariPark: public AnimalPark {
public:
    list <Animal*> getAnimal() {
        list <Animal*> a;
        a.push_back(new Bird);
        a.push_back(new Monkey);
        a.push_back(new Elephant);
        return a;
    }
};

class Aquarium: public AnimalPark {
public:
    list <Animal*> getAnimal() {
        list <Animal*> a;
        a.push_back(new Dolphin);
        a.push_back(new Seal);
        return a;
    }
};

void client(AnimalPark& park) {
    cout<<park.feedingShow()<<endl;
}
```

```
int main() {
    AnimalPark* park = new SafariPark;
    cout<<"Safari Park"<<endl;
    client(*park);
    delete park;

    cout<<"Aquarium"<<endl;
    park = new Aquarium;
    client(*park);
    delete park;

    return 0;
}
```

Safari Park
Feeding birds
Feeding monkeys
Feeding elephants

Aquarium
Feeding dolphins
Feeding seals

ควรใช้เมื่อไหร่

- กรณีที่เราไม่ทราบชนิดของ **product** ที่จะต้องทำงานด้วยล่วงหน้า
 - สามารถ **extend product** ได้ง่าย เป็นอิสระจากโค้ด
- เมื่อต้องการให้ **user** ที่ใช้ไลบรารีหรือเฟล์มเวิร์คของเราสามารถเพิ่มขยายส่วนประกอบของตนเองได้
 - เพื่อที่เฟล์มเวิร์คจะได้เรียกใช้เมธอดของคลาสลูกแทนการใช้เมธอดปกติ
- กรณีที่ต้องการประยัดทรัพยากรของระบบ โดยการใช้ออบเจกต์ที่มีอยู่แล้วแทนการสร้างใหม่ทุกครั้ง
 - แทนที่จะสร้างออบเจกต์ใหม่โดยใช้ค่อนสตรัคเตอร์ ก็ใช้ **factory method** ในกรณีที่ต้องการให้

ข้อดี ข้อเสีย

- ข้อดี
 - ลด coupling ระหว่าง creator กับ concrete product
 - Single Responsibility Principle
 - สามารถนำได้ดั่งที่ใช้สร้าง Product ไปได้ที่เดียวกันในโปรแกรม
 - Open/Closed Principle
 - สามารถสร้าง product ใหม่ๆ ได้โดยไม่กระทบกับโค้ดเดิม
- ข้อเสีย
 - โค้ดมีความซับซ้อน

การบ้าน

- ระบบขนส่ง ทางบก และ ทางน้ำ
 - ทางบกโดยรถบรรทุก
 - ทางน้ำโดยเรือ
- พิซซ่า
 - bacon pizza, tuna pizza, grilled pork neck pizza
 - ขั้นตอนการทำพิซซ่าเหมือนกัน prepare, bake, cut, box
 - หน้าของพิซซ่าต่างกัน bacon, tuna, grilled pork neck

อ้างอิง

- <https://refactoring.guru/design-patterns/factory-method>

Catalog of Design Patterns

Creational Design Patterns

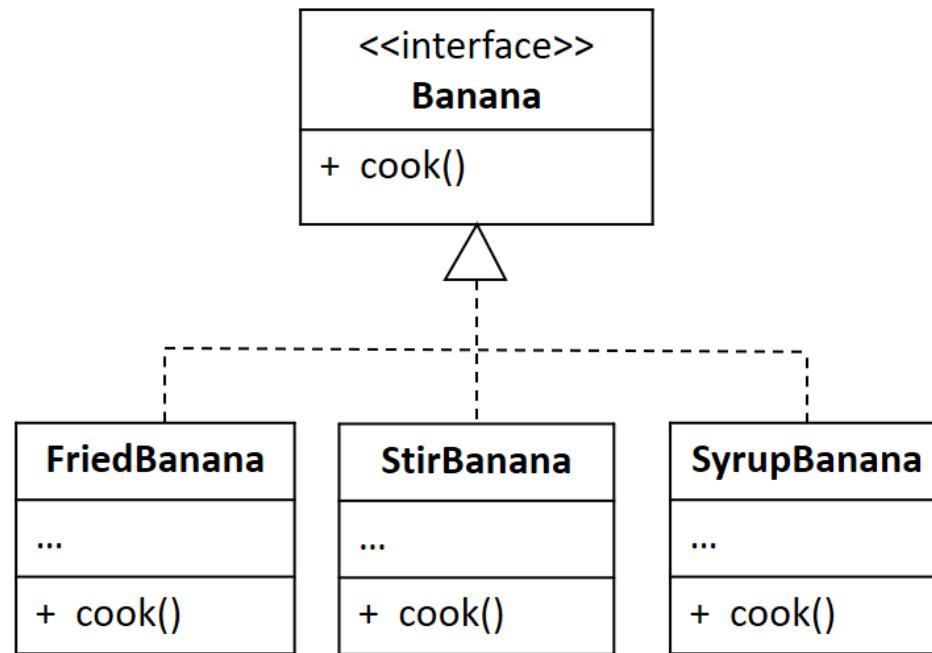
Abstract Factory

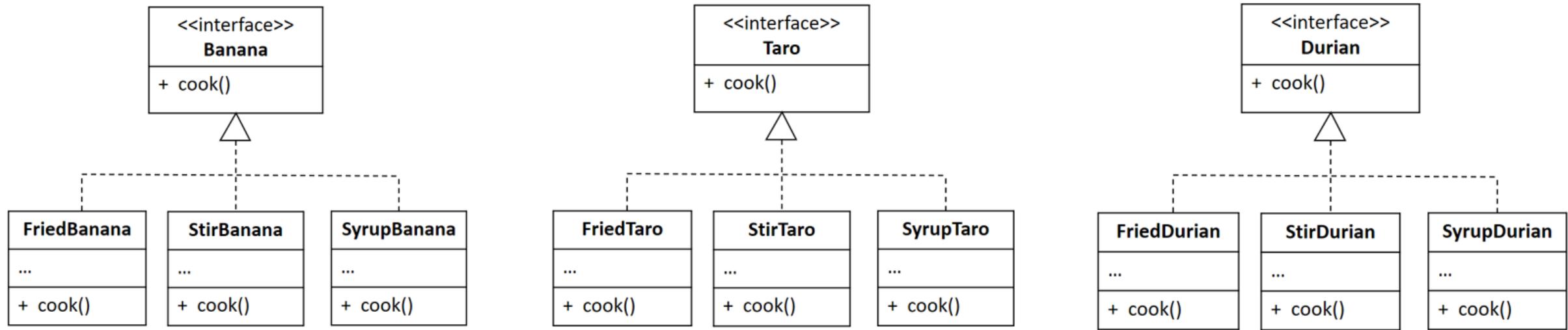
Abstract Factory

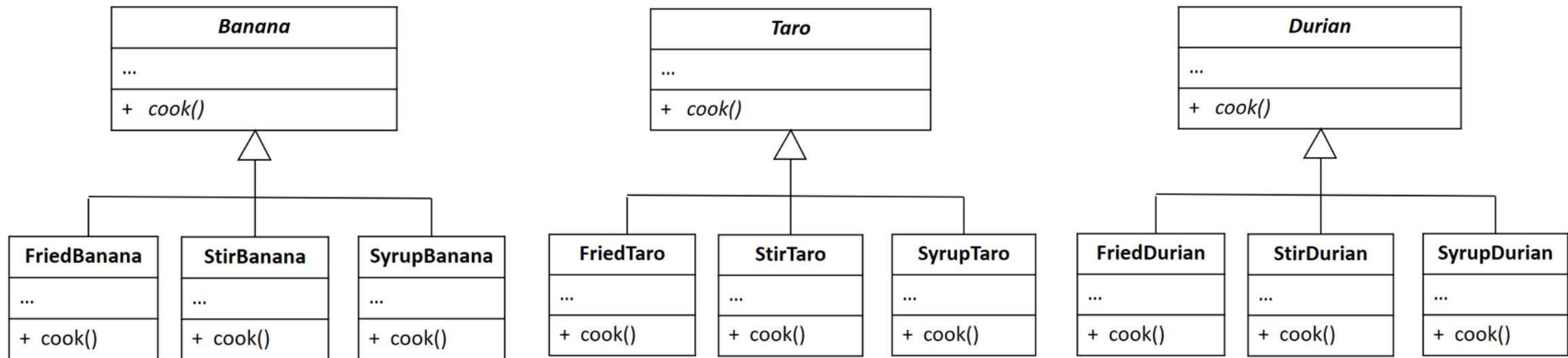
- สร้างอ็อบเจกต์ที่มีความเกี่ยวข้องกันให้อยู่ในกลุ่มเดียวกัน โดยไม่ต้องระบุ concrete class
- ตัวอย่าง

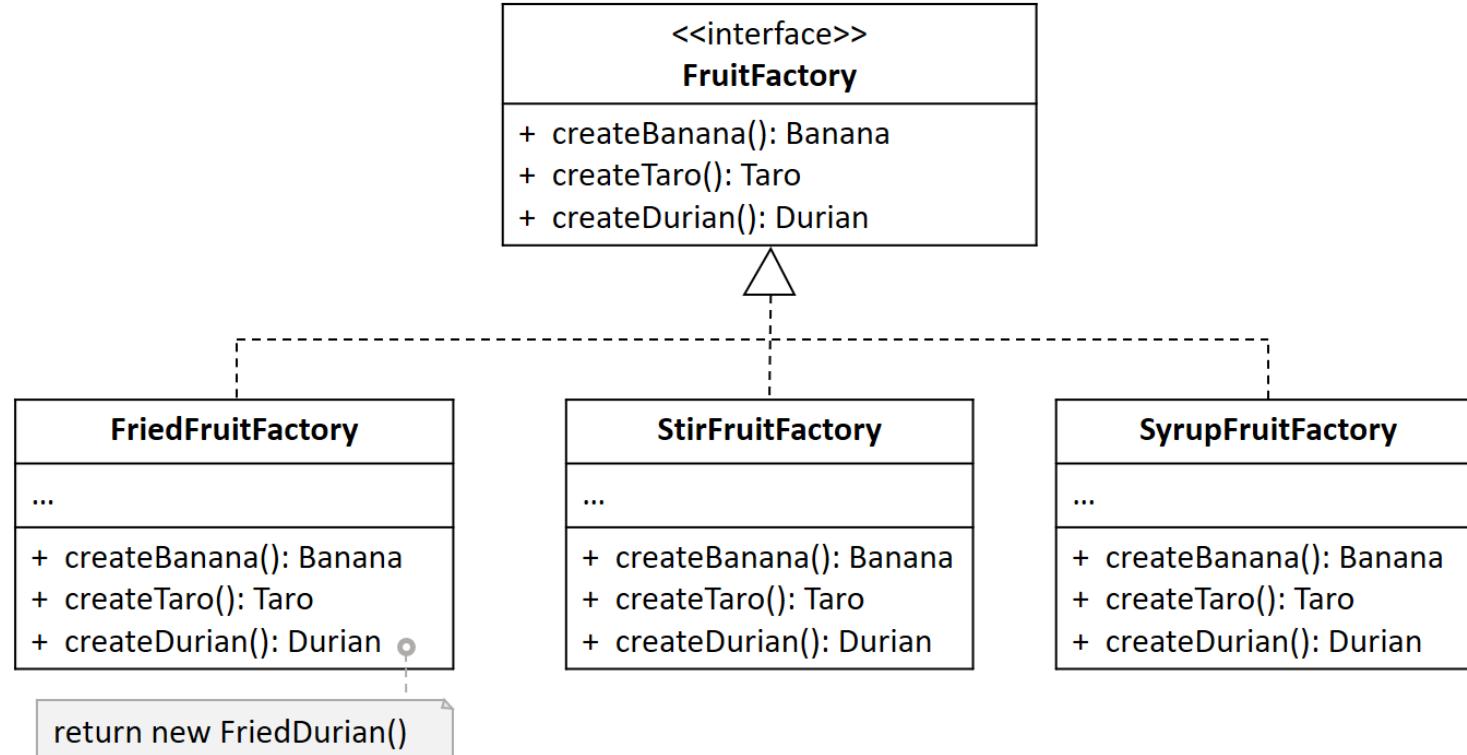
	Banana	Taro	Durian
Fry	กล้วยทอด	เผือกทอด	ทุเรียนทอด
Stir	กล้วยหวาน	เผือกหวาน	ทุเรียนหวาน
In Syrup	กล้วยเชื่อม	เผือกเชื่อม	ทุเรียนเชื่อม

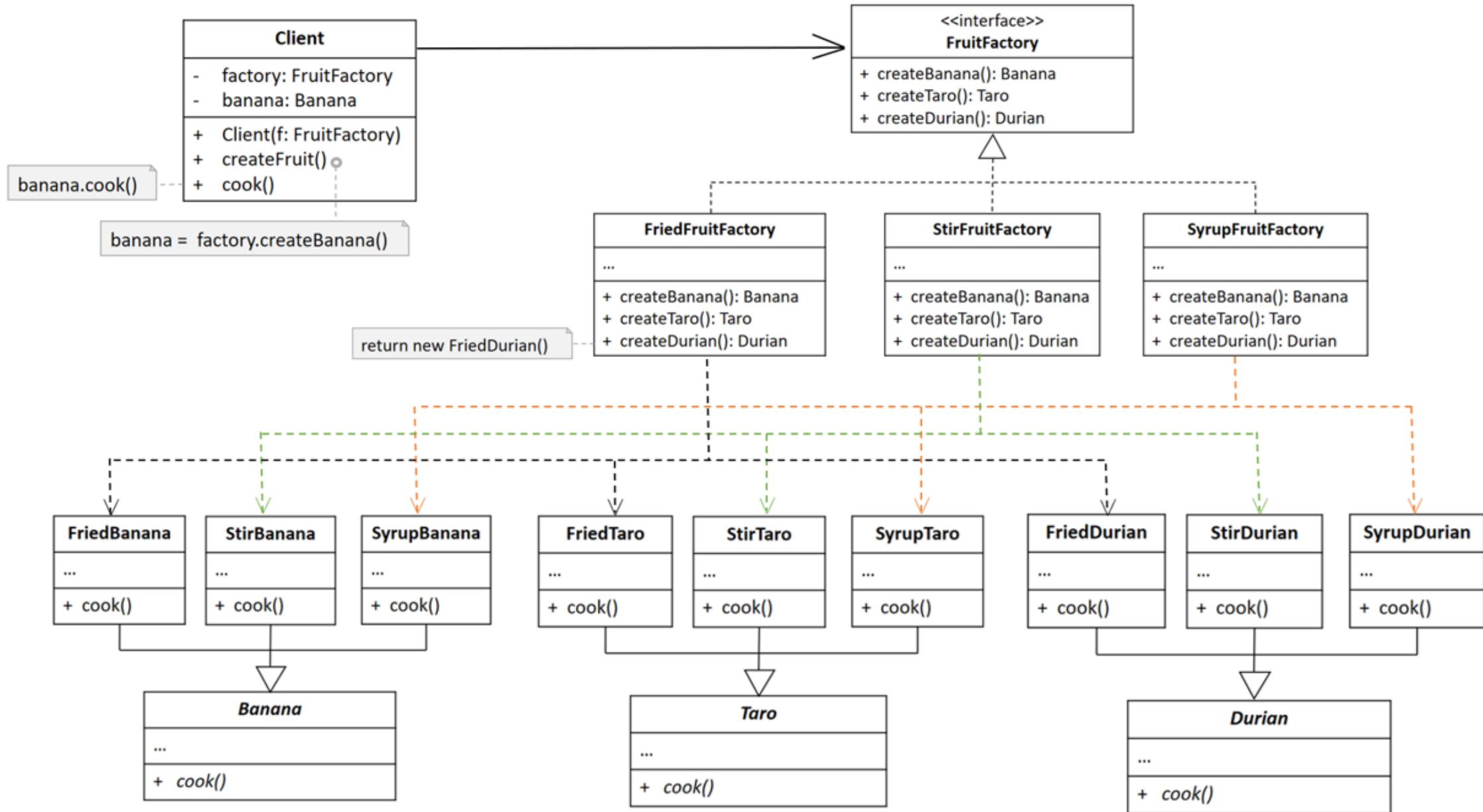
- เช่น สร้างกลุ่มผลิตภัณฑ์ที่เกี่ยวกับการทำด้วยไม่ต้องระบุคลาสตัวจริงที่สร้างอ็อบเจกต์











```
#include <iostream>
using namespace std;

class Durian {
public:
    virtual ~Durian() {};
    virtual string cook() = 0;
};

class FriedDurian: public Durian {
public:
    string cook() {
        return "fried durian";
    }
};

class StirDurian: public Durian {
public:
    string cook() {
        return "stir durian";
    }
};

class SyrupDurian: public Durian {
public:
    string cook() {
        return "syrup durian";
    }
};

class Taro {
public:
    virtual ~Taro() {};
    virtual string cook() = 0;
};

class FriedTaro: public Taro {
public:
    string cook() {
        return "fried taro";
    }
};

class StirTaro: public Taro {
public:
    string cook() {
        return "stir taro";
    }
};

class SyrupTaro: public Taro {
public:
    string cook() {
        return "syrup taro";
    }
};
```

```
class Banana {
public:
    virtual ~Banana() {};
    virtual string cook() = 0;
    virtual string mix(Taro &t) = 0;
};

class FriedBanana: public Banana {
public:
    string cook() {
        return "fried banana";
    }
    string mix(Taro &t) {
        string result = t.cook();
        return "fried banana + " + result;
    }
};

class StirBanana: public Banana {
public:
    string cook() {
        return "stir banana";
    }
    string mix(Taro &t) {
        string result = t.cook();
        return "stir banana + " + result;
    }
};

class SyrupBanana: public Banana {
public:
    string cook() {
        return "syrup banana";
    }
    string mix(Taro &t) {
        string result = t.cook();
        return "syrup banana + " + result;
    }
};

class FruitFactory {
public:
    virtual Durian *createDurian() = 0;
    virtual Taro *createTaro() = 0;
    virtual Banana *createBanana() = 0;
};

class FriedFruitFactory: public FruitFactory {
public:
    Durian *createDurian() {
        return new FriedDurian();
    }
    Taro *createTaro() {
        return new FriedTaro();
    }
    Banana *createBanana() {
        return new FriedBanana();
    }
};
```

```
class StirFruitFactory: public FruitFactory {
public:
    Durian *createDurian() {
        return new StirDurian();
    }
    Taro *createTaro() {
        return new StirTaro();
    }
    Banana *createBanana() {
        return new StirBanana();
    }
};

class SyrupFruitFactory: public FruitFactory {
public:
    Durian *createDurian() {
        return new SyrupDurian();
    }
    Taro *createTaro() {
        return new SyrupTaro();
    }
    Banana *createBanana() {
        return new SyrupBanana();
    }
};
```

```
void client(FruitFactory &factory) {
    Durian *durian = factory.createDurian();
    Taro *taro = factory.createTaro();
    Banana *banana = factory.createBanana();
    cout<<durian->cook()<<endl;
    cout<<taro->cook()<<endl;
    cout<<banana->cook()<<endl;
    cout<<banana->mix(*taro)<<endl;

    delete durian;
    delete taro;
    delete banana;
}
```

```

class BananaApp {
    FruitFactory *factory;
    Banana *banana;
public:
    BananaApp() {
        factory=0;
        banana=0;
    }
    BananaApp(FruitFactory *f) {
        factory = f;
        banana=0;
    }
    ~BananaApp() {
        delete banana;
    }
    void createFruit() {
        if (banana!=0)
            delete banana;
        banana = factory->createBanana();
    }
    string cook() {
        return banana->cook();
    }
};

int main() {
    cout<<"Fried Factory"<<endl;
    cout<<"-----"<<endl;
    FriedFruitFactory *fry = new FriedFruitFactory;
    client(*fry);
    cout<<endl<<endl;
}

cout<<"Stir Factory"<<endl;
cout<<"-----"<<endl;
StirFruitFactory *stir = new StirFruitFactory;
client(*stir);
cout<<endl<<endl;

cout<<"Syrup Factory"<<endl;
cout<<"-----"<<endl;
SyrupFruitFactory *syrup = new SyrupFruitFactory;
client(*syrup);
cout<<endl<<endl;

cout<<"using Banana App"<<endl;
cout<<"-----"<<endl;
BananaApp ba1(fry);
ba1.createFruit();
cout<<ba1.cook()<<endl;

BananaApp ba2(syrup);
ba2.createFruit();
cout<<ba2.cook()<<endl;

ba2.createFruit();
cout<<ba2.cook()<<endl;

delete fry;
delete stir;
delete syrup;
return 0;
}

```

Fried Factory

fried durian
fried taro
fried banana
fried banana + fried taro

Stir Factory

stir durian
stir taro
stir banana
stir banana + stir taro

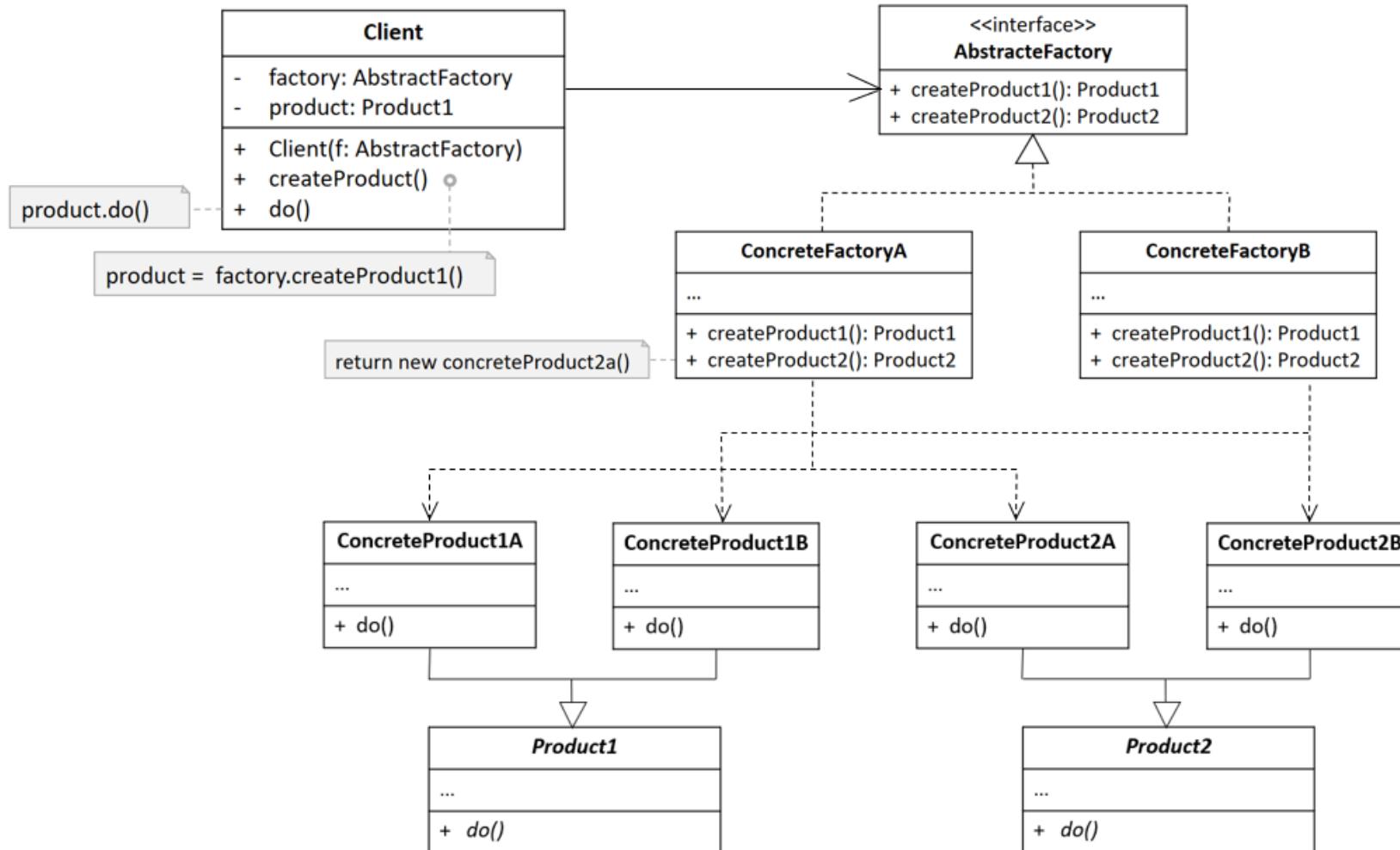
Syrup Factory

syrup durian
syrup taro
syrup banana
syrup banana + syrup taro

using Banana App

fried banana
syrup banana
syrup banana

โครงสร้าง



ควรใช้เมื่อไหร่

- เมื่อต้องการทำงานกับกลุ่มของ **product** หลายตัว และไม่ต้องการให้ขึ้นอยู่กับ **concrete class** ของ **product**

ข้อดี ข้อเสีย

- ข้อดี
 - มั่นใจได้ว่า product ต่างๆ ที่ได้จาก factory นั้นเข้ากันได้
 - ลดการขึ้นต่อ กันระหว่าง concrete product กับ client
 - Single responsibility principle
 - สามารถนำโค้ดที่ใช้สร้าง product ไปไว้ในที่เดียวกัน
 - Open/Closed principle
 - สามารถสร้าง product ใหม่โดยไม่กระทบกับโค้ดเดิม
- ข้อเสีย
 - โค้ดมีความซับซ้อน

การบ้าน

	Dining Table	Sofa	TV stand
Modern	โต๊ะอาหารสีตอล์โนเมเดริน	โซฟาสีตอล์โนเมเดริน	ตู้วางทีวีสีตอล์โนเมเดริน
The '80s	โต๊ะอาหารสีตอล์ยุค '80	โซฟาสีตอล์ยุค '80	ตู้วางทีวีสีตอล์ยุค '80
Ancient Thai	โต๊ะอาหารสีตอล์ไทยโบราณ	โซฟาสีตอล์ไทยโบราณ	ตู้วางทีวีสีตอล์ไทยโบราณ

	Bacon pizza	Tuna pizza	Grilled pork neck pizza
แบ่งบางกรอบ	พิซซ่าหน้าเบคอน แบ่งบางกรอบ	พิซซ่าหน้าทูน่า แบ่งบางกรอบ	พิซซ่าหน้าคอดหมูย่าง แบ่งบางกรอบ
แบ่งหนานุ่ม	พิซซ่าหน้าเบคอน แบ่งหนานุ่ม	พิซซ่าหน้าทูน่า แบ่งหนานุ่ม	พิซซ่าหน้าคอดหมูย่าง แบ่งหนานุ่ม

อ้างอิง

- <https://refactoring.guru/design-patterns/abstract-factory>

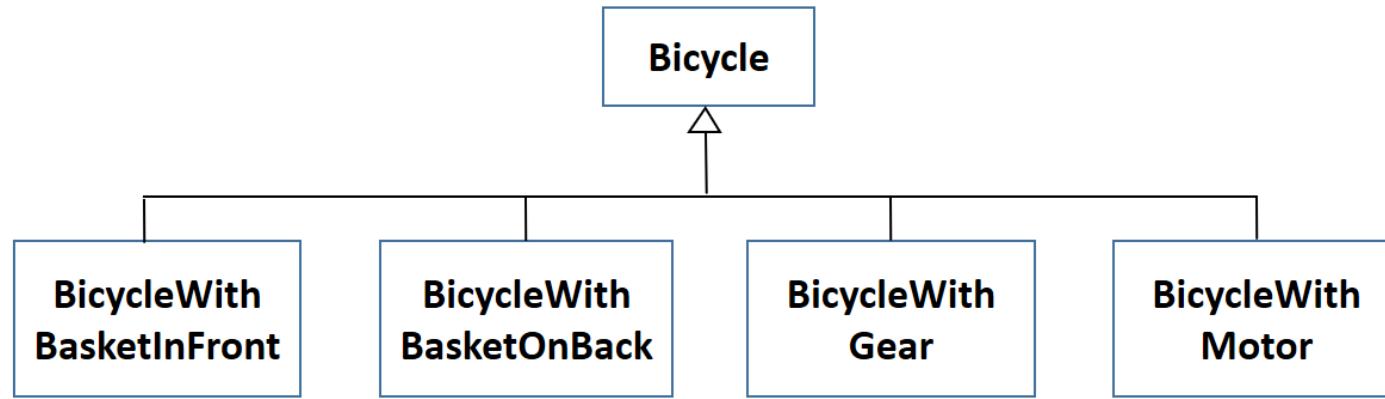
Catalog of Design Patterns

Creational Design Patterns

Builder

Builder

- ใช้สร้างอ็อบเจกต์ที่มีความซับซ้อน โดยค่อยๆ สร้างทีละขั้น สามารถใช้โค้ดชุดเดียวกันในการสร้างอ็อบเจกต์ที่มี type และ representation ที่แตกต่างกันได้



✗

Bicycle
...
+ Bicycle(seat, hasFrontBasket, hasBackBasket, hasGear, hasMotor, ...)

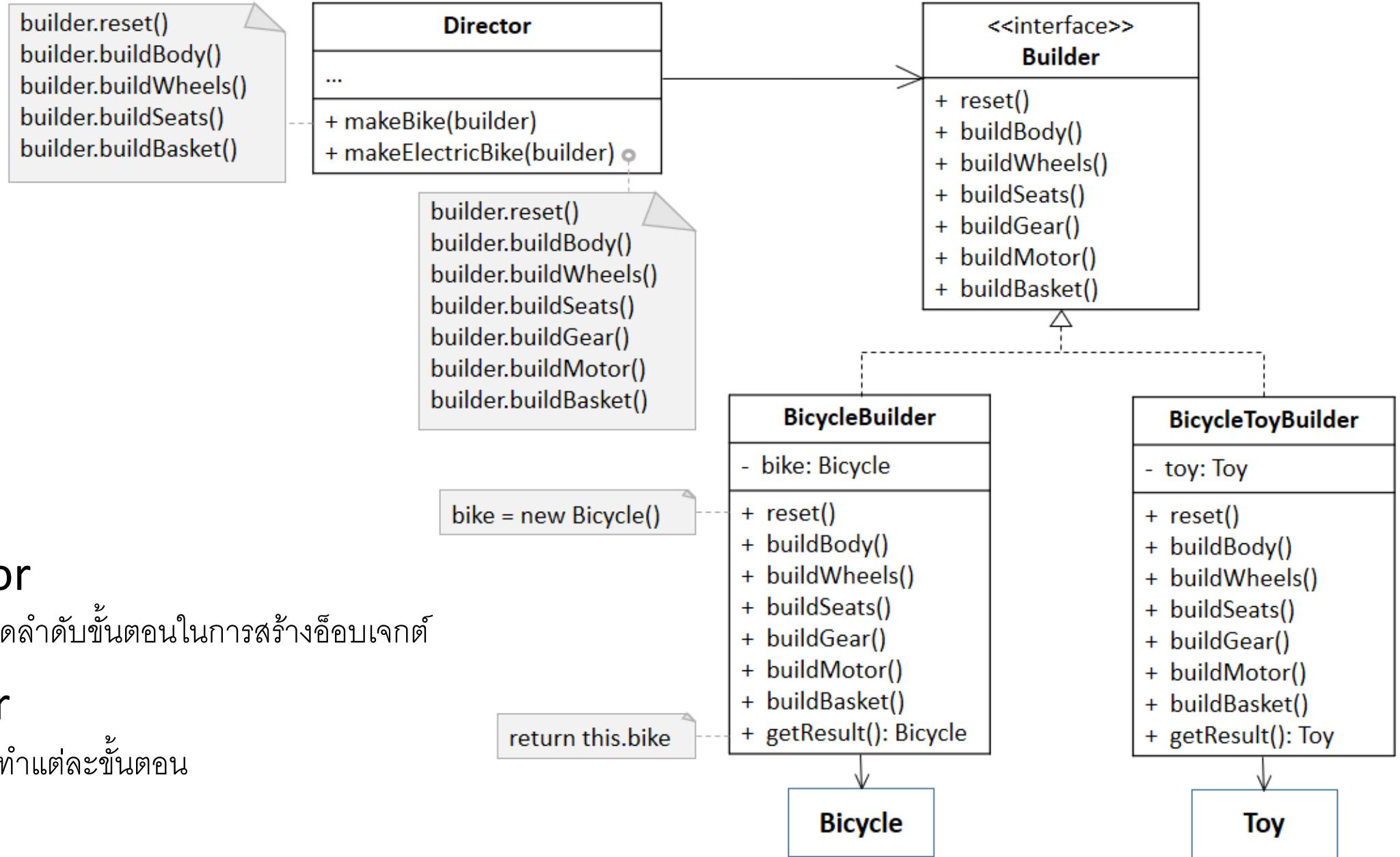
new Bicycle(2, true, null, null, null, ...)

new Bicycle(1, null, null, true, true, ...)



BicycleBuilder
...
+ buildBody() + buildWheels() + buildSeats() + buildGear() + buildMotor() + buildBasket() + getResult(): Bicycle

✓



```
#include <iostream>
using namespace std;

class Bicycle {
    int seat;
    bool gear;
    bool motor;
    bool basket;
public:
    Bicycle() {
        seat=2;
        gear=false;
        motor=false;
        basket=false;
    }
    void setBody() {
        cout<<"bike body frame is built"=<<endl;
    }
    void setWheels() {
        cout<<"wheels are built"=<<endl;
    }
    void setSeat(int i) {
        cout<<"seats are built"=<<endl;
        seat=i;
    }
    void setGear(bool b) {
        cout<<"gear is built"=<<endl;
        gear = b;
    }
    void setMotor(bool b) {
        cout<<"motor is built"=<<endl;
        motor = b;
    }
    void setBasket(bool b) {
        cout<<"basket is built"=<<endl;
        basket = b;
    }
    int getSeat() { return seat; }
    bool getGear() {
        return gear;
    }
    bool getMotor() {
        return motor;
    }
    bool getBasket() {
        return basket;
    }
    void show() {
        cout<<"Create:: Bicycle with "<<seat<<(seat>1? " seats":" seat");
        if (gear) cout<<, gear";
        if (motor) cout<<, motor";
        if (basket) cout<<" and basket"=<<endl;
    }
};
```

```
class Builder{
public:
    virtual ~Builder(){}
    virtual void reset() = 0;
    virtual void buildBody() = 0;
    virtual void buildWheels() = 0;
    virtual void buildSeats() = 0;
    virtual void buildGear() = 0;
    virtual void buildMotor() = 0;
    virtual void buildBasket() = 0;
};

class BicycleBuilder: public Builder {
    Bicycle *bike;
    void reset(){
        bike = new Bicycle();
    }
public:
    BicycleBuilder(){
        this->reset();
    }
    ~BicycleBuilder(){
        delete bike;
    }
    void buildBody() {
        bike->setBody();
    }
}
```

```
void buildWheels() {
    bike->setWheels();
}
void buildSeats() {
    bike->setSeat(2);
}
void buildGear() {
    bike->setGear(true);
}
void buildMotor() {
    bike->setMotor(true);
}
void buildBasket() {
    bike->setBasket(true);
}
Bicycle* getProduct() {
    Bicycle* result= this->bike;
    this->reset();
    return result;
}
};
```

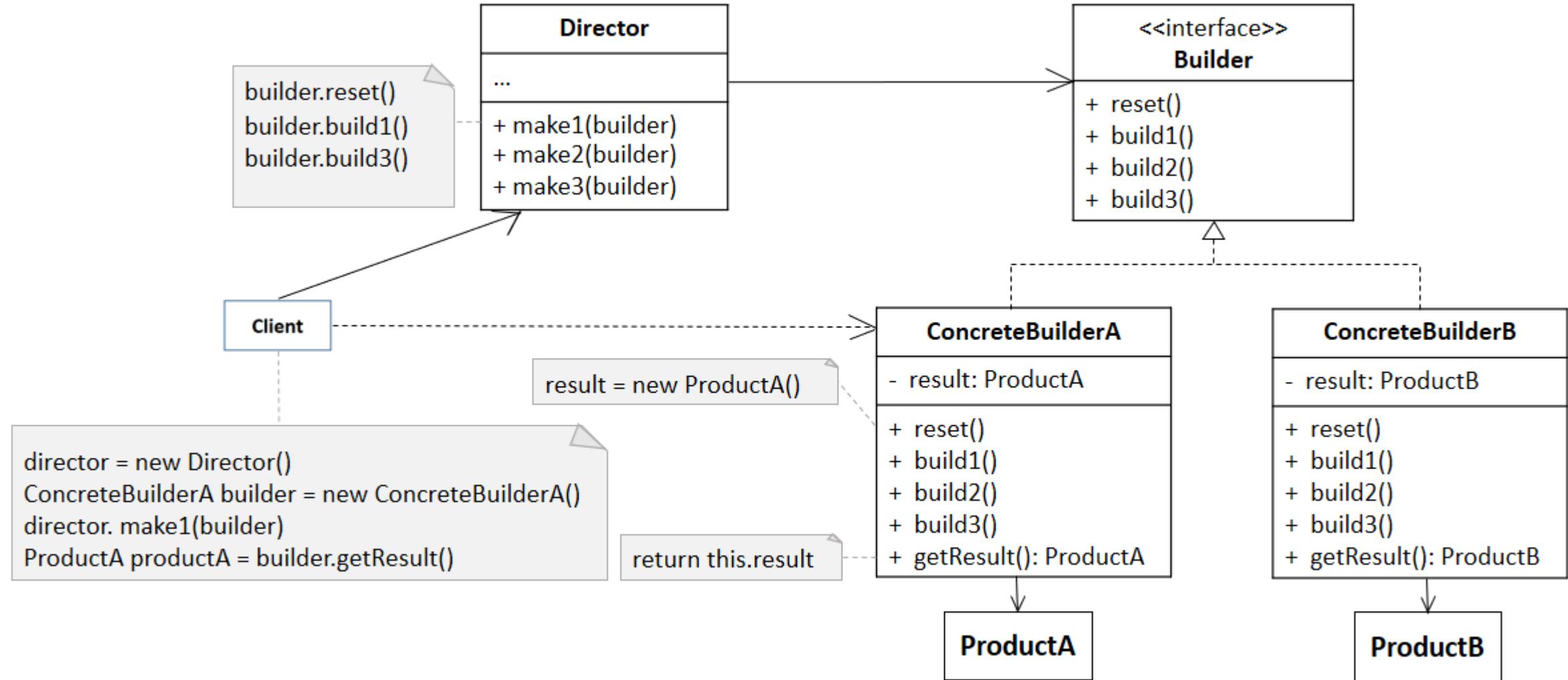
```
class Director{  
  
public:  
void makeBike(Builder *builder) {  
    builder->buildBody();  
    builder->buildWheels();  
    builder->buildSeats();  
    builder->buildBasket();  
}  
void makeElectricBike(Builder *builder) {  
    builder->buildBody();  
    builder->buildWheels();  
    builder->buildSeats();  
    builder->buildGear();  
    builder->buildMotor();  
    builder->buildBasket();  
}  
};
```

```
.....  
Create bicycle  
.....  
bike body frame is built  
wheels are built  
seats are built  
basket is built  
Create:: Bicycle with 2 seats and basket
```

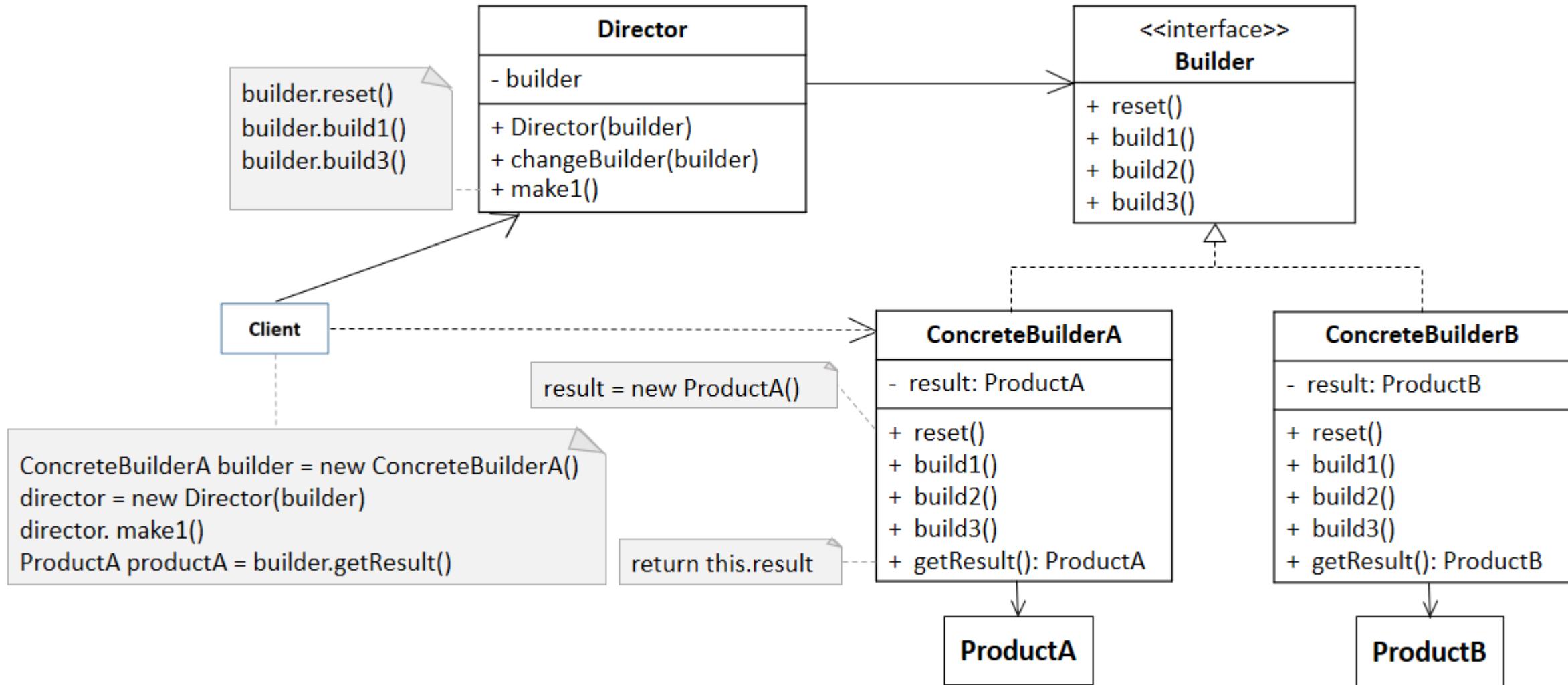
```
.....  
Create electric bicycle  
.....  
bike body frame is built  
wheels are built  
seats are built  
gear is built  
motor is built  
basket is built  
Create:: Bicycle with 2 seats, gear, motor and basket
```

```
void ClientCode(Director& director)  
{  
    BicycleBuilder *bikeBuilder = new BicycleBuilder();  
    cout<<"....." << endl;  
    cout<<"Create bicycle" << endl;  
    cout<<"....." << endl;  
    director.makeBike(bikeBuilder);  
    Bicycle *bike = bikeBuilder->getProduct();  
    bike->show();  
    delete bike;  
    cout<< endl;  
  
    cout<<"....." << endl;  
    cout<<"Create electric bicycle" << endl;  
    cout<<"....." << endl;  
    director.makeElectricBike(bikeBuilder);  
    bike = bikeBuilder->getProduct();  
    bike->show();  
    delete bike;  
  
    delete bikeBuilder;  
}  
  
int main()  
{  
    Director* director= new Director();  
    ClientCode(*director);  
    delete director;  
    return 0;  
}
```

Structure



Structure



ควรใช้เมื่อไหร่

- เมื่อไม่ต้องการให้มี telescopic constructor

- เช่น

```
class Bicycle {  
    Bicycle(int seat) {...}  
    Bicycle(int seat, bool hasBasket) {...}  
    Bicycle(int seat, bool hasBasket, bool hasGear) {...}  
    ...  
}
```

- เมื่อต้องการให้โค้ดสามารถสร้าง product ที่มี representation ต่างกันได้ เช่น รถจักรยาน กับ รถจักรยานของเล่น
 - ขั้นตอนการสร้างเหมือนกันแต่แตกต่างกันที่รายละเอียด

ควรใช้เมื่อไหร่

- เมื่อต้องการสร้างอ็อบเจกต์ที่มีความซับซ้อน
 - แต่ละขั้นตอน สามารถที่จะถ่วงเวลาในการ execute ได้ หรือสามารถทำงานขั้นตอนซ้ำๆ ได้ (recursive)
 - ในขณะที่กำลังสร้าง product นั้น builder pattern จะไม่ปล่อย product ที่ยังสร้างไม่สำเร็จออกมา

ข้อดี ข้อเสีย

- ข้อดี
 - สามารถสร้างอ็อบเจกต์ step-by-step, สามารถถ่วงเวลาการสร้างแต่ละ step ได้, สามารถทำ step ซ้ำๆ ได้ (recursive)
 - สามารถ reuse โค้ดที่ใช้สร้าง product ได้ ในการนี้ที่ต้องการสร้าง product ที่มี representation ต่างกัน
 - Single Responsibility Principle
 - แยกโค้ดที่ใช้ในการสร้าง product ออกจาก business logic
- ข้อเสีย
 - โค้ดมีความซับซ้อน

การบ้าน

- สร้างบ้าน 3 แบบ บ้านทรงไทย บ้านทรงโมเดิร์น บ้านทรงอังกฤษ
 - แต่ละแบบใช้วัสดุและรูป่างของส่วนประกอบบ้านต่างกัน เช่น ประตู หน้าต่าง ฝ้า หลังคา
 - แต่ละแบบมีขั้นตอนการสร้างเหมือนกัน เช่น สร้างฝ้า สร้างประตู สร้างหน้าต่าง สร้างหลังคา สร้างโรงรถ (**builder**)
 - บ้านอาจมีองค์ประกอบไม่เหมือนกัน เช่น บางบ้านมีลิฟต์ภายใน หรือมีบันไดเลื่อน (**director**)

อ้างอิง

- <https://refactoring.guru/design-patterns/builder>

Catalog of Design Patterns

Creational Design Patterns

Prototype

Prototype หรือ Clone

- ใช้ก็อปปี้ออบเจกต์ที่มีอยู่แล้ว โดยที่โค้ดของเราไม่ต้องเขียนอยู่กับคลาสของอ้อบเจกต์ที่เรา ก็อปปี้มา

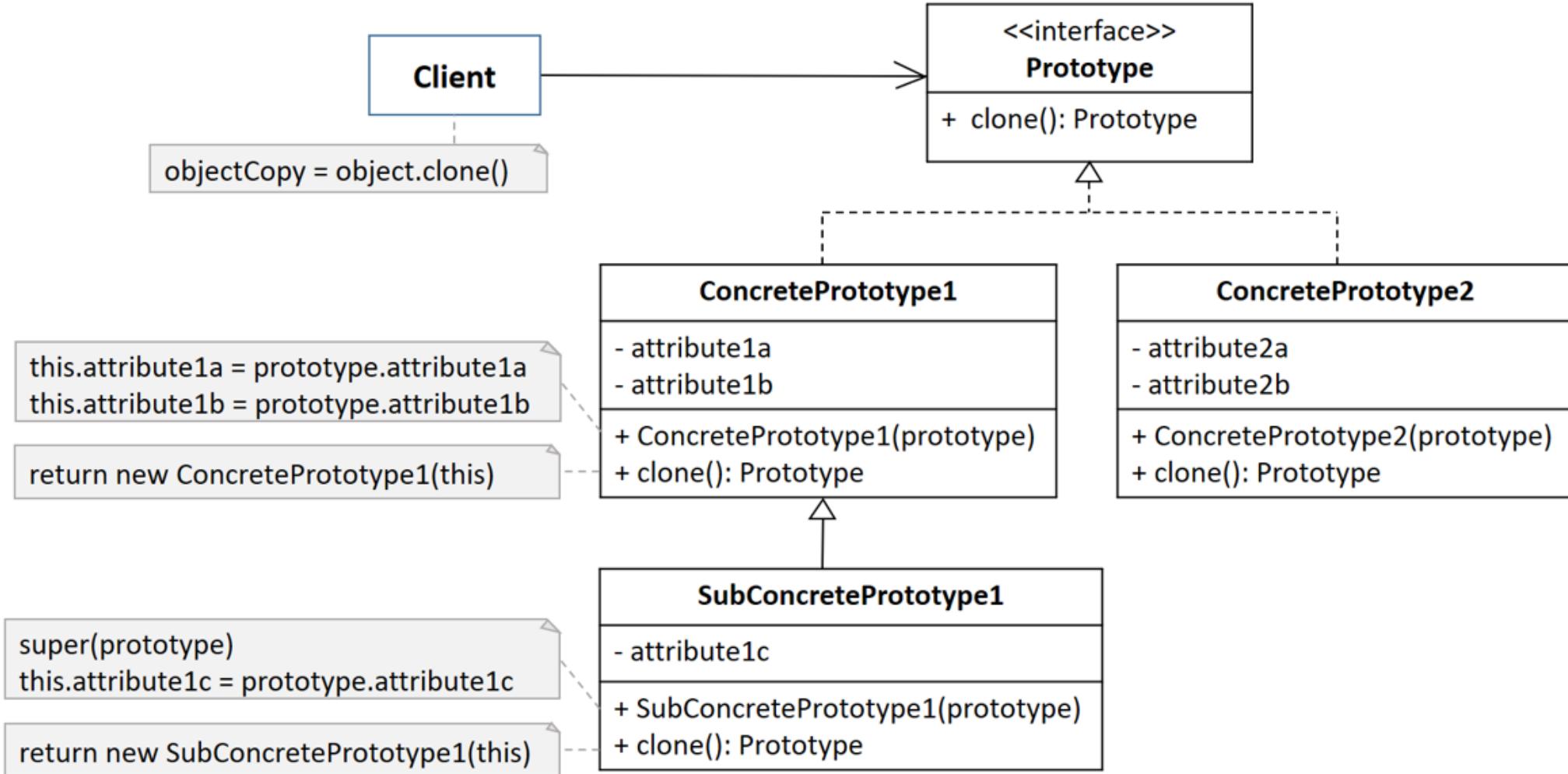
ปัญหา

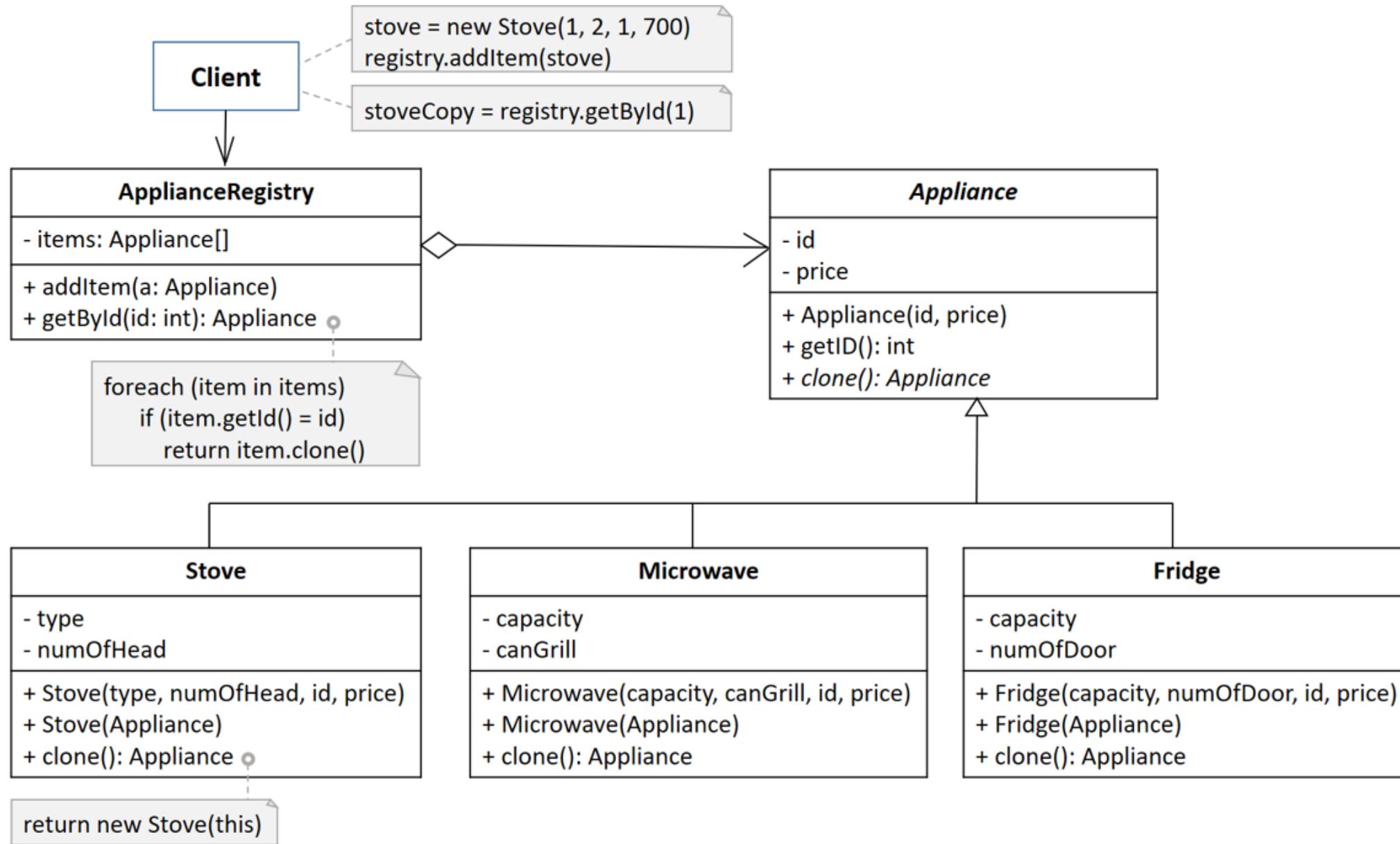
- กรณีก็อปปี้มาตรงๆ
 - บางครั้งก็ไม่สามารถก็อปปี้จากภาษาณอกได้ เพราะว่าแอทริบิวเป็น **private**
 - ต้องรู้จักคลาสของอ้อบเจกต์ที่เราจะก็อปปี้มา ดังนั้นโค้ดเราจะเขียนอยู่กับคลาสนั้น
 - บางทีเรารู้จักแค่ **interface** ของอ้อบเจกต์นั้น แต่ไม่รู้ว่า **concrete class** คือคลาสไหน

วิธีแก้ปัญหา

- ให้ออบเจกต์ที่เราต้องการจะก็อปปี้มา ทำการก็อปปี้ให้
- สร้าง interface สำหรับอ้อบเจกต์ทุกตัวที่จะทำการก็อปปี้
- เราจะใช้ interface นี้ในการก็อปปี้อ้อบเจกต์โดยไม่ต้องไปยุ่งกับคลาสของอ้อบเจกต์
- ใน interface จะมีเมธอดที่ชื่อว่า clone
- อ้อบเจกต์ที่ทำการ clone ได้ เราเรียกว่า prototype

Structure





```
#include <iostream>
using namespace std;

class Appliance {
protected:
    int id;
    double price;
public:
    Appliance() { id=0; price=0; }
    Appliance(int id, double price) {
        this->id = id;
        this->price = price;
    }
    virtual ~Appliance() {}
    virtual Appliance *clone() = 0;
    virtual void show() {
        cout<<"ID: "<<id<<endl;
        cout<<"Price: "<<price<<endl;
    }
    int getID() {
        return id;
    }
};
```

```
class Stove: public Appliance {
    int type;
    int numOfHead;
public:
    Stove() { type=0; numOfHead=0; }
    Stove(int type, int num, int id, double price):Appliance(id, price) {
        this->type = type;
        this->numOfHead = num;
    }
    Appliance *clone() {
        return new Stove(*this);
    }
    void show() {
        Appliance::show();
        cout<<"Type: "<<(type?"gas":"electric")<<endl;
        cout<<"Head: "<<numOfHead<<endl;
    }
};
```

```
class Microwave: public Appliance {
    int capacity;
    bool canGrill;
public:
    Microwave() { capacity=0; canGrill=false; }
    Microwave(int cap, bool grill, int id, double price):Appliance(id, price) {
        this->capacity = cap;
        this->canGrill = grill;
    }
    Appliance *clone() {
        return new Microwave(*this);
    }
    void show() {
        Appliance::show();
        cout<<"Capacity: "<<capacity<<endl;
        cout<<"Grill: "<<(canGrill?"yes":"no")<<endl;
    }
};
```

```
class Fridge: public Appliance {
    int capacity;
    int numofDoor;
public:
    Fridge() { capacity=0; numofDoor=0; }
    Fridge(int cap, int num, int id, double price):Appliance(id, price) {
        this->capacity = cap;
        this->numofDoor = num;
    }
    Appliance *clone() {
        return new Fridge(*this);
    }
    void show() {
        Appliance::show();
        cout<<"Capacity: "<<capacity<<endl;
        cout<<"Door: "<<numofDoor<<endl;
    }
};
```

```

class ApplianceRegistry {
    Appliance *items[3];
public:
    ApplianceRegistry() {
        items[0] = new Stove(1, 2, 1, 700);
        items[1] = new Microwave(30, true, 2, 5000);
        items[2] = new Fridge(300, 4, 3, 16000);
    }
    ~ApplianceRegistry() {
        for (int i=0; i<3; i++) {
            delete items[i];
        }
    }
    Appliance *createAppliance(int id) {
        for (int i=0; i<3; i++) {
            if (items[i]->getID()==id)
                return items[i]->clone();
        }
        return items[0]->clone();
    }
};

```

```

void client(ApplianceRegistry &a) {
    for (int i=1; i<=3; i++) {
        cout<<endl;
        cout<<"create a copy "<<i<<endl;
        Appliance *copy = a.createAppliance(i);
        copy->show();
        delete copy;
    }
}

int main() {
    ApplianceRegistry *r = new ApplianceRegistry();
    client(*r);
    delete r;

    return 0;
}

```

```

create a copy 1
ID: 1
Price: 700
Type: gas
Head: 2

create a copy 2
ID: 2
Price: 5000
Capacity: 30
Grill: yes

create a copy 3
ID: 3
Price: 16000
Capacity: 300
Door: 4

```

ควรใช้เมื่อไหร่

- เมื่อโค้ดของเราไม่ควรที่จะต้องไปเขียนอยู่กับ **concrete class** ของอ็อบเจกต์ที่เราต้องการจะก่อปั๊บ
 - บางครั้งเราอาจจะไม่รู้ว่า **concrete class** คือคลาสอะไร

ข้อดี ข้อเสีย

- ข้อดี
 - เราสามารถก่อปี้อ็อบเจกต์ได้โดยไม่ต้องไปยุ่งกับ **concrete class** ของอ็อบเจกต์นั้น
 - ไม่ต้องมีการเขียนโค้ดในการกำหนดค่าเริ่มต้นซ้ำๆ
 - สามารถสร้างอ็อบเจกต์ที่ซับซ้อนได้อย่างสະดวกسابายมากขึ้น
 - มีทางเลือกอื่นนอกเหนือจากการใช้ **inheritance**
- ข้อเสีย
 - อาจต้องระมัดระวังในกรณีที่ทำการก่อปี้อ็อบเจกต์ที่มี **circular references**

การบ้าน

- จงสร้างอีโคบเจกต์ โดยที่
 - ต้องแต่ละตัวเป็นต้องไม่ มีความกว้าง 80 cm. ยาว 120 cm. สูง 70 cm. สีน้ำตาล
 - เก้าอี้แต่ละตัวเป็นเก้าอี้ไม้ มีพนักพิงหลัง สีขาว

อ้างอิง

- <https://refactoring.guru/design-patterns/prototype>

Catalog of Design Patterns

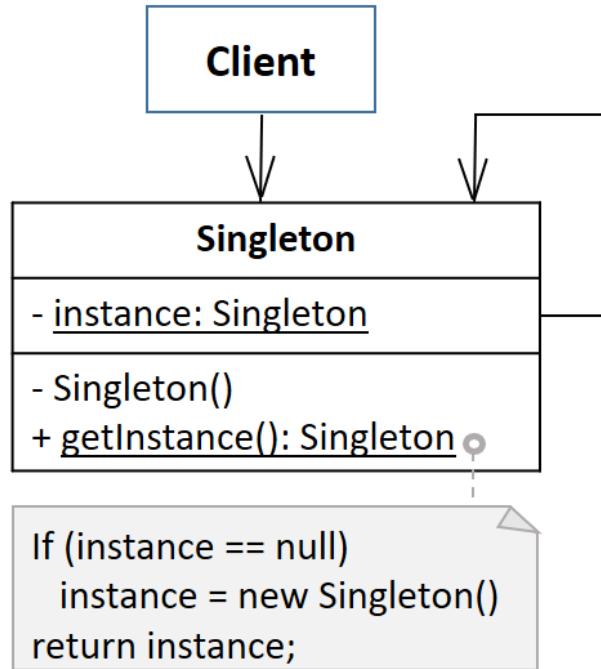
Creational Design Patterns

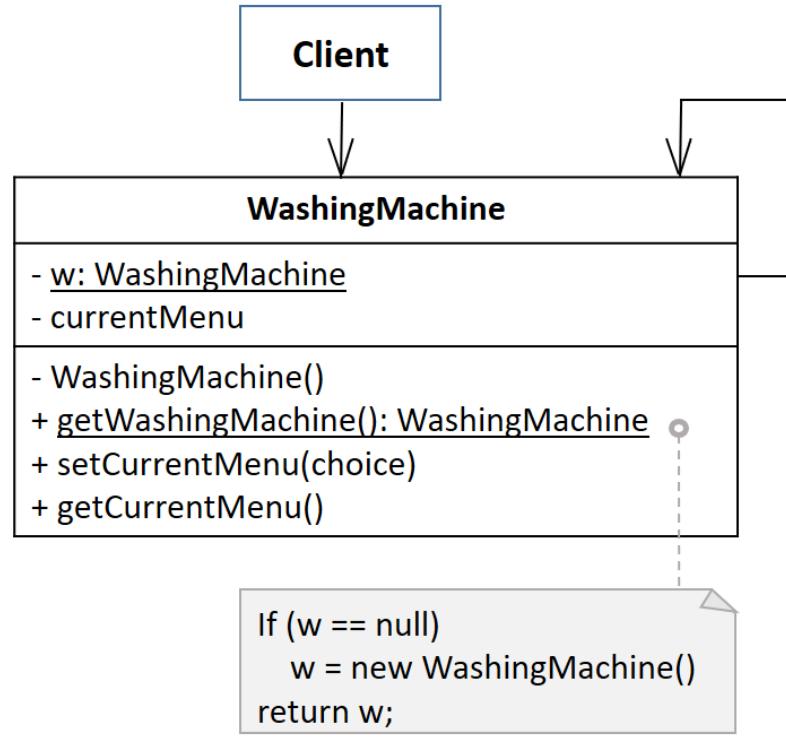
Singleton

Singleton

- คลาสจะมีแค่ 1 อ็อปเจกต์เท่านั้น และทุกคนสามารถเข้าถึงอ็อปเจกต์ตัวนี้ได้

Structure





```

#include <iostream>
using namespace std;

class WashingMachine {
    string currentMenu;
    static WashingMachine *w;

    WashingMachine() {
        cout<<"turn on" << endl;
        this->currentMenu = "Cotton";
    }
    WashingMachine(WashingMachine&);
    void operator=(WashingMachine&);

public:
    static WashingMachine *getWashingMachine();

    void setCurrentMenu(string s) {
        currentMenu = s;
    }
    string getCurrentMenu() const{
        return currentMenu;
    }
    ~WashingMachine() {
        cout<<"turn off" << endl;
        cout<<"last menu: "<<currentMenu << endl;
    }
};

WashingMachine* WashingMachine::w;

```

```

WashingMachine *WashingMachine::getWashingMachine() {
    if(w == NULL){
        w = new WashingMachine();
    }
    return w;
}

int main() {
    WashingMachine *w1 = WashingMachine::getWashingMachine();
    cout<<"current menu: "<<w1->getCurrentMenu() << endl;
    w1->setCurrentMenu("Sports wear");
    cout<<"current menu: "<<w1->getCurrentMenu() << endl;
    cout<< endl;

    WashingMachine *w2 = WashingMachine::getWashingMachine();
    w2->setCurrentMenu("Mix");
    cout<<"current menu: "<<w2->getCurrentMenu() << endl;
    cout<< endl;

    delete w1;
    return 0;
}

```

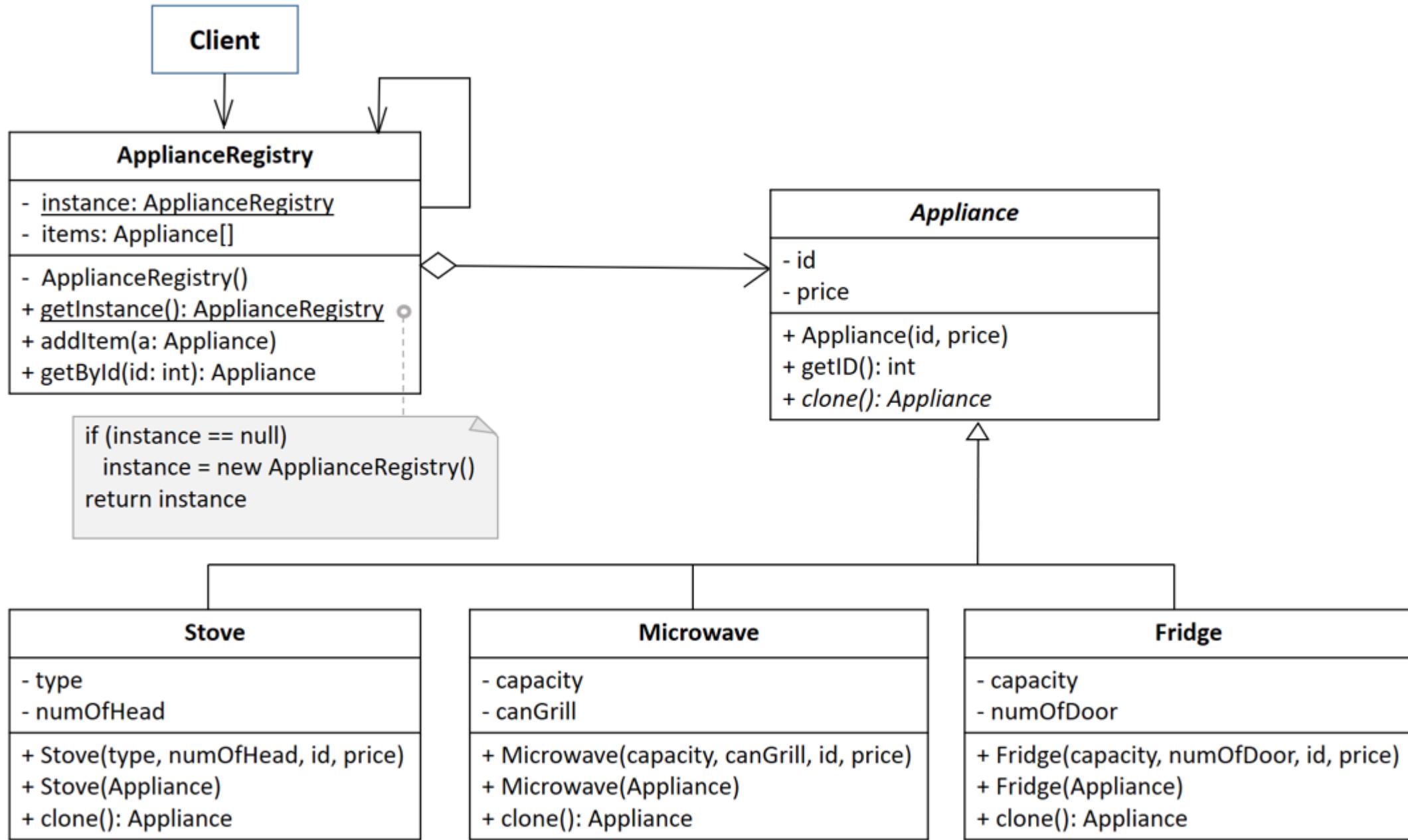
```

turn on
current menu: Cotton
current menu: Sports wear

current menu: Mix

turn off
last menu: Mix

```



```
#include <iostream>
#include <list>
using namespace std;

class Appliance {
protected:
    int id;
    double price;
public:
    Appliance() { id=0; price=0; }
    Appliance(int id, double price) {
        this->id = id;
        this->price = price;
    }
    virtual ~Appliance() {}
    virtual Appliance *clone() = 0;
    virtual void show() {
        cout<<"ID: "<<id<<endl;
        cout<<"Price: "<<price<<endl;
    }
    int getID() {
        return id;
    }
};
```

```
class Stove: public Appliance {
    int type;
    int numOfHead;
public:
    Stove() { type=0; numOfHead=0; }
    Stove(int type, int num, int id, double price):Appliance(id, price) {
        this->type = type;
        this->numOfHead = num;
    }
    Appliance *clone() {
        return new Stove(*this);
    }
    void show() {
        Appliance::show();
        cout<<"Type: "<<(type?"gas":"electric")<<endl;
        cout<<"Head: "<<numOfHead<<endl;
    }
    ~Stove() {
        cout<<"destroy stove"<<endl;
    }
};
```

```
class Microwave: public Appliance {
    int capacity;
    bool canGrill;
public:
    Microwave() { capacity=0; canGrill=false; }
    Microwave(int cap, bool grill, int id, double price):Appliance(id, price) {
        this->capacity = cap;
        this->canGrill = grill;
    }
    Appliance *clone() {
        return new Microwave(*this);
    }
    void show() {
        Appliance::show();
        cout<<"Capacity: "<<capacity<<endl;
        cout<<"Grill: "<<(canGrill?"yes":"no")<<endl;
    }
    ~Microwave() {
        cout<<"destroy microwave"<<endl;
    }
};
```

```
class Fridge: public Appliance {
    int capacity;
    int numOfDoor;
public:
    Fridge() { capacity=0; numOfDoor=0; }
    Fridge(int cap, int num, int id, double price):Appliance(id, price) {
        this->capacity = cap;
        this->numOfDoor = num;
    }
    Appliance *clone() {
        return new Fridge(*this);
    }
    void show() {
        Appliance::show();
        cout<<"Capacity: "<<capacity<<endl;
        cout<<"Door: "<<numOfDoor<<endl;
    }
    ~Fridge() {
        cout<<"destroy fridge"<<endl;
    }
};
```

```
class ApplianceRegistry {
    static ApplianceRegistry *instance;
    list <Appliance*> items;

    ApplianceRegistry() {
        cout<<"create registry" << endl;
    }
    ApplianceRegistry(ApplianceRegistry&);
    void operator=(ApplianceRegistry&);

public:
    ~ApplianceRegistry() {
        cout<<"destructor ApplianceRegistry" << endl;
        for (Appliance *a : items) {
            delete (a);
        }
    }
    Appliance *getById(int id) {
        for (Appliance *a : items) {
            if (a->getID() == id)
                return a->clone();
        }
        return items.front()->clone();
    }
    void addItem(Appliance *a) {
        items.push_back(a);
        cout<<"add item id: "<<a->getID() << endl;
    }
    static ApplianceRegistry *getInstance();
};
```

```
ApplianceRegistry *ApplianceRegistry::getInstance() {
    if (instance == NULL) {
        cout<<"create unique prototype" << endl;
        instance = new ApplianceRegistry();
    }
    else {
        cout<<"prototype exists" << endl;
    }
    return instance;
}
```

```

int main() {
    ApplianceRegistry *r1 = ApplianceRegistry::getInstance();

    r1->addItem(new Stove(1, 2, 1, 700));
    r1->addItem(new Microwave(30, true, 2, 5000));

    cout<<"...."<<endl;
    Appliance *c1 = r1->getById(1);
    c1->show();
    delete c1;

    cout<<endl;
    ApplianceRegistry *r2 = ApplianceRegistry::getInstance();

    r2->addItem(new Fridge(300, 4, 3, 16000));

    cout<<"...."<<endl;
    Appliance *c2 = r2->getById(3);
    c2->show();
    delete c2;

    cout<<endl;
    delete r1;
}

```

```

create unique prototype
create registry
add item id: 1
add item id: 2
....
ID: 1
Price: 700
Type: gas
Head: 2
destroy stove

prototype exists
add item id: 3
....
ID: 3
Price: 16000
Capacity: 300
Door: 4
destroy fridge

destructor ApplianceRegistry
destroy stove
destroy microwave
destroy fridge

```

ควรใช้เมื่อไหร่

- เมื่อคลาสของเรามีแค่อ็อบเจกต์ตัวเดียว
 - เข่น **database object** ความมีแค่ตัวเดียว

ข้อดี ข้อเสีย

- ข้อดี
 - มันใจได้ว่าคลาสเรา มีอ็อบเจกต์แค่ตัวเดียว
 - ทุกคนสามารถเข้าถึงอ็อบเจกต์ตัวเดียววนนั้นได้
 - อ็อบเจกต์จะถูกกำหนดค่าเริ่มต้นในครั้งแรกเพียงครั้งเดียวเท่านั้น
- ข้อเสีย
 - ขัดต่อหลักการ **Single Responsibility Principle**
 - เนื่องจาก pattern นี้ แก้ปัญหา 2 ปัญหาพร้อมๆ กัน
 - มีแค่อ็อบเจกต์เดียว และ มี **global access point** ในการเข้าถึงอ็อบเจกต์นี้

การบ้าน

- ให้นำ **prototype pattern** ที่นักศึกษาสร้างขึ้น มาปรับปรุงให้มีการสร้างต้นฉบับได้เพียงอันเดียว

อ้างอิง

- <https://refactoring.guru/design-patterns/singleton>

Catalog of Design Patterns

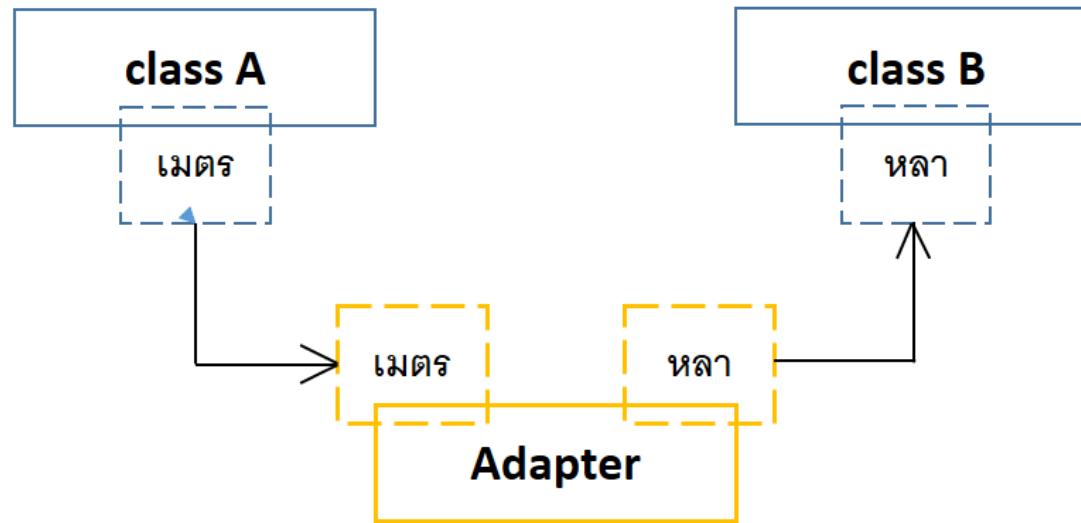
Structural Design Patterns

Adapter

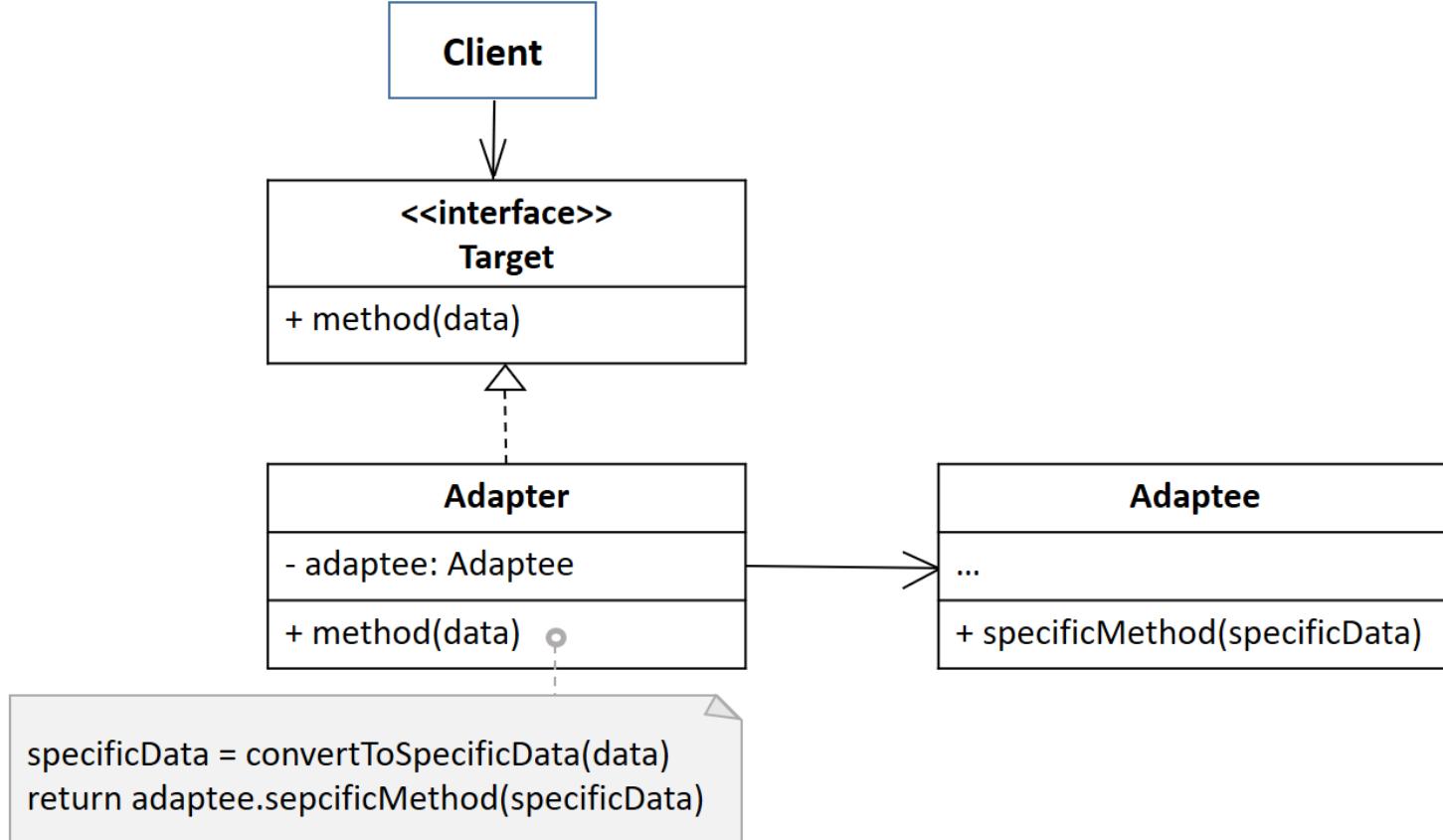
Adapter หรือ Wrapper

- ช่วยทำให้ออปเจกต์ที่เข้ากันไม่ได้ ให้สามารถทำงานร่วมกันได้

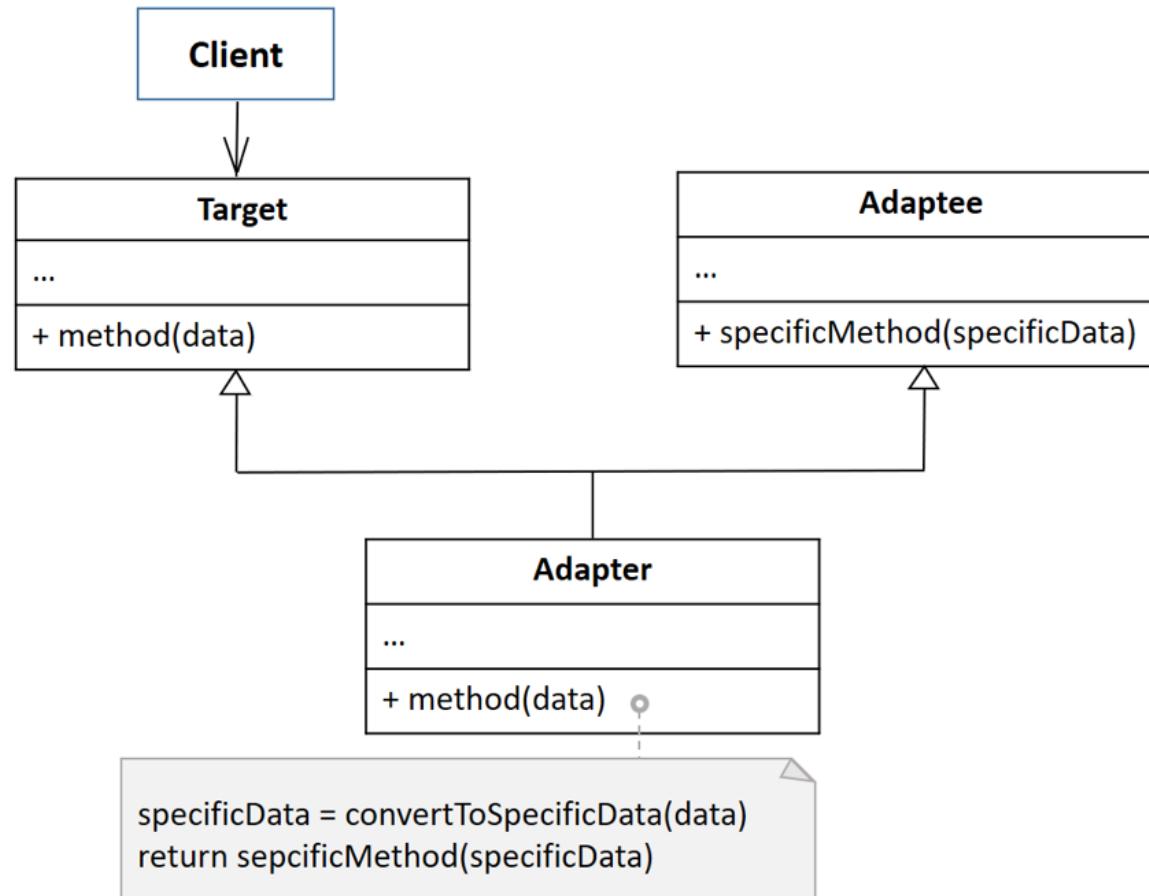


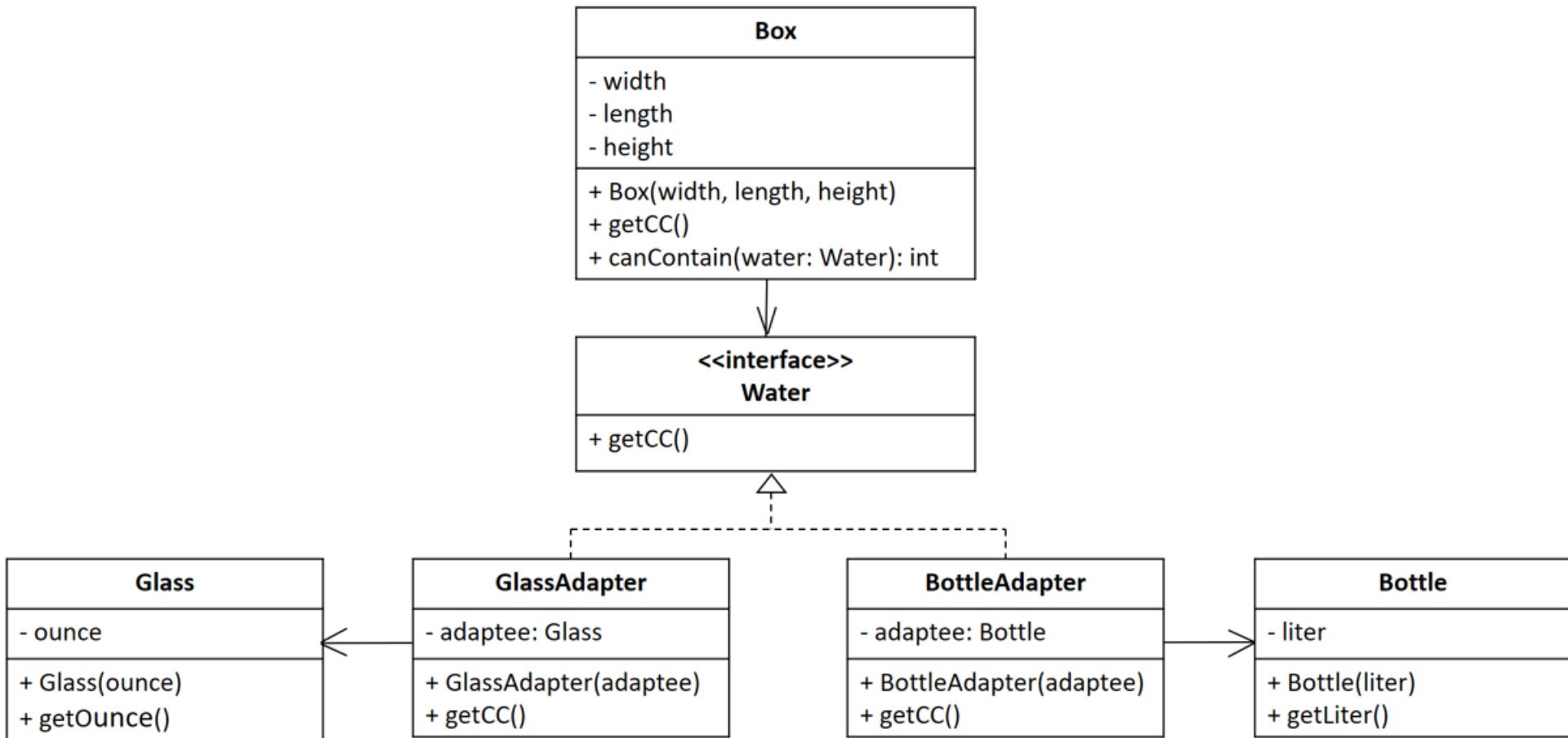


Structure



Structure (multiple inheritance)





```
#include <iostream>
using namespace std;

class Bottle {
    double liter;
public:
    Bottle() {
        set(1);
    }
    Bottle(double liter) {
        set(liter);
    }
    void set(double liter) {
        this->liter = liter;
    }
    double getLiter() {
        return liter;
    }
};
```

```
class Glass {
    double ounce;
public:
    Glass() {
        set(100);
    }
    Glass(double o) {
        set(o);
    }
    void set(double o) {
        ounce = o;
    }
    double getOunce() {
        return ounce;
    }
};
```

```
class Water {
public:
    virtual double getCC() = 0;
    virtual ~Water() {}
};

class BottleAdapter: public Water {
    Bottle *adaptee;
public:
    BottleAdapter(Bottle *b) {
        adaptee = b;
    }
    double getCC() {
        return adaptee->getLiter()*1000;
    }
};

class GlassAdapter: public Water {
    Glass *adaptee;
public:
    GlassAdapter(Glass *g) {
        adaptee = g;
    }
    double getCC() {
        return adaptee->getOunce()*30;
    }
};

class Box {
    double width;
    double length;
    double height;
public:
    Box() {
        set(1,1,1);
    }
    Box(int w, int l, int h) {
        set(w, l, h);
    }
    void set(int w, int l, int h) {
        width=w;
        length=l;
        height=h;
    }
    double getCC() {
        return width*length*height;
    }
    int canContain(Water *w) {
        return getCC()/w->getCC();
    }
};
```

```

int main() {
    Box *box = new Box(30,40,50);
    Bottle *bottle = new Bottle(2);
    BottleAdapter *adapterBottle = new BottleAdapter(bottle);
    Glass *glass = new Glass(5);
    GlassAdapter *adapterGlass = new GlassAdapter(glass);

    cout<<"bottle: "<<bottle->getLiter()<<" liter"<<endl;
    cout<<"glass: "<<glass->getOunce()<<" once"<<endl;
    cout<<endl;
    cout<<"box: "<<box->getCC()<<" cc"<<endl;

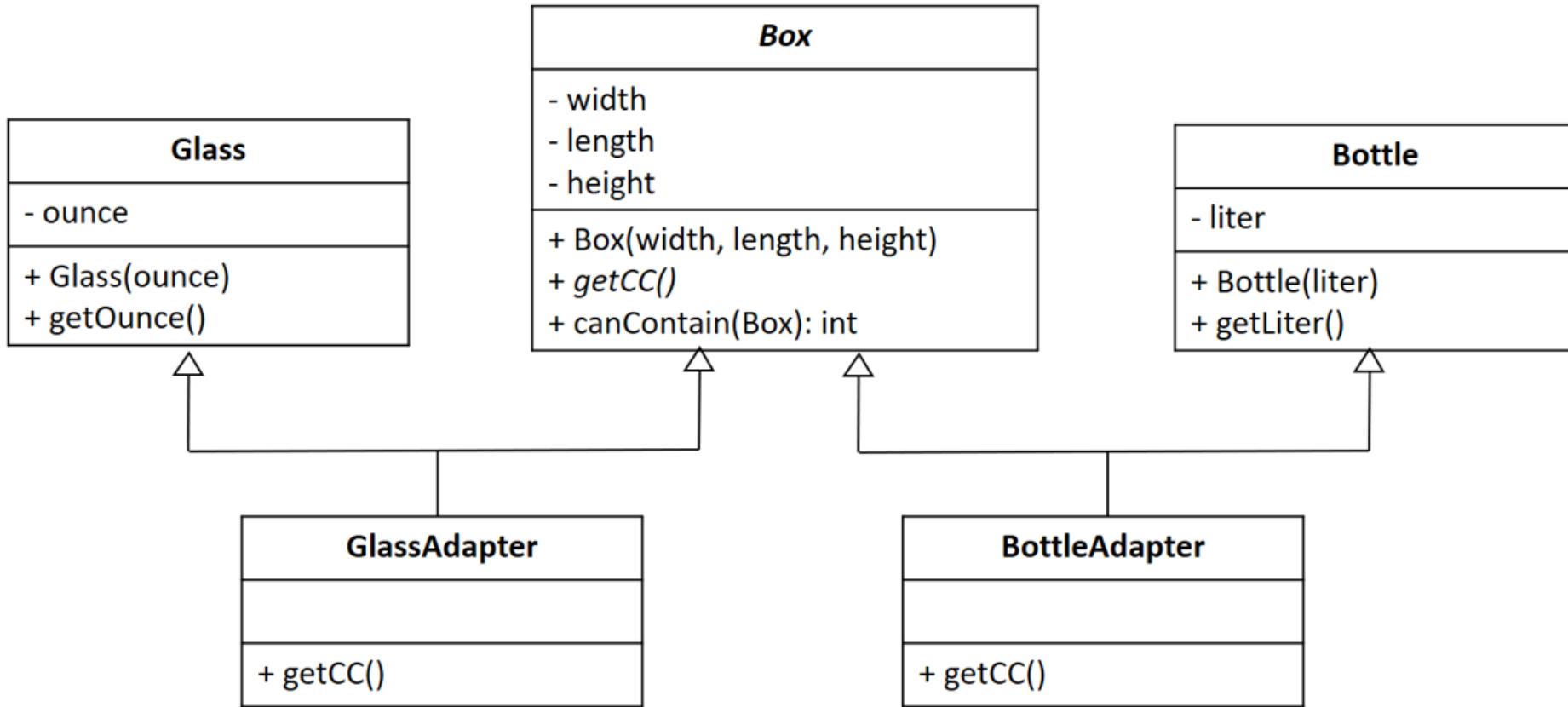
    cout<<"The box can contain water "<<box->canContain(adapterBottle)<<" bottle"<<endl;
    cout<<"The box can contain water "<<box->canContain(adapterGlass)<<" glass"<<endl;
    cout<<endl;

    box->set(10,20,10);
    cout<<"box: "<<box->getCC()<<" cc"<<endl;
    cout<<"The box can contain water "<<box->canContain(adapterBottle)<<" bottle"<<endl;
    cout<<"The box can contain water "<<box->canContain(adapterGlass)<<" glass"<<endl;

    delete box;
    delete bottle;
    delete adapterBottle;
    delete glass;
    delete adapterGlass;
    return 0;
}

```

bottle: 2 liter
 glass: 5 once
 box: 60000 cc
 The box can contain water 30 bottle
 The box can contain water 400 glass
 box: 2000 cc
 The box can contain water 1 bottle
 The box can contain water 13 glass



```
#include <iostream>
using namespace std;

class Bottle {
    double liter;
public:
    Bottle() {
        set(1);
    }
    Bottle(double liter) {
        set(liter);
    }
    Bottle(Bottle& b) {
        *this = b;
    }
    void set(double liter) {
        this->liter = liter;
    }
    double getLiter() {
        return liter;
    }
};
```

```
class Glass {
    double ounce;
public:
    Glass() {
        set(100);
    }
    Glass(double o) {
        set(o);
    }
    Glass(Glass &g) {
        *this = g;
    }
    void set(double o) {
        ounce = o;
    }
    double getOunce() {
        return ounce;
    }
};
```

```
class Box {
    double width;
    double length;
    double height;
public:
    Box() {
        set(1,1,1);
    }
    Box(int w, int l, int h) {
        set(w, l, h);
    }
    Box(Box &b) {
        *this = b;
    }
    virtual ~Box() {}
    void set(int w, int l, int h) {
        width=w;
        length=l;
        height=h;
    }
    virtual double getCC() {
        return width*length*height;
    }
    int canContain(Box *b) {
        return getCC()/b->getCC();
    }
};
```

```
class BottleAdapter: public Box, public Bottle {
public:
    BottleAdapter(Box *b, Bottle *t): Box(*b), Bottle(*t) {}
    double getCC() {
        return getLiter()*1000;
    }
};

class GlassAdapter: public Box, public Glass {
public:
    GlassAdapter(Box *b, Glass *g): Box(*b), Glass(*g) {}
    double getCC() {
        return getOunce()*30;
    }
};
```

```
int main() {
    Box *box = new Box(30,40,50);
    Bottle *bottle = new Bottle(2);
    BottleAdapter *adapterBottle = new BottleAdapter(box, bottle);
    Glass *glass = new Glass(5);
    GlassAdapter *adapterGlass = new GlassAdapter(box, glass);

    cout<<"bottle: "<<bottle->getLiter()<<" liter"<<endl;
    cout<<"glass: "<<glass->getOunce()<<" once"<<endl;
    cout<<endl;
    cout<<"box: "<<box->getCC()<<" cc"<<endl;

    cout<<"The box can contain water "<<box->canContain(adapterBottle)<<" bottle"<<endl;
    cout<<"The box can contain water "<<box->canContain(adapterGlass)<<" glass"<<endl;
    cout<<endl;

    box->set(10,20,10);
    cout<<"box: "<<box->getCC()<<" cc"<<endl;
    cout<<"The box can contain water "<<box->canContain(adapterBottle)<<" bottle"<<endl;
    cout<<"The box can contain water "<<box->canContain(adapterGlass)<<" glass"<<endl;

    delete box;
    delete bottle;
    delete adapterBottle;
    delete glass;
    delete adapterGlass;
    return 0;
}
```

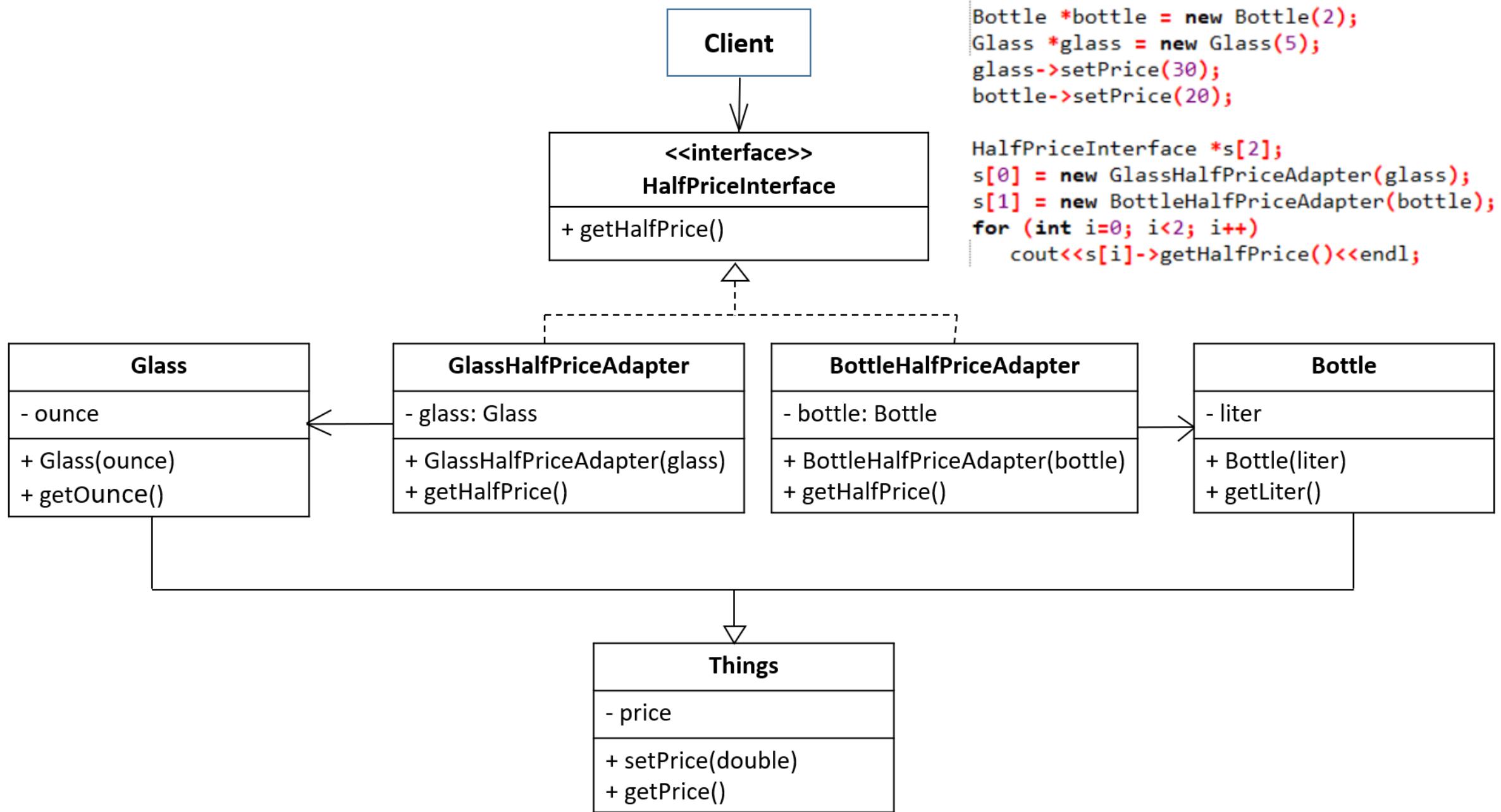
```
bottle: 2 liter
glass: 5 once

box: 60000 cc
The box can contain water 30 bottle
The box can contain water 400 glass

box: 2000 cc
The box can contain water 1 bottle
The box can contain water 13 glass
```

ควรใช้เมื่อไหร่

- เมื่อต้องการใช้คลาสที่มีอยู่แล้ว แต่ **interface** ของคลาสนั้นไม่สอดคล้องกับโ الدีดของเรา
- เมื่อต้องการจะ **reuse** บรรดาคลาสลูกที่มีอยู่แล้ว แต่คลาสลูกเหล่านั้นมีฟังก์ชันร่วมกันในคลาสมาก
- ให้นำฟังก์ชันนั้นไปไว้ใน **Adapter class** โดยที่มี **adaptee** เป็น **subclass**



ข้อดี ข้อเสีย

- ข้อดี
 - Single Responsibility Principle
 - แยกโค้ดที่ใช้ในการแปลงข้อมูลออกจากงานหลักของโปรแกรม
 - Open/Closed Principle
 - สามารถมี adapter ใหม่ๆ เพิ่มเข้าไปในโปรแกรมโดยไม่ต้องไปแก้โค้ดเดิม
- ข้อเสีย
 - โปรแกรมมีความซับซ้อนมากขึ้น

การบ้าน

- เชือก (Rope) ประกอบด้วยข้อมูล ความยาวของเชือก (หลา)
- ลูกбол (Ball) ประกอบด้วยข้อมูล รัศมี (นิ้ว) และมีฟังก์ชันในการหาขนาดเส้นรอบวง
- ลูกเต่า (cube) ประกอบด้วยข้อมูล ความยาวด้าน (เซนติเมตร) และมีฟังก์ชันในการหาความยาวรอบลูกเต่า
- ในคลาส Rope ให้สร้างฟังก์ชันในการตรวจสอบว่า ความยาวเชือกที่มีอยู่นั้นสามารถนำไปล้อมรอบลูกбол และลูกเต่าได้หรือไม่

อ้างอิง

- <https://refactoring.guru/design-patterns/adapter>

Catalog of Design Patterns

Structural Design Patterns

Bridge

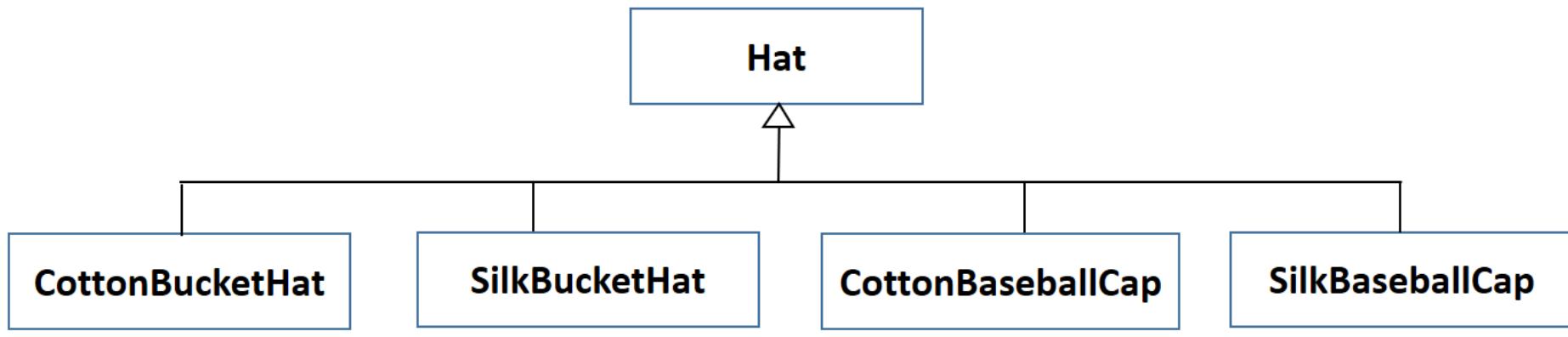
Bridge

- การแยกคลาสที่มีขนาดใหญ่ หรือแยกกลุ่มของคลาสที่มีความสัมพันธ์กัน โดยแยกเป็น 2 ส่วน คือ abstraction และ implementation ซึ่งแต่ละส่วนเป็นอิสระต่อกัน

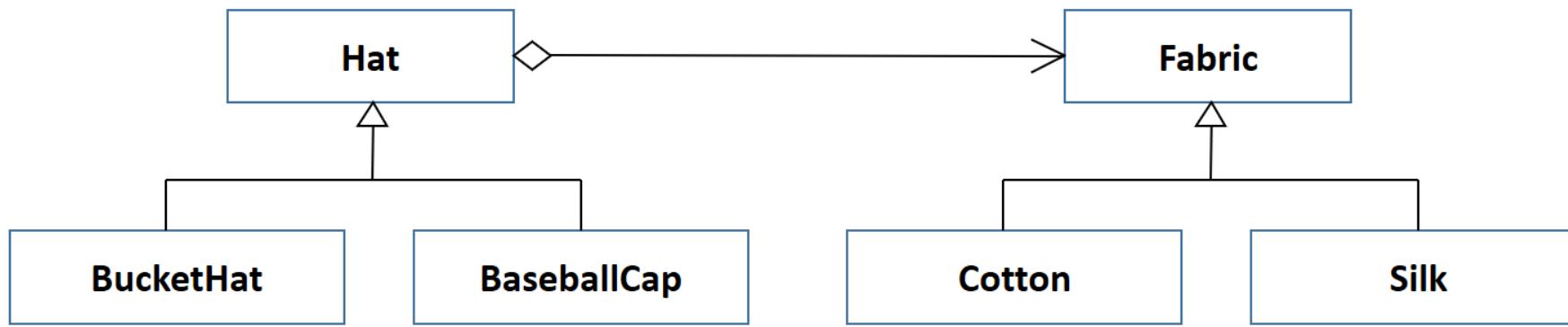
ប័ណ្ណ

- Hat
 - Bucket Hat
 - Baseball Cap
- Fabric
 - Cotton
 - Silk
- ឈាក់ខាងក្រោម
 - Cotton Bucket Hat
 - Silk Bucket Hat
 - Cotton Baseball Cap
 - Silk Baseball Cap

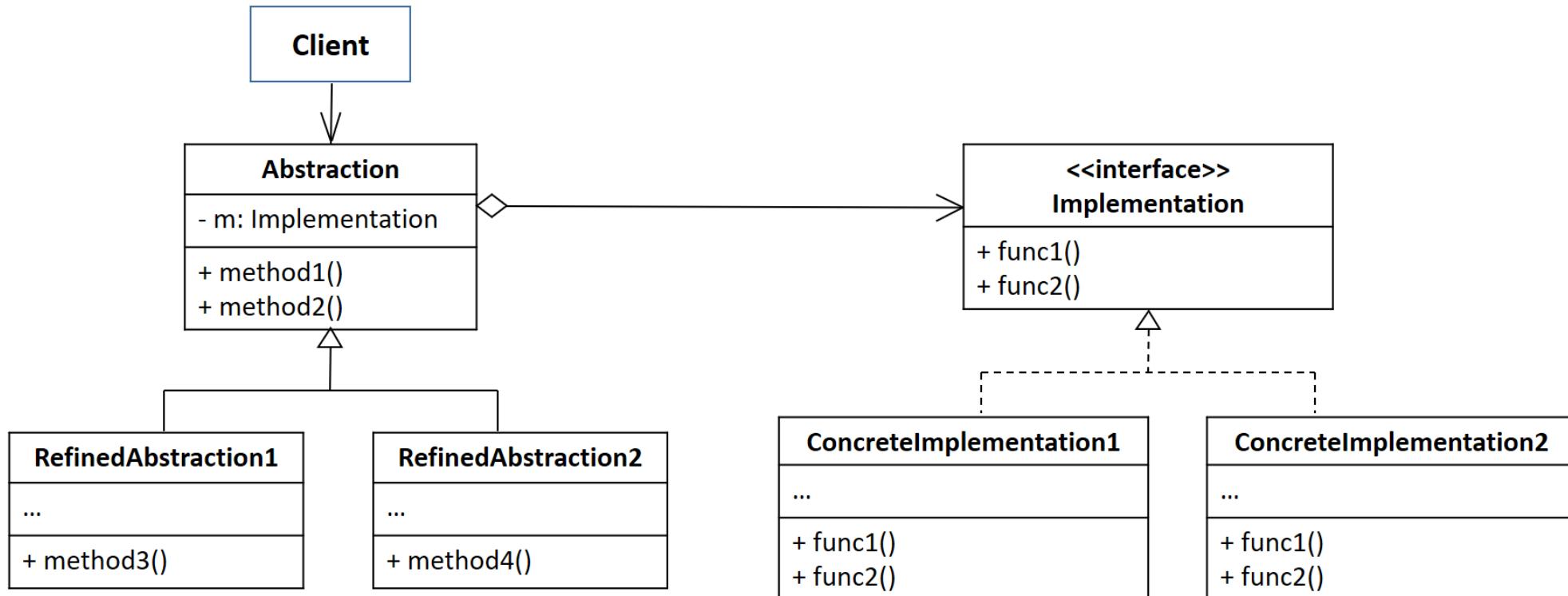


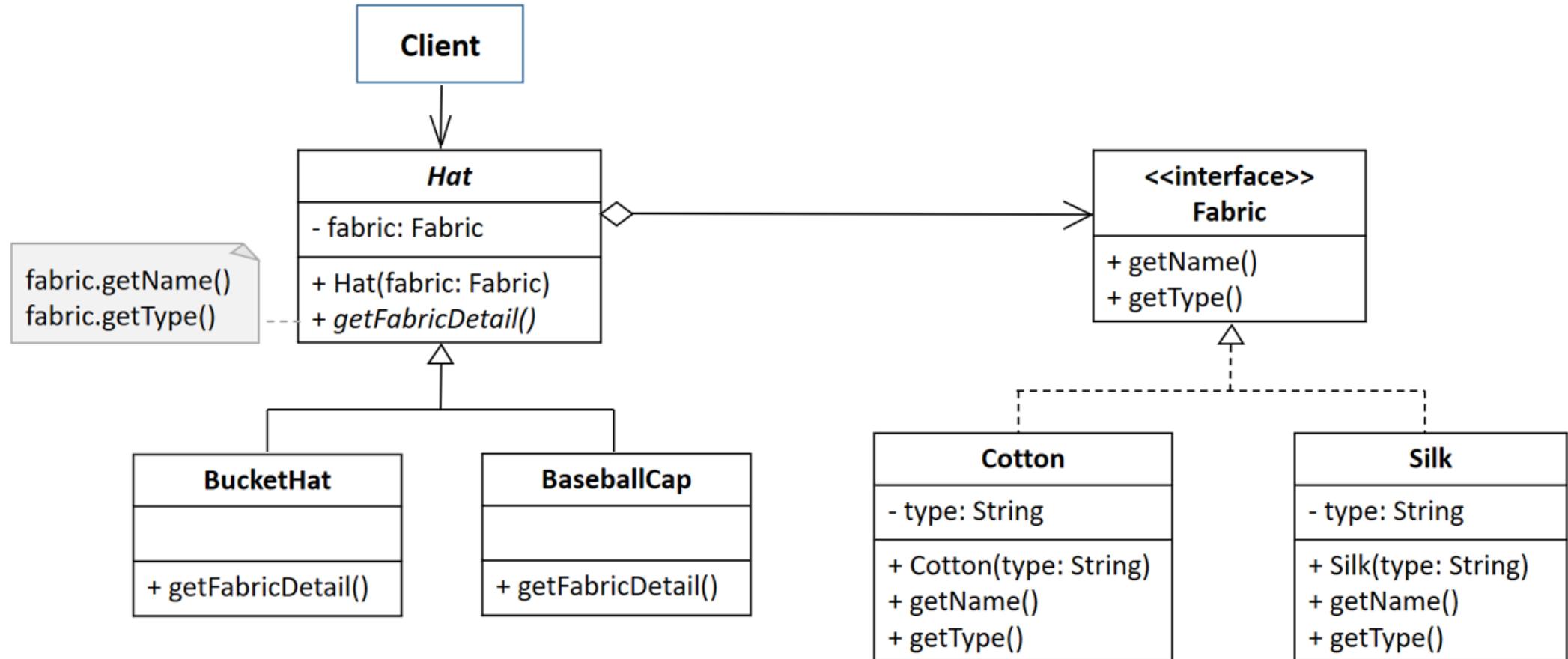


✗



Structure





```
#include <iostream>
using namespace std;

class Fabric {
public:
    virtual ~Fabric() {}
    virtual string getName() = 0;
    virtual string getType() = 0;
};

class Cotton: public Fabric {
    string type;
public:
    Cotton(string t) {
        type = t;
    }
    string getName() {
        return "Cotton Fabric";
    }
    string getType() {
        return type;
    }
};
```

```
class Silk: public Fabric {
    string type;
public:
    Silk(string t) {
        type = t;
    }
    string getName() {
        return "Silk Fabric";
    }
    string getType() {
        return type;
    }
};

class Hat {
protected:
    Fabric* fabric;

public:
    Hat(Fabric* f): fabric(f) {}

    virtual ~Hat() {}

    virtual string getFabricDetail() {
        return fabric->getName() + ": " + fabric->getType() + "\n";
    }
};
```

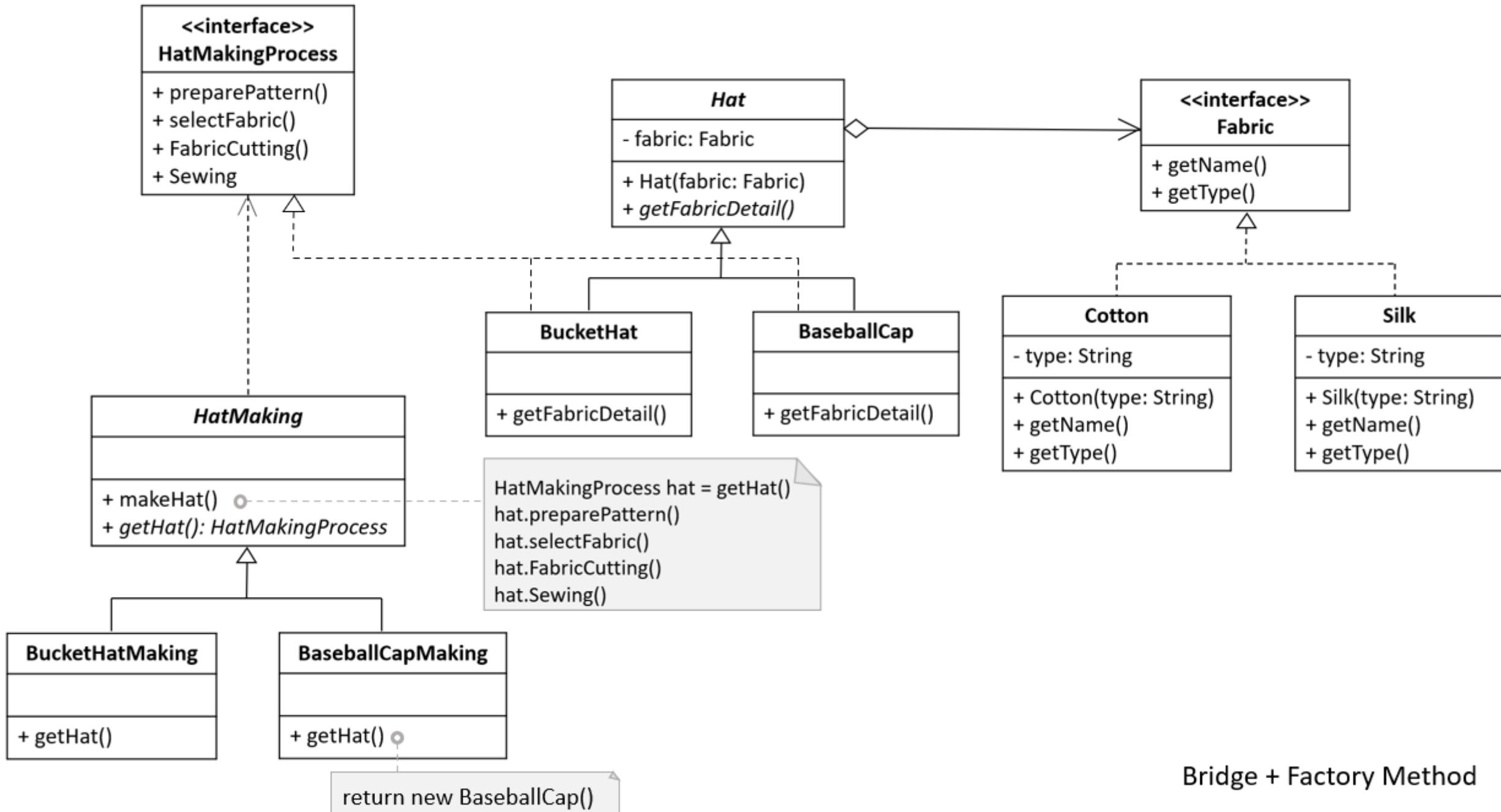
```
class BucketHat: public Hat {  
public:  
    BucketHat(Fabric* f): Hat(f) {}  
    string getFabricDetail() {  
        return "Bucket Hat >> "+Hat::getFabricDetail()+"\n";  
    }  
};  
  
class BaseballCap: public Hat {  
public:  
    BaseballCap(Fabric* f): Hat(f) {}  
    string getFabricDetail() {  
        return "Baseball Cap >> "+Hat::getFabricDetail()+"\n";  
    }  
};  
  
void client(Hat& hat) {  
    cout<<hat.getFabricDetail();  
}
```

```
int main() {  
    Fabric* fabric = new Silk("Thai Silk");  
    Hat* hat = new BucketHat(fabric);  
    client(*hat);  
    delete fabric;  
    delete hat;  
  
    fabric = new Cotton("100% Cotton");  
    hat = new BaseballCap(fabric);  
    client(*hat);  
    delete hat;  
  
    hat = new BucketHat(fabric);  
    client(*hat);  
    delete fabric;  
    delete hat;  
  
    return 0;  
}
```

Bucket Hat >> Silk Fabric: Thai Silk

Baseball Cap >> Cotton Fabric: 100% Cotton

Bucket Hat >> Cotton Fabric: 100% Cotton



Bridge + Factory Method

ควรใช้เมื่อไหร่

- เมื่อต้องการแบ่งคลาสที่ซึ่งมีขนาดใหญ่มากและมีฟังก์ชันการทำงานหลากหลายหน้าที่
- เมื่อต้องการเพิ่มความสามารถของคลาสให้มีหลากหลายมิติและเป็นอิสระในการจัดการ
 - แทนที่จะทำเอง ก็ให้คลาสที่เกี่ยวข้องเป็นคนทำ
- เมื่อต้องการเปลี่ยน **implementation** ในช่วงเวลา **runtime**

ข้อดี ข้อเสีย

- ข้อดี
 - Platform-independent
 - client สามารถทำงานกับ abstraction โดยไม่ต้องลงรายละเอียด
 - Open/Closed Principle
 - สามารถสร้าง abstraction และ implementation ใหม่ได้ และเป็นอิสระต่อกัน
 - Single Responsibility Principle
- ข้อเสีย
 - โค้ดมีความซับซ้อน

การบ้าน

- โต๊ะ (Table) และ เก้าอี้ (Chair) เป็นเฟอร์นิเจอร์ (Furniture) ที่ทำจากวัสดุ (material) แบบต่างๆ เช่น ไม้ (wood) และ พลาสติก (plastic)
- เราสามารถสร้าง โต๊ะไม้ โต๊ะพลาสติก เก้าอี้ไม้ และ เก้าอี้พลาสติก ได้
- ไม้มีหลายแบบ เช่น ไม้สัก ไม้ไผ่ ดังนั้นเราสามารถมี โต๊ะไม้สัก โต๊ะไม้ไผ่
- พลาสติก ก็มีหลายแบบ เช่น พลาสติกเกรดA พลาสติกเกรดB

อ้างอิง

- <https://refactoring.guru/design-patterns/bridge>

Catalog of Design Patterns

Structural Design Patterns

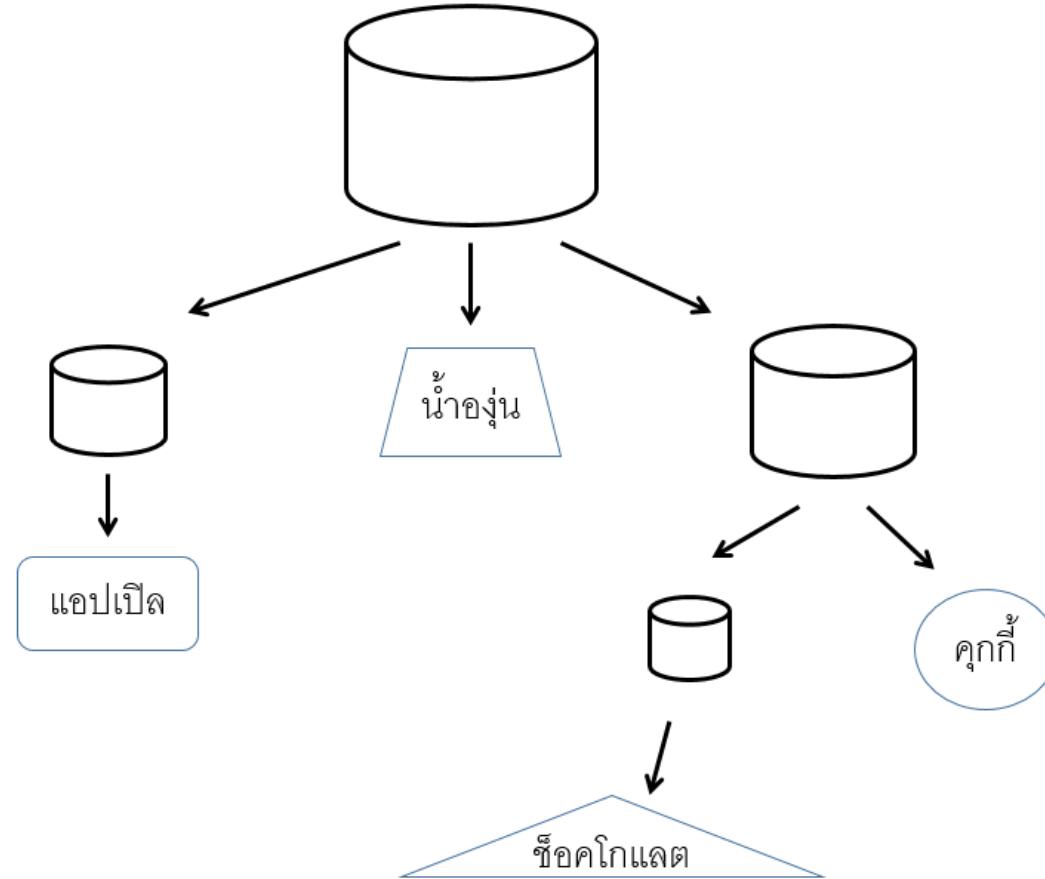
Composite

Composite (หรือ Object Tree)

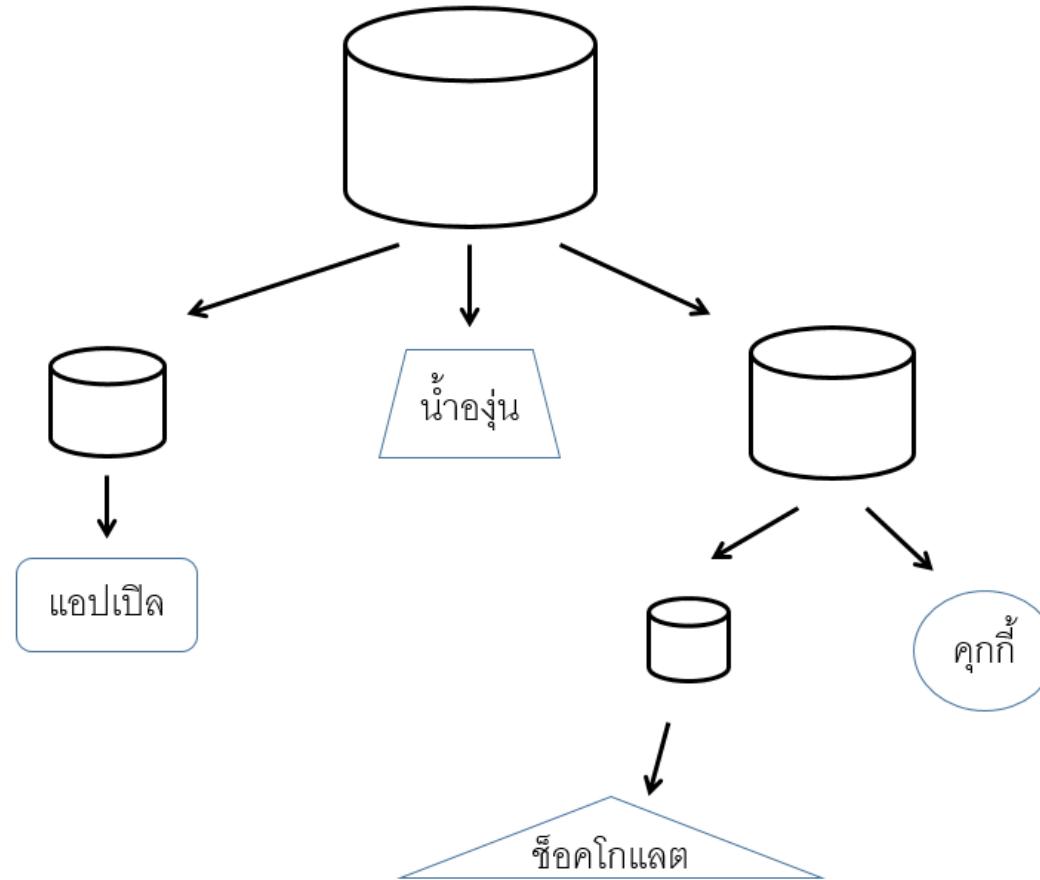
- การประกอบอ็อบเจกต์ขึ้นเป็นโครงสร้างต้นไม้ โดยที่เราสามารถทำงานกับโครงสร้างต้นไม้เหล่านี้ร่วงกับว่า แต่ละโครงสร้างต้นไม้มีนี่เป็นอ็อบเจกต์แต่ละตัว

- เมื่อกดกรุณีที่ app ของเราสามารถอยู่ในรูปแบบของ tree เท่านั้น

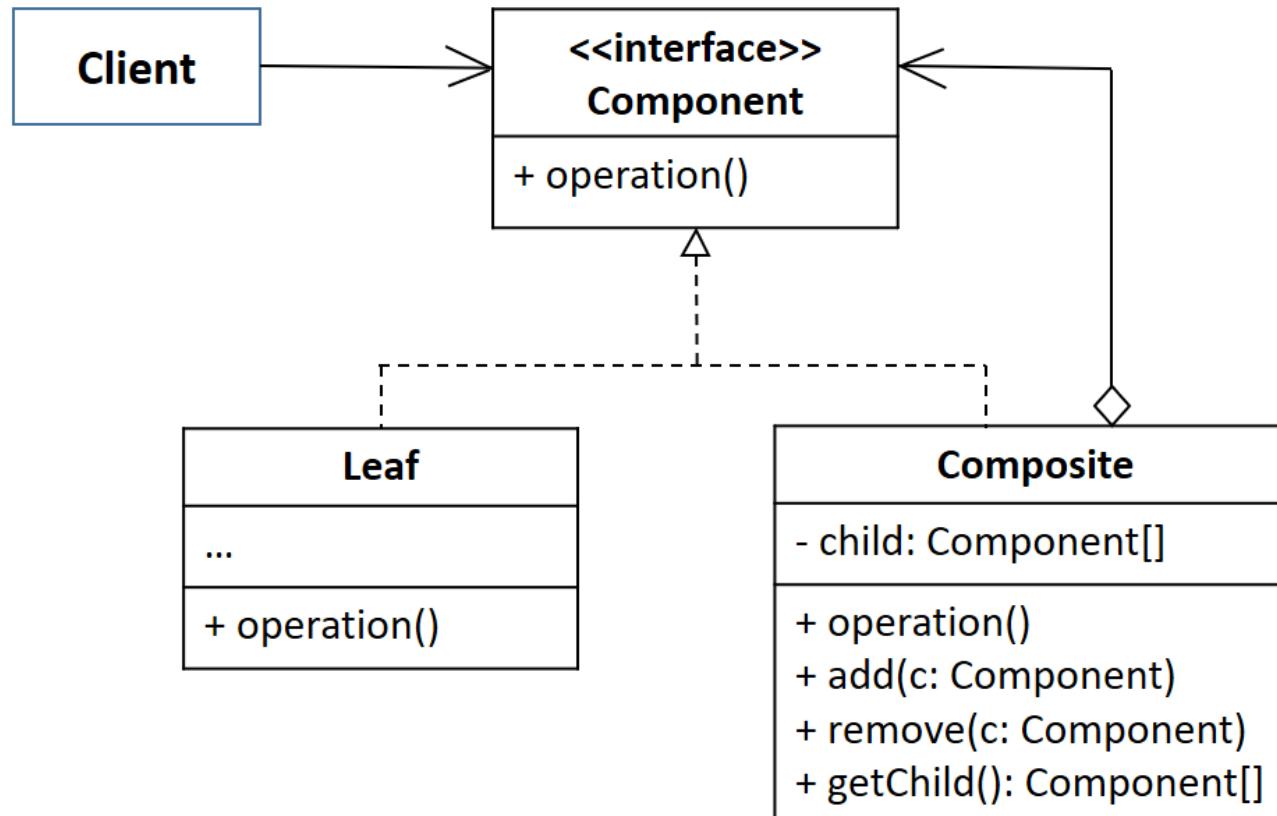
- จงหารากความ

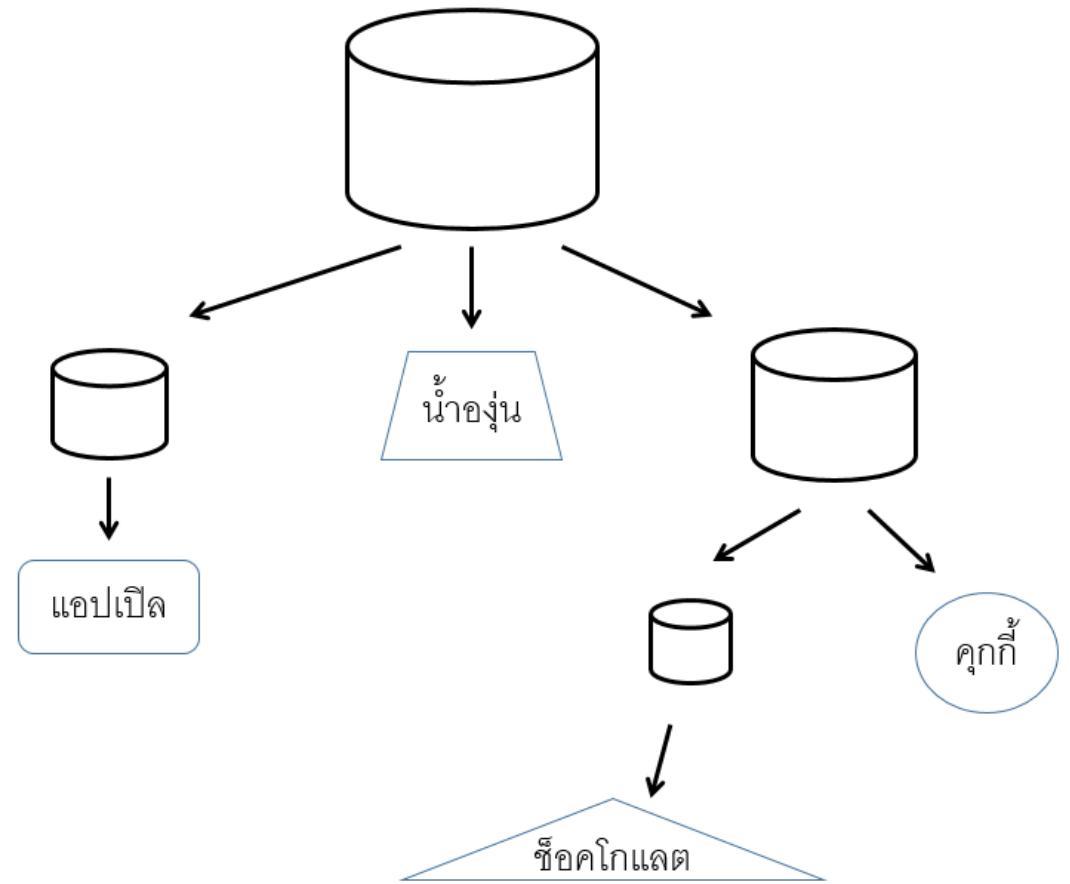
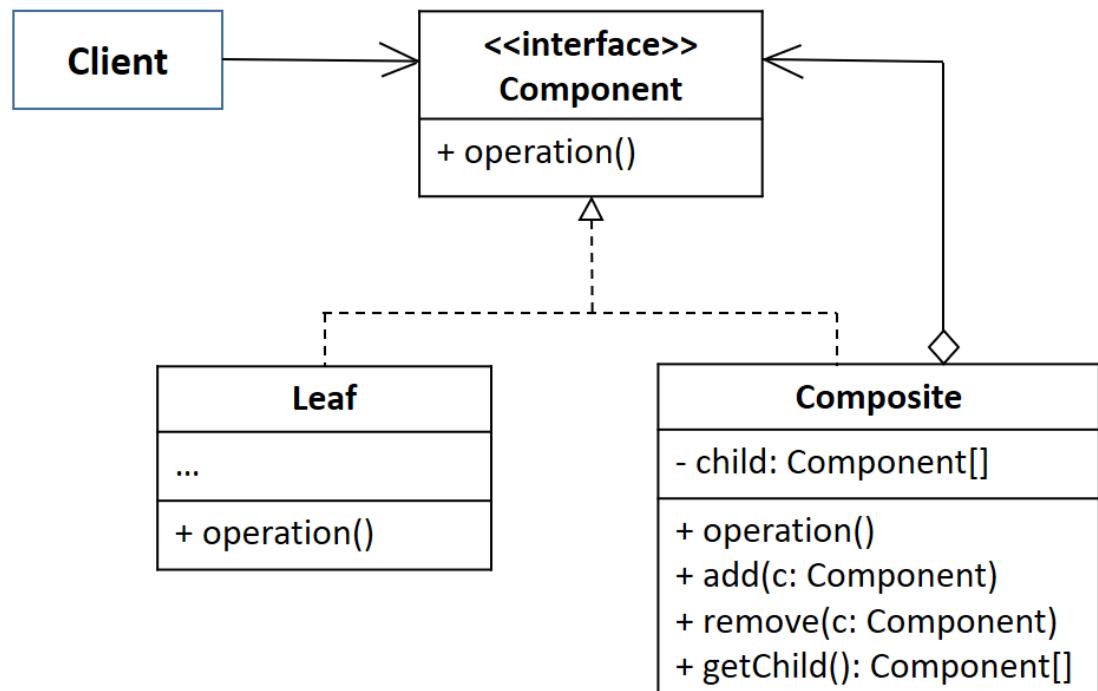


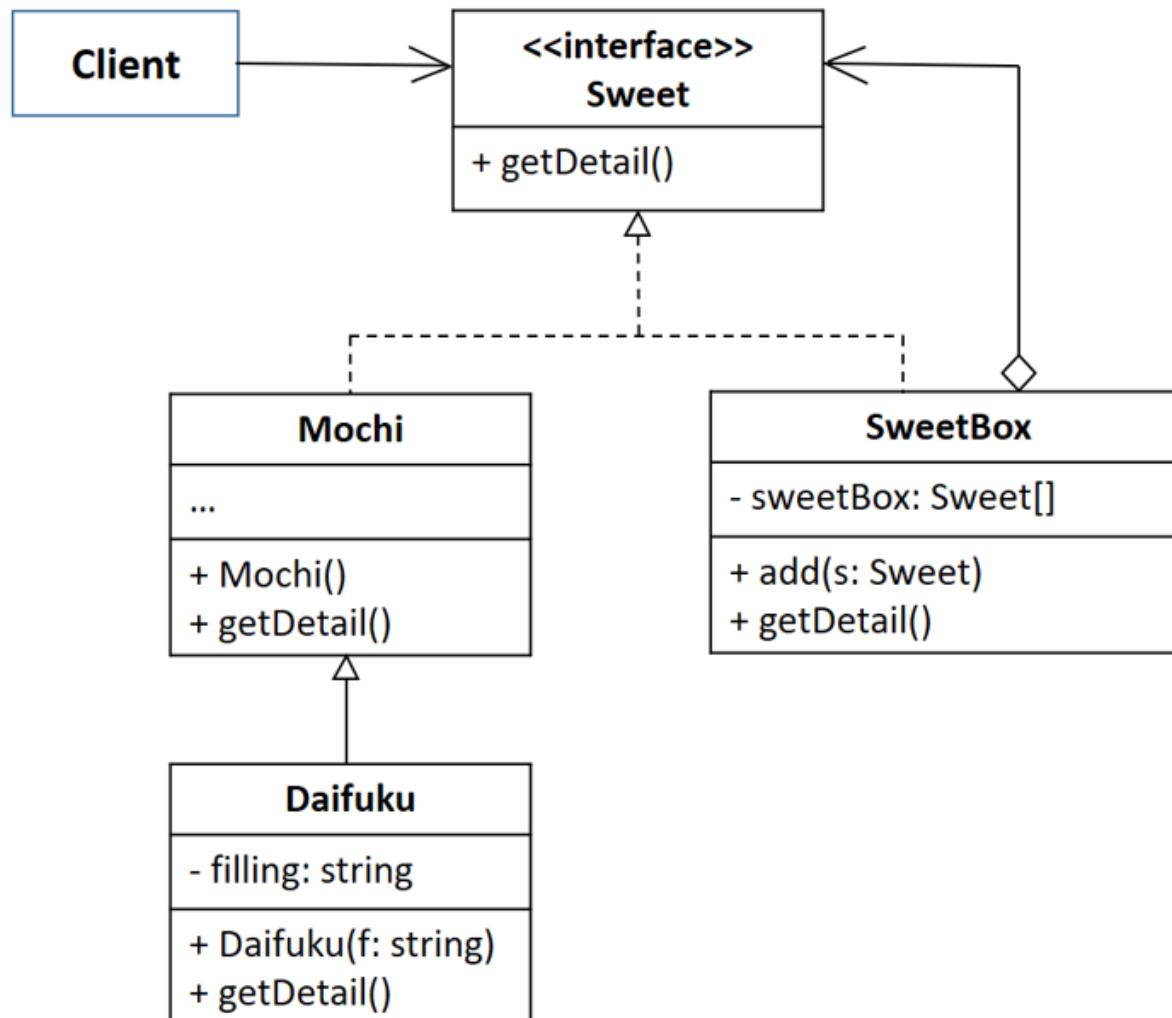
- ให้ Box กับ Product ใช้ Interface ร่วมกันที่มี method ในการคำนวณราคารวม

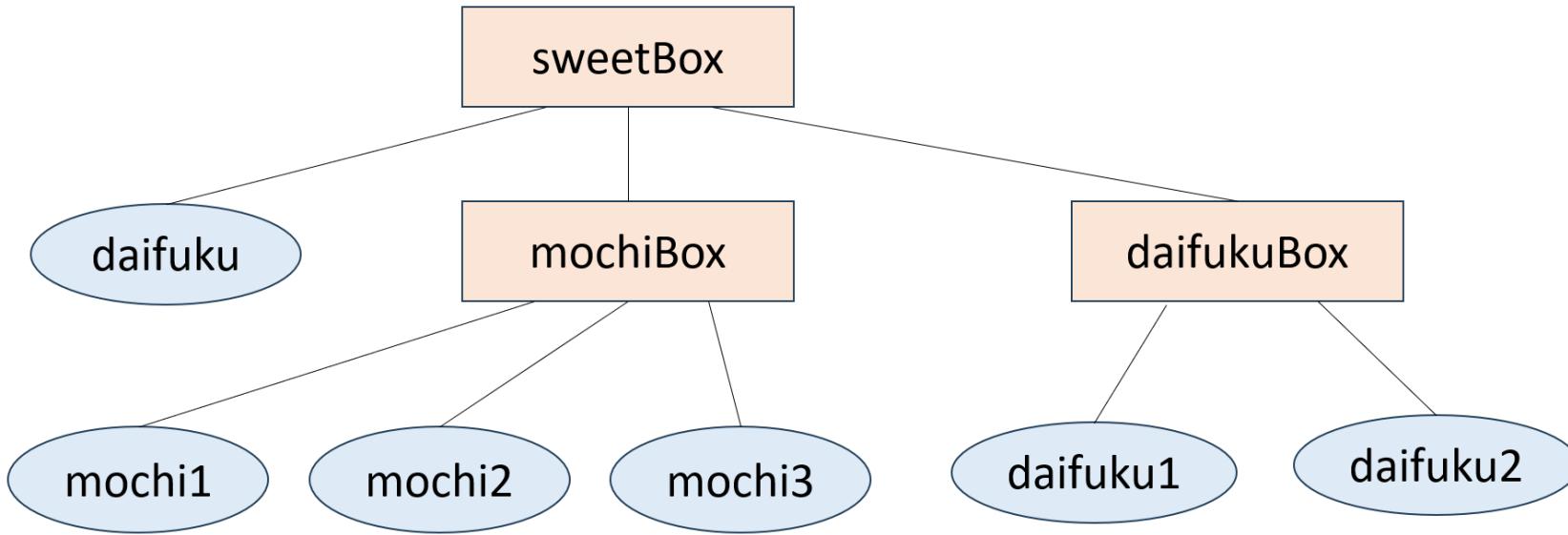


Structure









```
#include <iostream>
#include <list>

using namespace std;

class Sweet {
protected:
    Sweet *parent;

public:
    Sweet() {
        parent=0;
    }
    virtual ~Sweet() {}
    void setParent(Sweet *parent) {
        this->parent = parent;
    }
    Sweet *getParent() {
        return this->parent;
    }

    virtual bool isBox() {
        return false;
    }

    virtual void add(Sweet *s) { }
    virtual void remove(Sweet *s) { }

    virtual string getDetail() = 0;
};
```

```
class SweetBox: public Sweet {
protected:
    list <Sweet *> sweet;
    string name;

public:
    SweetBox(string name) {
        this->name=name;
    }
    void add(Sweet *s) {
        sweet.push_back(s);
        s->setParent(this);
    }
    bool isBox() {
        return true;
    }
    string getDetail() {
        string result;
        for ( list<Sweet*>::iterator it = sweet.begin(); it != sweet.end(); it++ ) {
            result += (*it)->getDetail();
        }
        return name + "... \n" + result;
    }
};
```

```
class Mochi: public Sweet {
public:
    string getDetail() {
        return " Mochi\n";
    }
};

class Daifuku: public Mochi {
    string filling;
public:
    Daifuku(string s) {
        filling=s;
    }
    string getDetail() {
        return " Daifuku " + filling + "\n";
    }
};
```

```

void client1(Sweet *sweet) {
    cout<<sweet->getDetail();
}

void client2(Sweet *s1, Sweet *s2) {
    if (s1->isBox()) {
        s1->add(s2);
    }
    cout<<s1->getDetail();
}

int main() {
    cout<<"Only 1 daifuku" << endl;
    Sweet *daifuku = new Daifuku("strawberry");
    client1(daifuku);
    cout<<endl << endl;

    Sweet *sweetBox = new SweetBox("Sweet Box");

    Sweet *mochiBox = new SweetBox("Mochi Box");
    Sweet *mochi1 = new Mochi();
    Sweet *mochi2 = new Mochi();
    Sweet *mochi3 = new Mochi();
    mochiBox->add(mochi1);
    mochiBox->add(mochi2);
    mochiBox->add(mochi3);

    Sweet *daifukuBox = new SweetBox("Daifuku Box");
    Sweet *daifuku1 = new Daifuku("green tea");
    Sweet *daifuku2 = new Daifuku("red bean");
    daifukuBox->add(daifuku1);
    daifukuBox->add(daifuku2);
}

```

```

sweetBox->add(daifuku);
sweetBox->add(mochiBox);
sweetBox->add(daifukuBox);

client1(sweetBox);
cout<<endl << endl;

client2(daifukuBox, daifuku);
cout<<endl;

cout<<"add more sweet box" << endl;
client2(sweetBox, daifukuBox);
cout<<endl;

delete daifuku;
delete daifuku2;
delete daifuku1;
delete mochi3;
delete mochi2;
delete mochi1;
delete mochiBox;
delete daifukuBox;
delete sweetBox;
}

```

```

Only 1 daifuku
Daifuku strawberry

Sweet Box...
Daifuku strawberry
Mochi Box...
Mochi
Mochi
Mochi
Daifuku Box...
Daifuku green tea
Daifuku red bean

Daifuku Box...
Daifuku green tea
Daifuku red bean
Daifuku strawberry

add more sweet box
Sweet Box...
Daifuku strawberry
Mochi Box...
Mochi
Mochi
Mochi
Daifuku Box...
Daifuku green tea
Daifuku red bean
Daifuku strawberry
Daifuku Box...
Daifuku green tea
Daifuku red bean
Daifuku strawberry

```

ควรใช้เมื่อไหร่

- เมื่อต้องการจะทำโครงสร้างในรูปแบบ tree
- เมื่อต้องการทำงานกับ element ที่มีโครงสร้างแบบง่าย และ element ที่มีโครงสร้างแบบซับซ้อน ให้ทั้งคู่สามารถทำงานได้ในรูปแบบเดียวกัน

ข้อดี ข้อเสีย

- ข้อดี
 - สามารถทำงานกับโครงสร้าง **tree** ที่ซับซ้อนได้อย่างสะดวก
 - ใช้ polymorphism, recursion
 - **Open/Closed Principle**
 - สามารถสร้าง **Type** ใหม่ได้โดยไม่ต้องแก้โค้ดเดิม
- ข้อเสีย
 - อาจเป็นการยากที่จะต้องสร้าง **Interface** สำหรับคลาสต่างๆ ที่มีฟังก์ชันการทำงานที่แตกต่างกันมากๆ

การบ้าน

- จงสร้างโครงสร้างขององค์กร ที่ประกอบด้วย ฝ่าย และ แผนก ต่างๆ ซึ่งเราสามารถอ้างได้ว่าเป็น unit ที่ประกอบด้วย unit ต่างๆ ได้
- แต่ละหน่วย มีข้อ และ หน้าที่การทำงาน
- เช่น
 - ฝ่ายการตลาด Marketing
 - แผนกขาย Sales
 - แผนกบริการ Service
 - แผนกวิจัยตลาด Market Research
 - ฝ่ายผลิต Production
 - ฝ่ายการเงิน Finance
 - ฝ่ายบุคคล Personnel

อ้างอิง

- <https://refactoring.guru/design-patterns/composite>

Catalog of Design Patterns

Structural Design Patterns

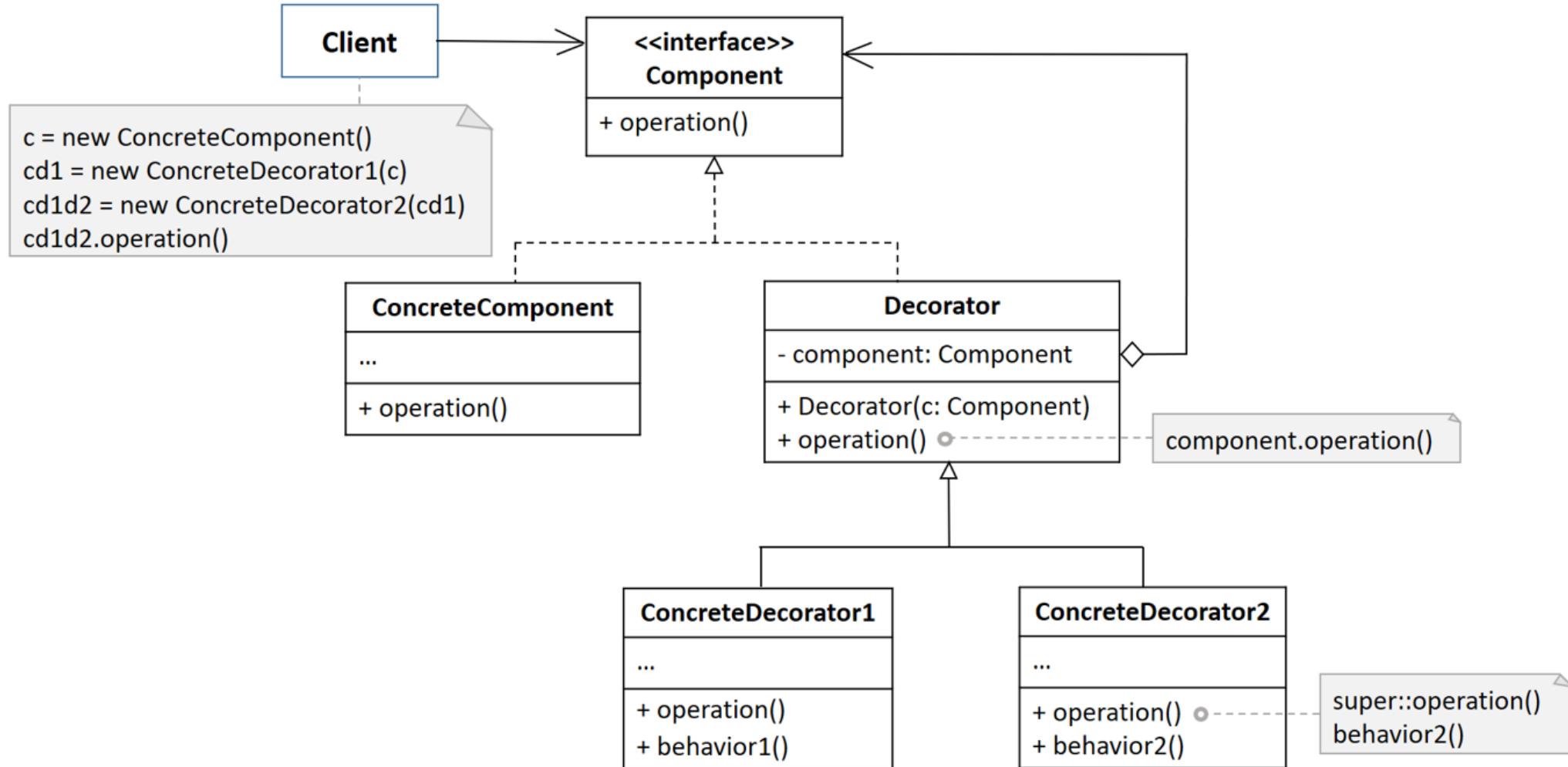
Decorator

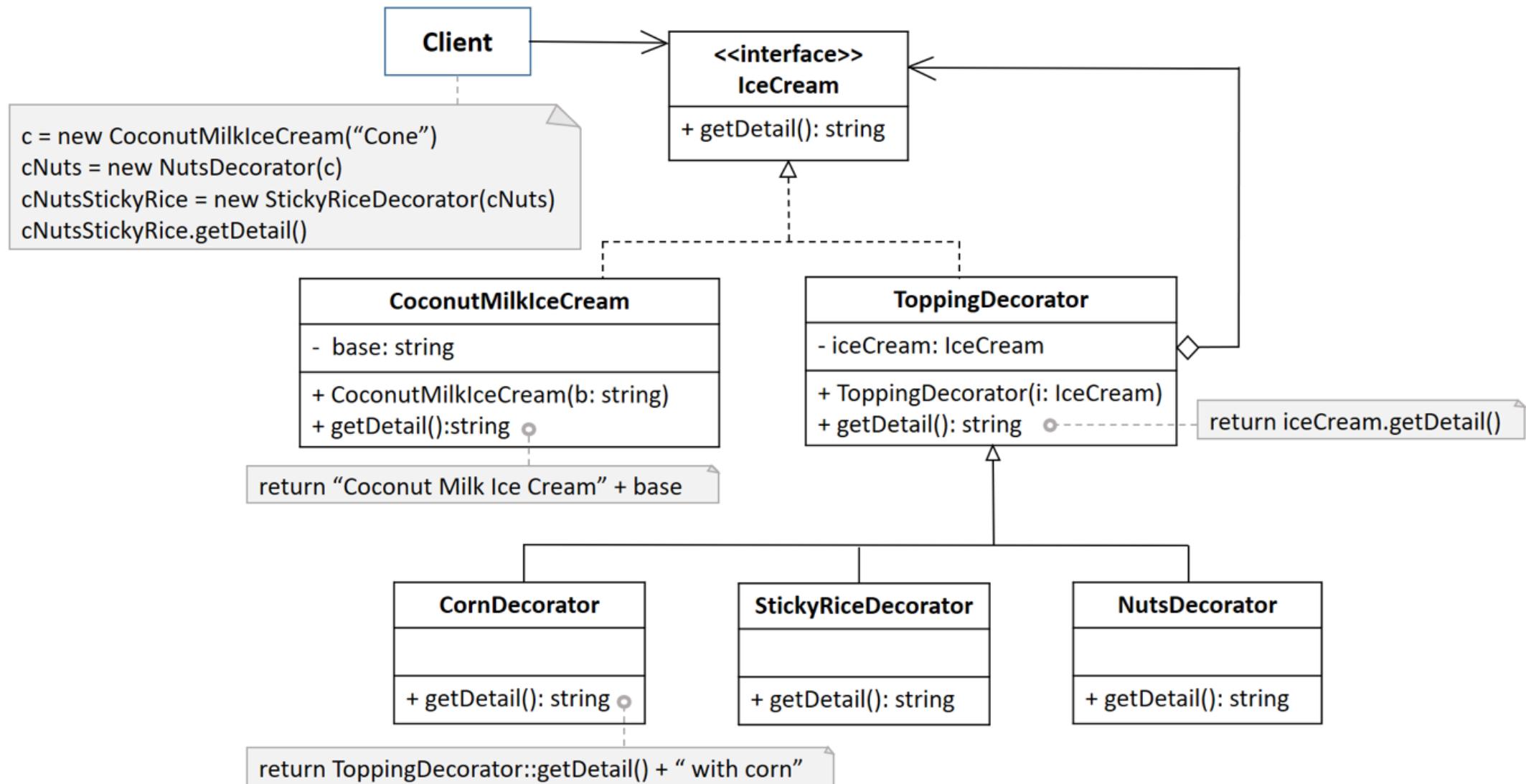
Decorator หรือ Wrapper

- การเพิ่ม behavior ใหม่ๆ ให้กับอ็อบเจกต์โดยไม่ต้องแก้ไขอ็อบเจกต์เดิม
- ทำได้โดย การสร้าง wrapper object ที่มี behavior ใหม่ๆ นั้น และนำอ็อบเจกต์ที่ต้องการเพิ่ม behavior นั้นใส่เข้าไปใน wrapper object

- ถ้าหาก ก็ใส่เสื้อกันหนาว
- ถ้ายังหนาวอยู่ ก็ใส่เสื้อโค้ททับ
- ถ้าฝนตก ก็ใส่เสื้อกันฝนอีกชั้นนึง

Structure





```
#include <iostream>
using namespace std;

class IceCream {
public:
    virtual ~IceCream() {}
    virtual string getDetail() = 0;
};

class CoconutMilkIceCream: public IceCream {
    string base;
public:
    CoconutMilkIceCream(string b) {
        base = b;
    }
    string getDetail() {
        return "Coconut milk icecream" + base;
    }
};

class ToppingDecorator: public IceCream {
protected:
    IceCream* iceCream;

public:
    ToppingDecorator(IceCream* m): iceCream(m) {}

    string getDetail() {
        return iceCream->getDetail();
    }
};

class CornDecorator: public ToppingDecorator {
public:
    CornDecorator(IceCream* m): ToppingDecorator(m) {}

    string getDetail() {
        return ToppingDecorator::getDetail() + "\n with corn";
    }
};

class StickyRiceDecorator: public ToppingDecorator {
public:
    StickyRiceDecorator(IceCream* m): ToppingDecorator(m) {}

    string getDetail() {
        return ToppingDecorator::getDetail() + "\n with sticky rice";
    }
};

class NutsDecorator: public ToppingDecorator {
public:
    NutsDecorator(IceCream* m): ToppingDecorator(m) {}

    string getDetail() {
        return ToppingDecorator::getDetail() + "\n with nuts";
    }
};
```

```
void client(IceCream* m) {
    cout<<m->getDetail()<<endl;
}

int main() {
    IceCream* iceCreamCone = new CoconutMilkIceCream(" cone ");
    client(iceCreamCone);
    cout<<endl;

    IceCream* iceCreamConeWithNuts = new NutsDecorator(iceCreamCone);
    IceCream* iceCreamConeWithNutsAndStickyRice = new StickyRiceDecorator(iceCreamConeWithNuts);
    client(iceCreamConeWithNutsAndStickyRice);
    cout<<endl;

    IceCream* iceCreamConeWithNutsAndStickyRiceAndCorn = new CornDecorator(iceCreamConeWithNutsAndStickyRice);
    cout<<iceCreamConeWithNutsAndStickyRiceAndCorn->getDetail()<<endl;

    delete iceCreamCone;
    delete iceCreamConeWithNuts;
    delete iceCreamConeWithNutsAndStickyRice;

    return 0;
}
```

Coconut milk icecream cone
Coconut milk icecream cone
with nuts
with sticky rice
Coconut milk icecream cone
with nuts
with sticky rice
with corn

ควรใช้เมื่อไหร่

- เมื่อต้องการเพิ่มพฤติกรรมใหม่ๆ ให้กับอ็อบเจกต์ในช่วงเวลา run-time โดยไม่ต้องไปแก้ไขโค้ดที่ใช้อ็อบเจกต์เหล่านี้
- เมื่อไม่สามารถเพิ่มพฤติกรรมของอ็อบเจกต์โดยใช้ **inheritance**
 - เช่น **final class**

ข้อดี ข้อเสีย

- ข้อดี
 - สามารถเพิ่มพฤติกรรมของอ็อบเจกต์ได้โดยไม่ต้องสร้างคลาสลูก
 - สามารถเพิ่มหรือลบหน้าที่ความรับผิดชอบจากอ็อบเจกต์ได้ในช่วงเวลา.ran ใหม่
 - สามารถรวมหลาย ๆ พฤติกรรมได้โดยใช้ **decorator** หลายอัน
 - **Single Responsibility Principle**
- ข้อเสีย
 - อาจจะยากถ้าต้องการจะลบ **wrapper** บางตัวออกจากกลุ่มของ **wrapper** ที่กองซ้อนกัน

การบ้าน

- radix หน้า เส้นหมี่ เส้นไข่ปู หมี่กรอบ (component)
- ใส่ หมู ปลา กุ้ง ปลาหมึก (decorator)

อ้างอิง

- <https://refactoring.guru/design-patterns/decorator>

Catalog of Design Patterns

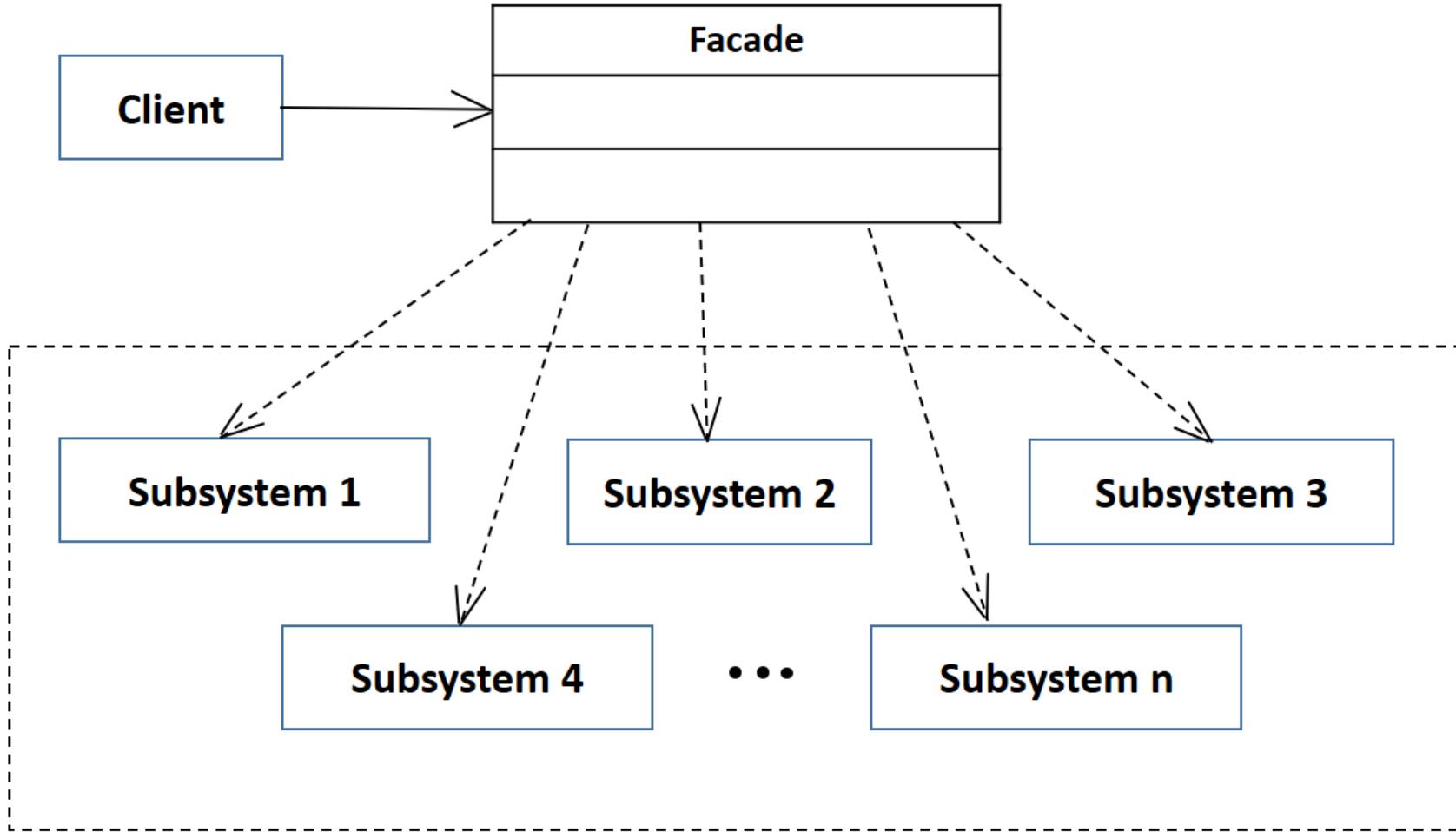
Structural Design Patterns

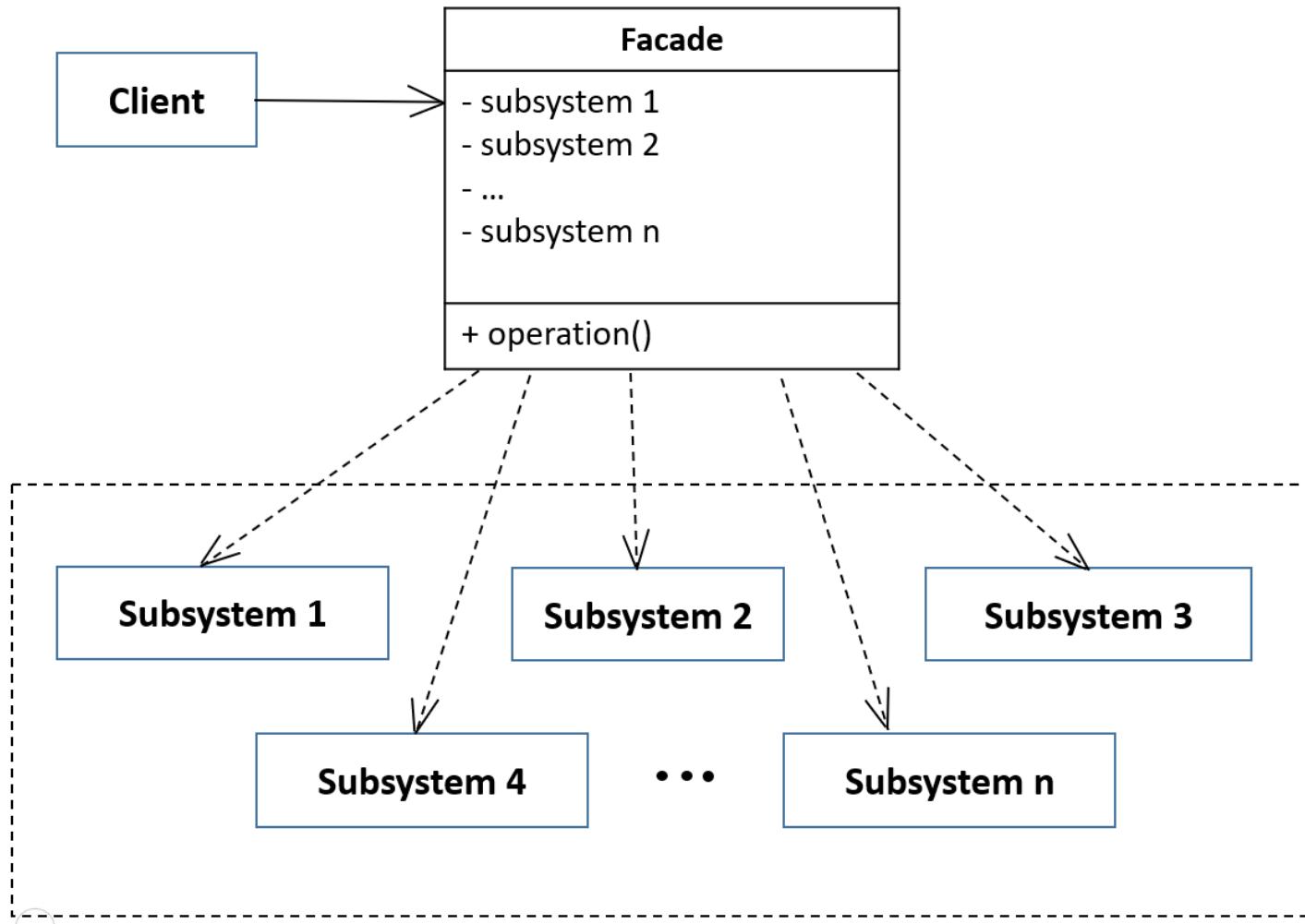
Facade

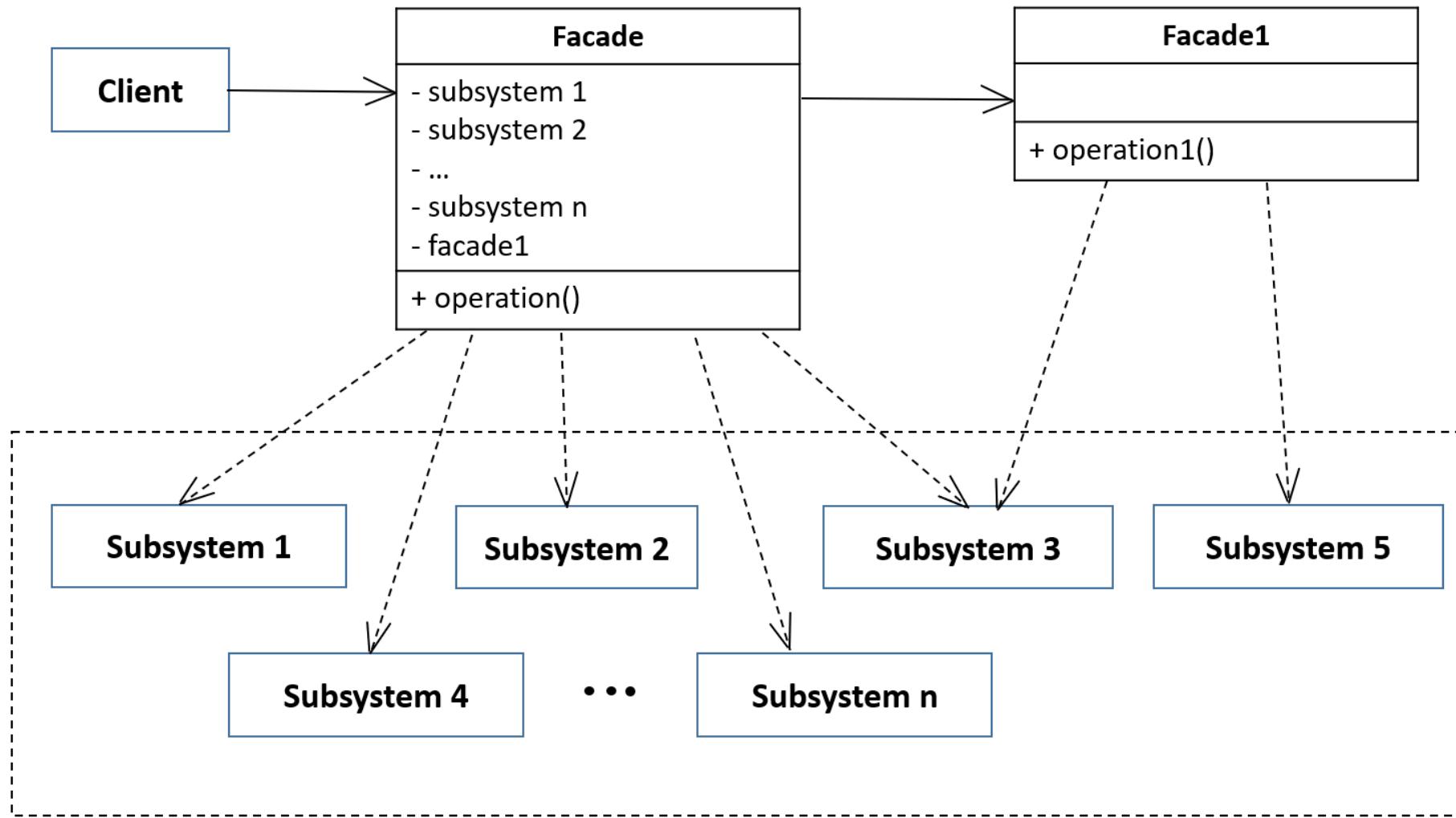
Facade

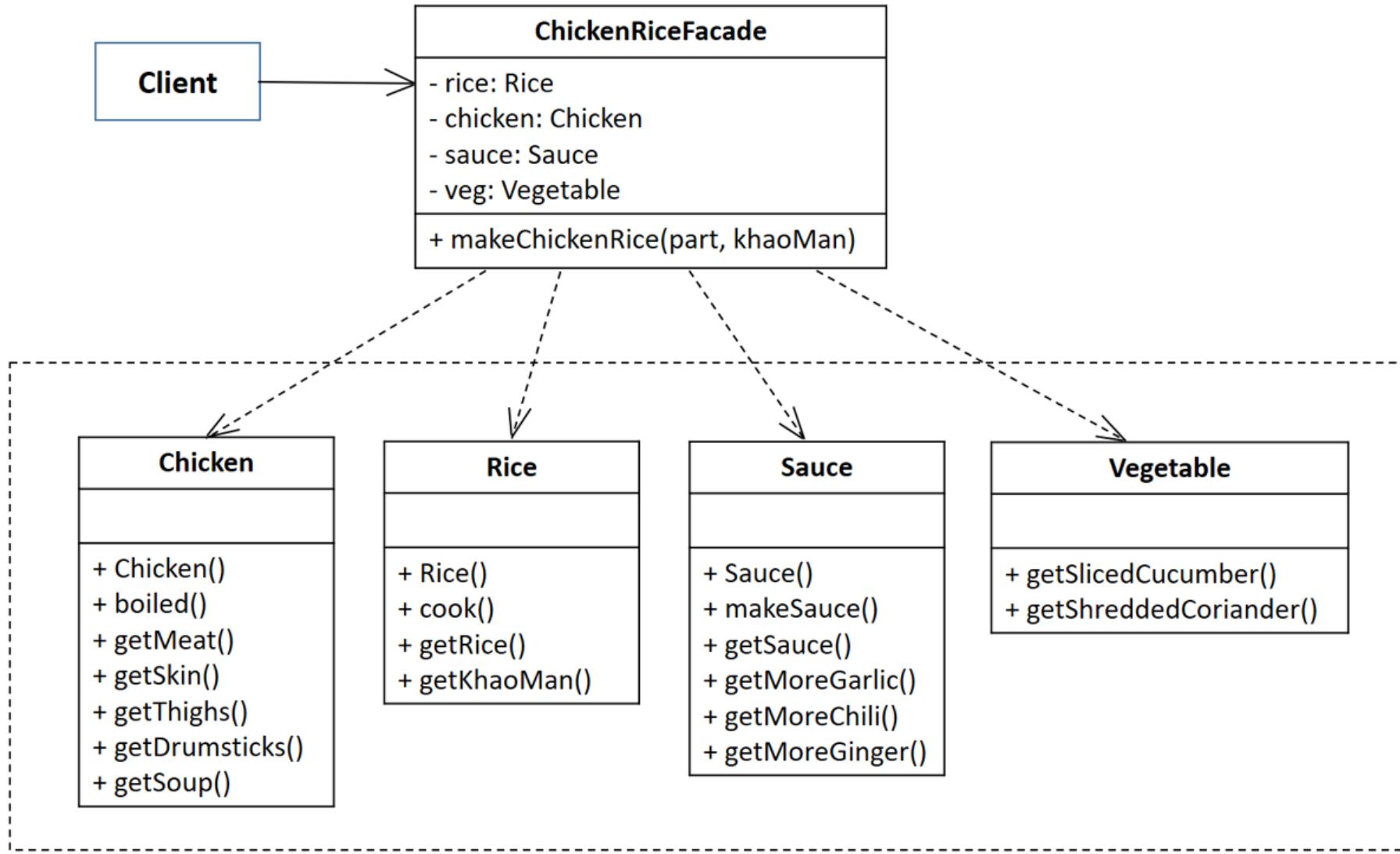
- สร้างอินเตอร์เฟสอย่างง่ายให้กับกลุ่มของคลาสที่มีความซับซ้อน

Structure









```
#include <iostream>
using namespace std;

class Rice {
public:
    Rice() {
        cook();
    }
    void cook() {
        cout<<"cooking rice"<<endl;
    }
    string getRice() {
        return "Khao ";
    }
    string getKhaoMan() {
        return "Khao Man ";
    }
};

class Vegetable {
public:
    string getSlicedCucumber() {
        return "sliced cucumber ";
    }
    string getShreddedCoriander() {
        return "shredded coriander ";
    }
};
```

```
class Chicken {
public:
    Chicken() {
        boiled();
    }
    void boiled() {
        cout<<"boiled chicken"<<endl;
    }
    string getChickenMeat() {
        return "Gai only Meat ";
    }
    string getChickenSkin() {
        return "Gai ";
    }
    string getChickenThighs() {
        return "Gai Sapok ";
    }
    string getChickenDrumSticks() {
        return "Gai Nua Nong ";
    }
    string getChickenSoup() {
        return "+ Nam Soup ";
    }
};
```

```
class Sauce {
public:
    Sauce() {
        makeSauce();
    }
    void makeSauce() {
        cout<<"making sauce"<<endl;
    }
    string getSauce() {
        return "Nam Jim Khao Man Gai ";
    }
    string getMoreGarlic() {
        return "+ garlic ";
    }
    string getMoreChili() {
        return "+ chili ";
    }
    string getMoreGinger() {
        return "+ ginger ";
    }
};
```

```
class ChickenRiceFacade {
protected:
    Rice *rice;
    Chicken *chicken;
    Sauce *sauce;
    Vegetable *veg;

public:
    ChickenRiceFacade(Rice *r=0, Chicken *c=0, Sauce *s=0, Vegetable *v=0) {
        rice = r ? : new Rice;
        chicken = c ? : new Chicken;
        sauce = s ? : new Sauce;
        veg = v ? : new Vegetable;
    }

    ~ChickenRiceFacade() {
        delete rice;
        delete chicken;
        delete sauce;
        delete veg;
    }

    void client(ChickenRiceFacade *facade) {
        cout<<"\norder 1"<<endl;
        cout<<facade->makeChickenRice("drumstick",true);
        cout<<endl;
        cout<<"\norder 2"<<endl;
        cout<<facade->makeChickenRice("thigh",false);
        cout<<endl;
        cout<<"\norder 3"<<endl;
        cout<<facade->makeChickenRice("",true);
    }
}
```

```
string makeChickenRice(string part, bool khaoman) {
    string result = "";

    if (khaoman)
        result+=rice->getKhaoMan();
    else
        result+=rice->getRice();

    if (part == "meat")
        result+=chicken->getChickenMeat();
    else if (part == "thigh")
        result+=chicken->getChickenThighs();
    else if (part == "drumstick")
        result+=chicken->getChickenDrumSticks();
    else
        result+=chicken->getChickenSkin();

    result+=chicken->getChickenSoup();

    result+="\n";
    result+=sauce->getSauce();
    result+=sauce->getMoreChili();
    result+=sauce->getMoreGarlic();
    result+=sauce->getMoreGinger();
    result+="\n";
    result+=veg->getSlicedCucumber();
    result+="and ";
    result+=veg->getShreddedCoriander();

    return result;
};
```

```
int main() {
    Rice *r = new Rice;
    Chicken *c = new Chicken;
    Sauce *s = new Sauce;
    Vegetable *v = new Vegetable;

    ChickenRiceFacade *facade = new ChickenRiceFacade(r,c,s,v);

    client(facade);

    delete facade;

    return 0;
}
```

```
cooking rice
boiled chicken
making sauce

order 1
Khao Man Gai Nua Nong + Nam Soup
Nam Jim Khao Man Gai + chili + garlic + ginger
sliced cucumber and shredded coriander

order 2
Khao Gai Sapok + Nam Soup
Nam Jim Khao Man Gai + chili + garlic + ginger
sliced cucumber and shredded coriander

order 3
Khao Man Gai + Nam Soup
Nam Jim Khao Man Gai + chili + garlic + ginger
sliced cucumber and shredded coriander
```

ควรใช้เมื่อไหร่

- เมื่อต้องการสร้างอินเตอร์เฟสเพื่อจัดการกับ subsystem ที่มีความซับซ้อน
- เมื่อต้องการจัดโครงสร้างของ subsystem ให้เป็นเลเยอร์
 - สร้าง facade ในการกำหนด entry point ให้กับแต่ละ level ของ subsystem
 - เป็นการลดการติดต่อกันระหว่าง subsystem โดยเปลี่ยนมาติดต่อกันผ่าน facade

ข้อดี ข้อเสีย

- ข้อดี
 - สามารถแยกโค้ดของเราออกจาก subsystem ที่ซับซ้อนได้
- ข้อเสีย
 - อาจกลายเป็น god object ที่ไปเกี่ยวข้องกับทุกๆ คลาสในแอพ

การบ้าน

- ให้คิดเมนูอาหารที่มีความซับซ้อนมา 1 รายการ

อ้างอิง

- <https://refactoring.guru/design-patterns/facade>

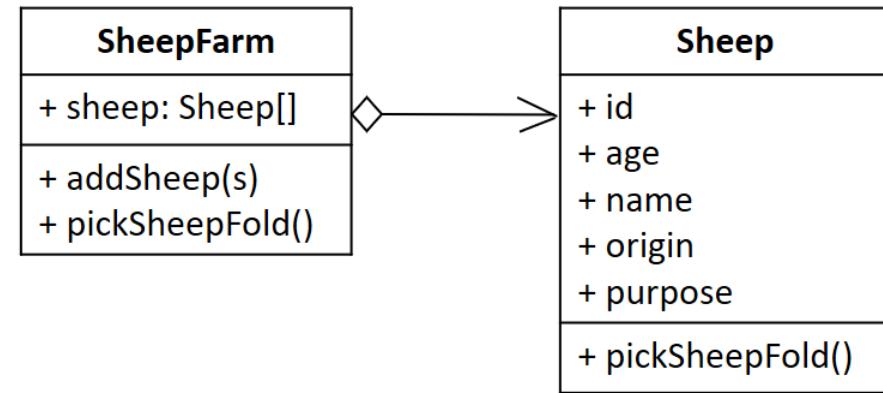
Catalog of Design Patterns

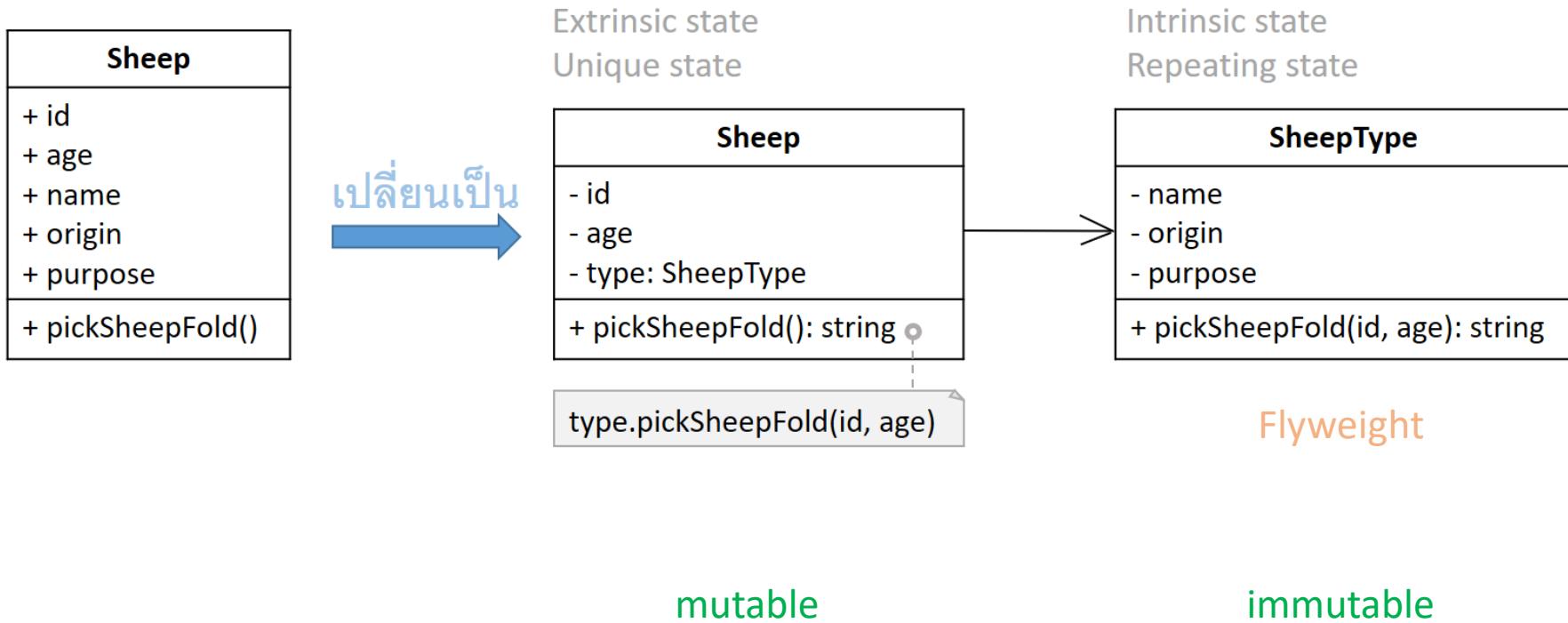
Structural Design Patterns

Flyweight

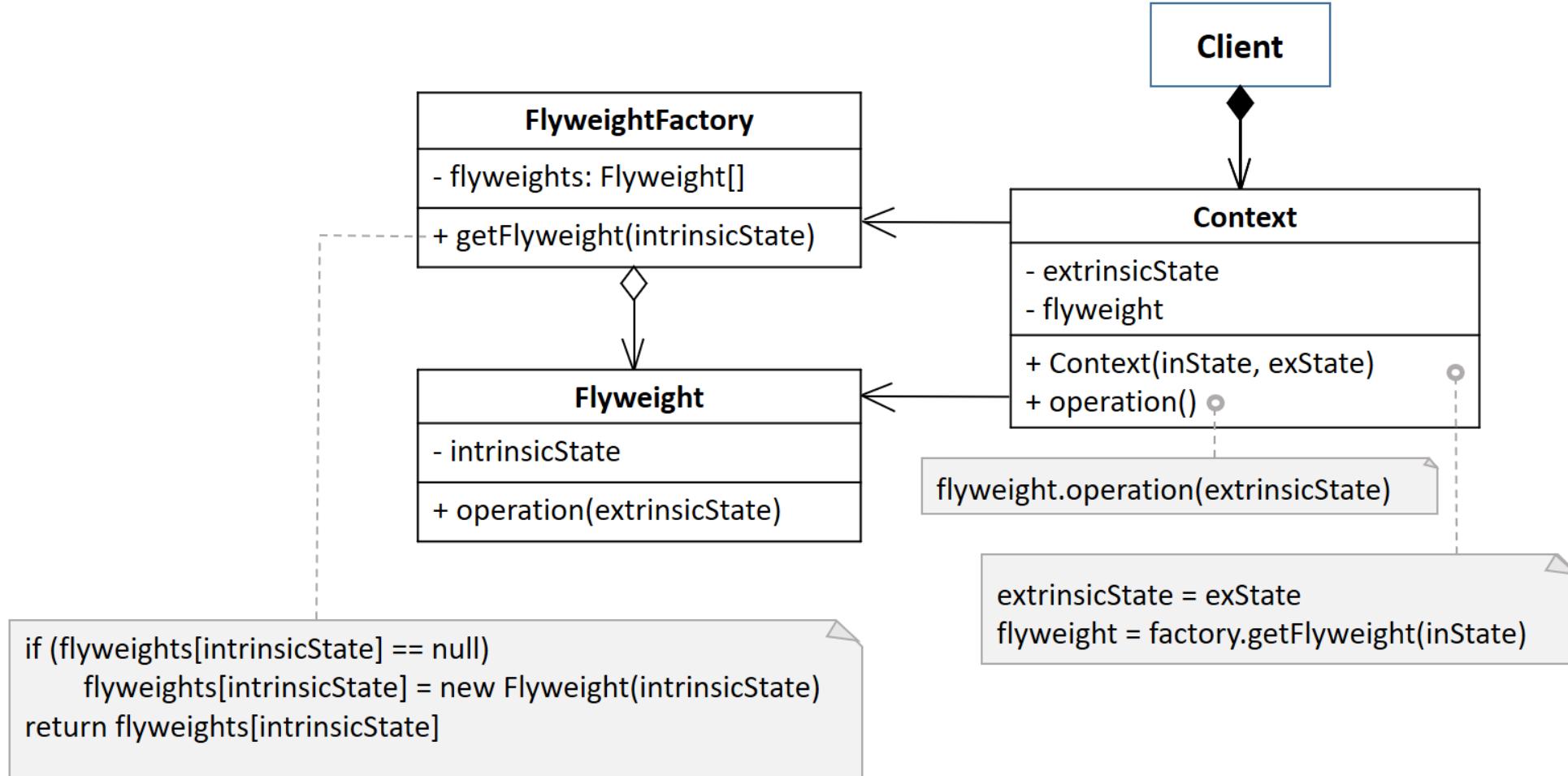
Flyweight หรือ Cache

- เป็นการแชร์ข้อมูลระหว่างอ็อบเจกต์ แทนที่จะเก็บข้อมูลทุกอย่างไว้ในอ็อบเจกต์แต่ละตัว ทั้งนี้เพื่อลดการใช้งานหน่วยความจำ





Structure



```
if (sheepType[(name, origin, purpose)] == null)
    sheepType[(name, origin, purpose)] = new SheepType(name, origin, purpose)
return sheepType[(name, origin, purpose)]
```

```
for s in sheep
    s.pickSheepFold()
```

SheepFactory

```
- sheepType: SheepType[]
+ getSheepType(name, origin, purpose)
```

SheepFarm

```
- sheep: Sheep[]
+ addSheep(id, age, name, origin, purpose)
+ pickSheepFold(): string
```

SheepType

```
- name: string
- origin: string
- purpose: string
+ SheepType(name, origin, purpose)
+ pickSheepFold(id, age)
```

Sheep

```
- id: string
- age: int
- type: SheepType
+ Sheep(id, age, type)
+ pickSheepFold(): string
```

```
result = id + name + origin
If (age<=3) return result += "for kids"
If (purpose == "wool") return += "for wool"
If (purpose == "milk") return += "for milk"
else return result += "for meat"
```

```
type.pickSheepFold(id, age)
```

```
class SheepType {
private:
    string name;
    string origin;
    string purpose;
public:
    SheepType(string n, string o, string p) {
        name=n; purpose=p; origin=o;
    }
    ~SheepType() {
        cout<<"delete type "<<name<<endl;
    }
    string getName() {
        return name;
    }
    string getPurpose() {
        return purpose;
    }
    string getOrigin() {
        return origin;
    }
    string pickSheepFold(string id, int age) {
        string result=[ " + id + " ] " + name + " " + origin;
        if (age<=3)
            return result + " => sheepfold for Kids \n";
        if (purpose == "wool")
            return result + " => sheepfold for Wool \n";
        else if (purpose == "milk")
            return result + " => sheepfold for Milk \n";
        else
            return result + " => sheepfold for Meat \n";
    }
};
```

```
class Sheep {
    string id;
    int age;
    SheepType *type;
public:
    Sheep(string i, int a, SheepType *s) {
        id = i;
        age = a;
        type = s;
    }
    ~Sheep() {
        cout<<"delete sheep "<<id<<endl;
    }
    string pickSheepFold() {
        return type->pickSheepFold(id, age);
    }
    string getID() {
        return id;
    }
};
```

```
class SheepFactory {
    list <SheepType *> sheepType;
public:
    SheepFactory() {
        sheepType.push_back(new SheepType("wiltipoll", "Australia", "meat"));
        sheepType.push_back(new SheepType("Targhee", "USA", "wool"));
        sheepType.push_back(new SheepType("Sardinian", "Italy", "milk"));
        sheepType.push_back(new SheepType("Sakiz", "Turkey", "wool"));
    }
    ~SheepFactory() {
        for (list<SheepType*>::iterator it = sheepType.begin(); it != sheepType.end(); it++)
            delete *it;
    }
    SheepType *getSheepType(string n, string o, string p) {
        SheepType *type=0;
        for (list<SheepType*>::iterator it = sheepType.begin(); it != sheepType.end(); it++) {
            if (n == (*it)->getName() && p == (*it)->getPurpose() && o == (*it)->getOrigin()) {
                type = *it;
                break;
            }
        }
        if (type == 0) {
            type = new SheepType(n, o, p);
            sheepType.push_back(type);
        }
        return type;
    }
};
```

```
class SheepFarm {
    list<Sheep*> sheep;
public:
    void addSheep(string i, int a, string n, string o, string p, SheepFactory *f) {
        SheepType *type;
        type = f->getSheepType(n, o, p);
        Sheep *s = new Sheep(i, a, type);
        sheep.push_back(s);
    }
    ~SheepFarm() {
        for (list<Sheep*>::iterator it = sheep.begin(); it != sheep.end(); it++)
            delete (*it);
    }
    string pickSheepFold() {
        string result;
        for (list<Sheep*>::iterator it = sheep.begin(); it != sheep.end(); it++) {
            result+=(*it)->pickSheepFold();
        }
        return result;
    }
};
```

```
//for (Sheep* s : sheep) {
//    result+=s->pickSheepFold();
//}
```

```

int main() {
    SheepFactory *factory = new SheepFactory();
    SheepFarm *myFarm = new SheepFarm;
    myFarm->addSheep("no.1", 5, "Sakiz", "Turkey", "wool", factory);
    myFarm->addSheep("no.2", 2, "wiltipoll", "Australia", "meat", factory);
    myFarm->addSheep("no.3", 6, "Targhee", "USA", "wool", factory);
    myFarm->addSheep("no.4", 3, "Sardinian", "Italy", "milk", factory);
    myFarm->addSheep("no.5", 12, "wiltipoll", "Australia", "meat", factory);
    myFarm->addSheep("no.6", 11, "Sakiz", "Turkey", "wool", factory);
    myFarm->addSheep("no.7", 7, "Roslag", "Sweden", "meat", factory);
    myFarm->addSheep("no.8", 8, "Sardinian", "Italy", "milk", factory);
    myFarm->addSheep("no.9", 7, "Roslag", "Sweden", "wool", factory);
    myFarm->addSheep("no.10", 1, "Roslag", "Sweden", "wool", factory);
    myFarm->addSheep("no.11", 6, "Roslag", "Sweden", "meat", factory);
    myFarm->addSheep("no.12", 5, "Pinzirita", "Sicily", "milk", factory);

    cout<<myFarm->pickSheepFold()<<endl;

    delete myFarm;
    delete factory;

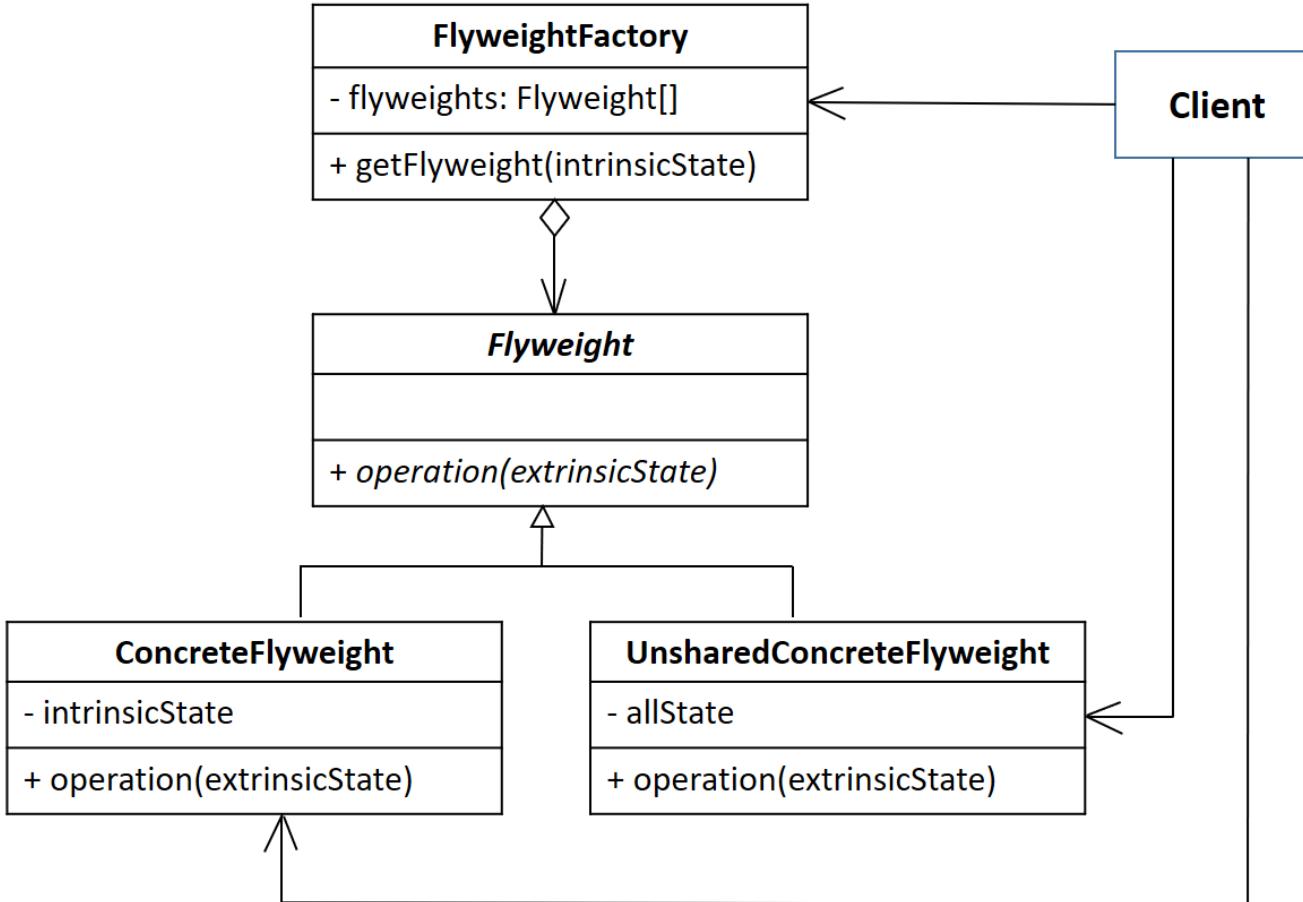
    return 0;
}

```

[no. 1]	Sakiz Turkey => sheepfold for Wool
[no. 2]	wiltipoll Australia => sheepfold for Kids
[no. 3]	Targhee USA => sheepfold for Wool
[no. 4]	Sardinian Italy => sheepfold for Kids
[no. 5]	wiltipoll Australia => sheepfold for Meat
[no. 6]	Sakiz Turkey => sheepfold for Wool
[no. 7]	Roslag Sweden => sheepfold for Meat
[no. 8]	Sardinian Italy => sheepfold for Milk
[no. 9]	Roslag Sweden => sheepfold for Wool
[no. 10]	Roslag Sweden => sheepfold for Kids
[no. 11]	Roslag Sweden => sheepfold for Meat
[no. 12]	Pinzirita Sicily => sheepfold for Milk

delete sheep no. 1
delete sheep no. 2
delete sheep no. 3
delete sheep no. 4
delete sheep no. 5
delete sheep no. 6
delete sheep no. 7
delete sheep no. 8
delete sheep no. 9
delete sheep no. 10
delete sheep no. 11
delete sheep no. 12
delete type wiltipoll
delete type Targhee
delete type Sardinian
delete type Sakiz
delete type Roslag
delete type Roslag
delete type Pinzirita

Structure



ควรใช้เมื่อไหร่

- เมื่อโปรแกรมของเราต้องเกี่ยวข้องกับอีกบอร์ดจำนวนมากและมี RAM จำกัด

ข้อดี ข้อเสีย

- ข้อดี
 - ประยุกต์ RAM
- ข้อเสีย
 - ได้ดีมีความซับซ้อน

การบ้าน

- สุนัข มีข้อมูล
 - รหัส
 - อายุ
 - ชื่อเล่น
 - สุนัขพันธุ์เล็ก หรือ ใหญ่
 - ชื่อสายพันธุ์
 - นิสัยเฉพาะสายพันธุ์
- จงแสดงข้อมูลสุนัขแต่ละตัว

อ้างอิง

- <https://refactoring.guru/design-patterns/flyweight>

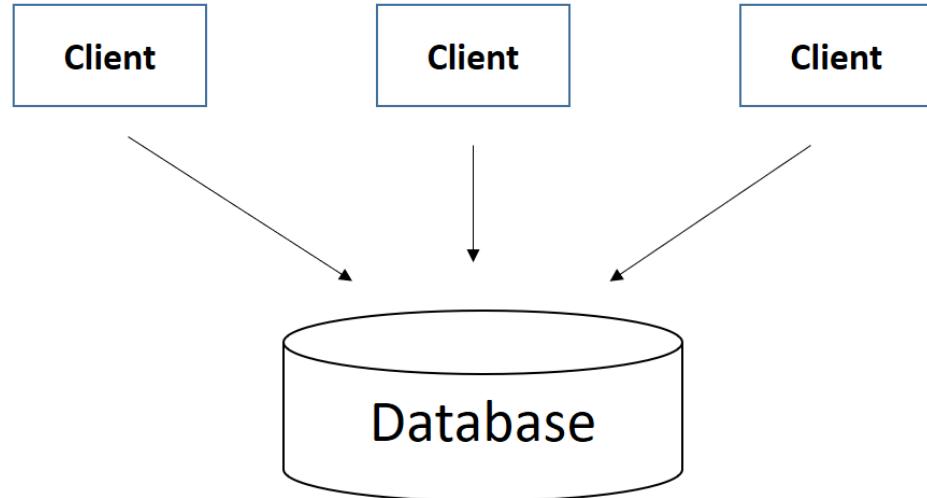
Catalog of Design Patterns

Structural Design Patterns

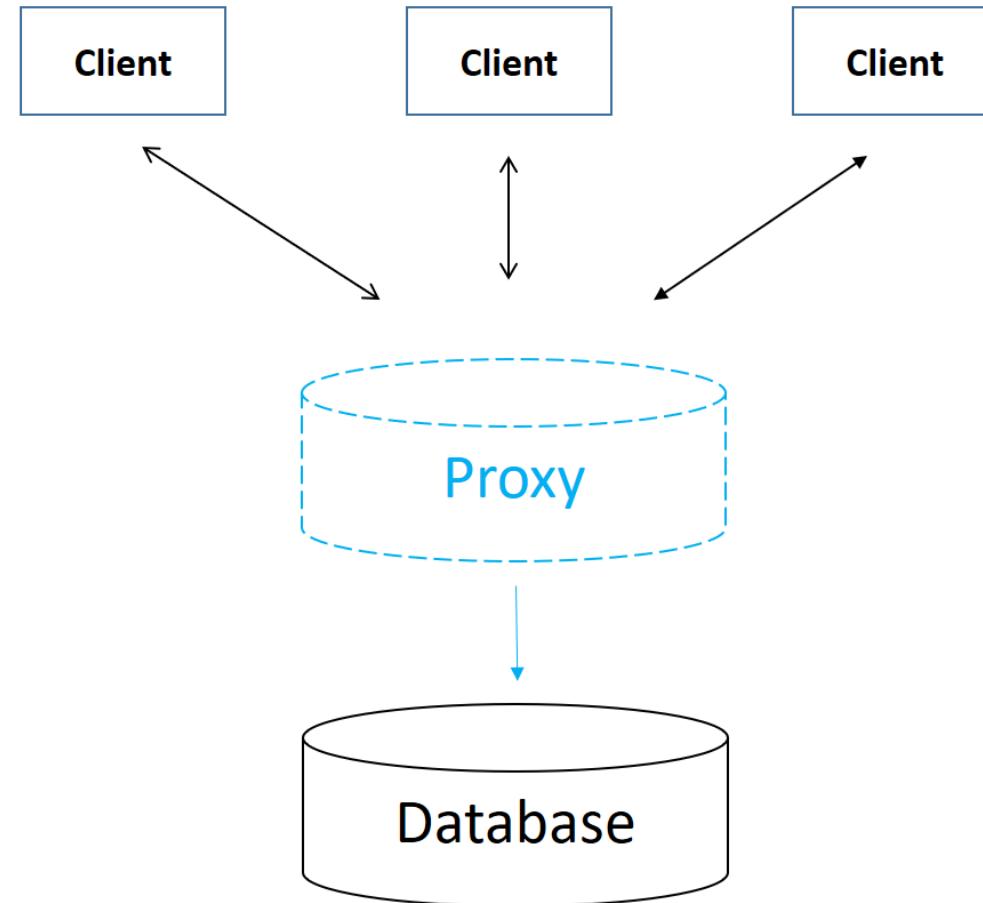
Proxy

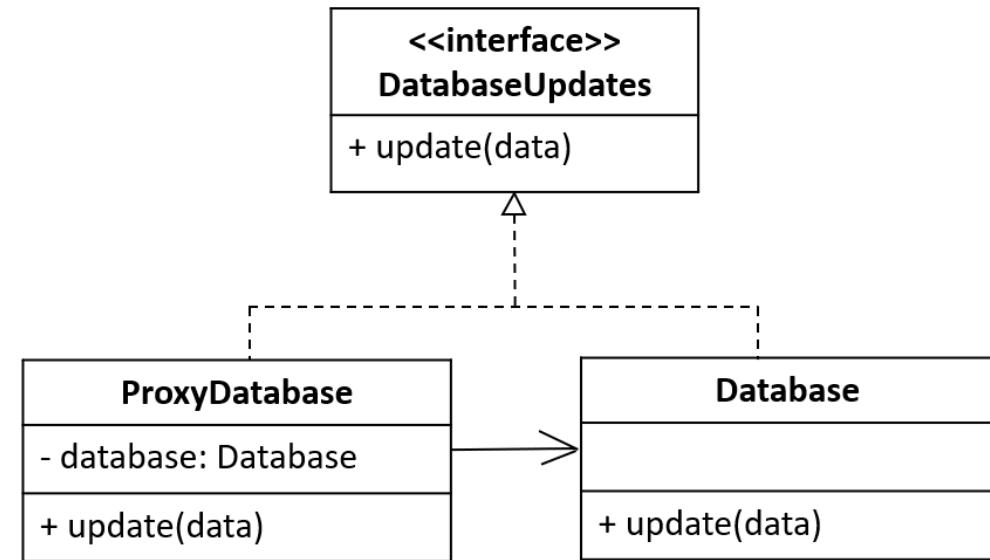
Proxy

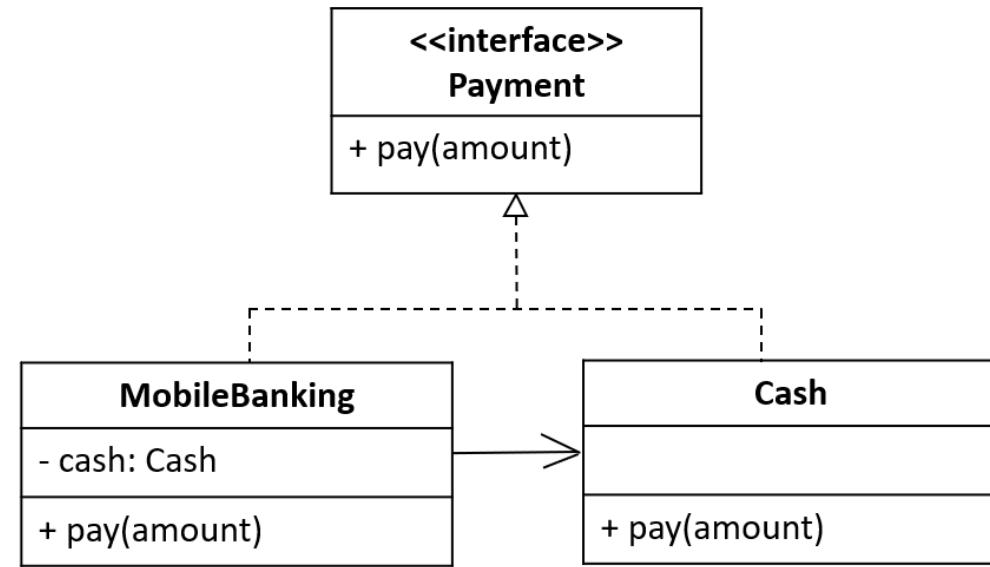
- เป็นการสร้างตัวแทนของอ็อกบเจกต์ขึ้นมา ทำหน้าที่ควบคุมการเข้าถึงอ็อกบเจกต์นั้น ทั้งก่อนและหลังการเรียกใช้อ็อกบเจกต์



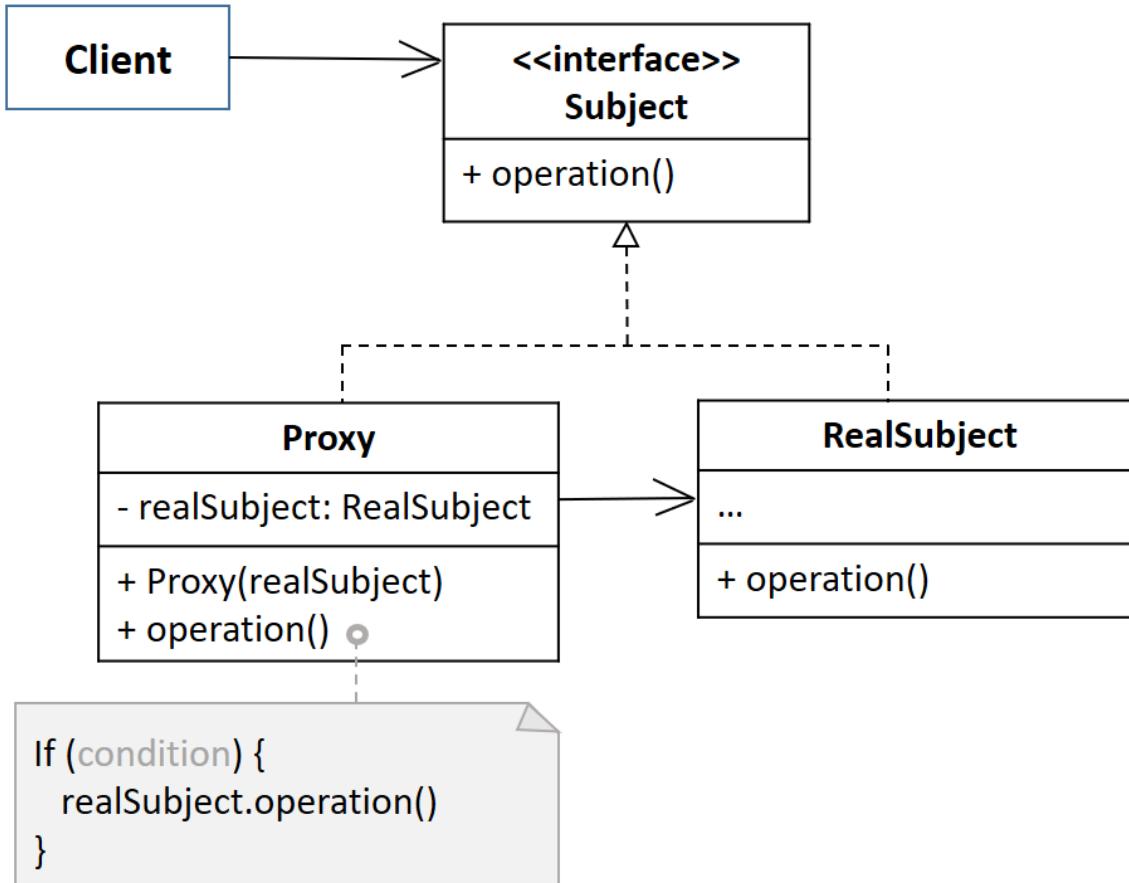
- Lazy initialization
- Code duplication

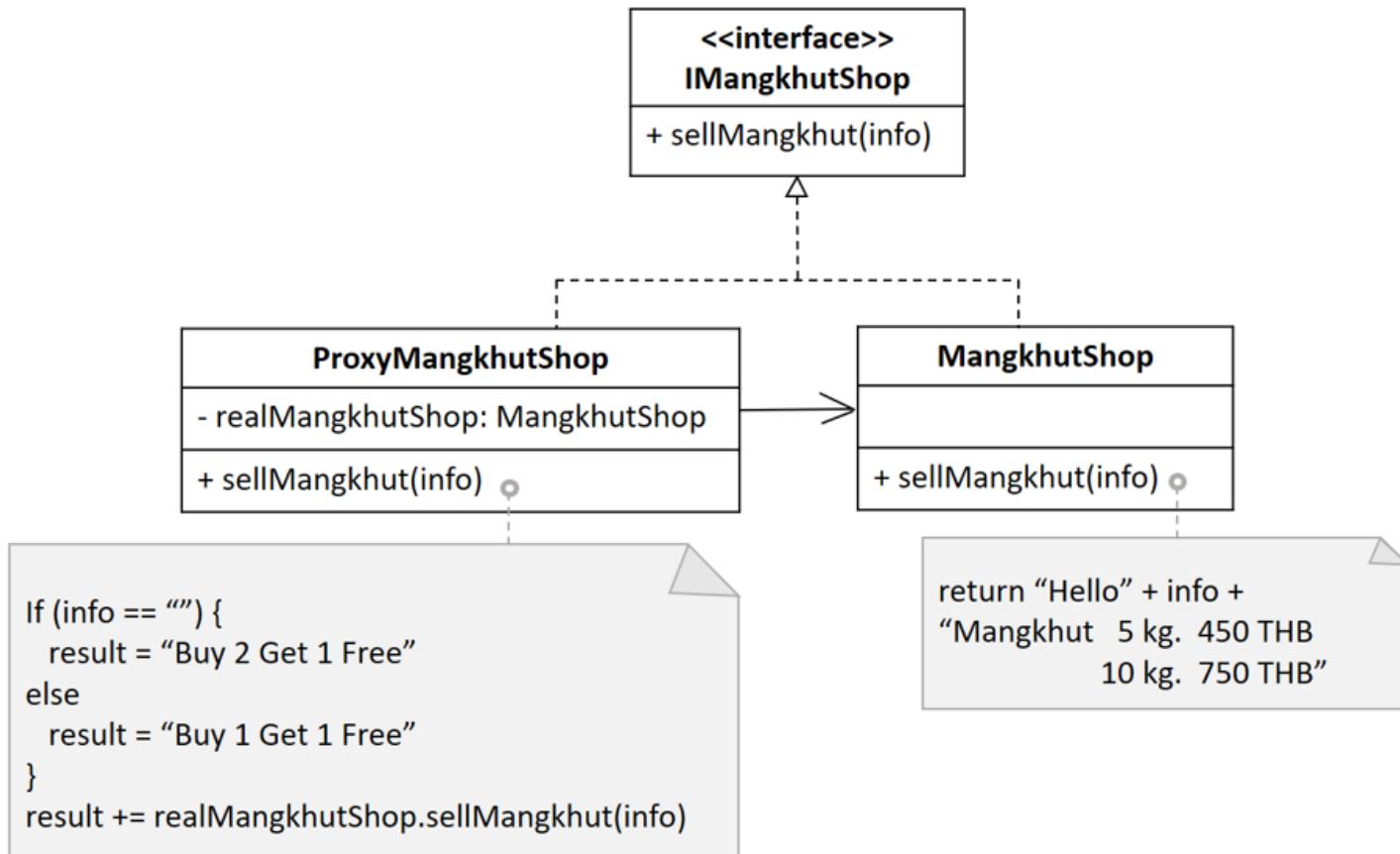






Structure





```
#include <iostream>
using namespace std;

class IMangkhutShop {
public:
    virtual string sellMangkhut(string info) = 0;
};

class MangkhutShop: public IMangkhutShop {
public:
    string sellMangkhut(string info) {
        return "Hello " + info + "\nMangkhut 5 kg. 450 THB \n" + "10 kg. 750 THB";
    }
};

class ProxyMangkhutShop: public IMangkhutShop {
    MangkhutShop *realMangkhutShop;

public:
    ProxyMangkhutShop(MangkhutShop *m) {
        realMangkhutShop = m;
    }

    string sellMangkhut(string info) {
        string result="Promotion: ";
        if (info == "")
            result += "Buy 2 Get 1 Free\n";
        else
            result += "Buy 1 Get 1 Free\n";
        result += realMangkhutShop->sellMangkhut(info);
        return result;
    }
};
```

```
void client(IMangkhutShop &shop, string info) {
    cout<<shop.sellMangkhut(info)<<endl;
}

int main() {
    cout<<"Real Mangkhut Shop:"<<endl;
    MangkhutShop *realShop = new MangkhutShop;
    client(*realShop, "Micky");
    cout<<endl;
    cout<<"Proxy Mangkhut Shop:"<<endl;
    ProxyMangkhutShop *proxy = new ProxyMangkhutShop(realShop);
    client(*proxy, "lilly");
    cout<<endl;
    client(*proxy, "");

    delete realShop;
    delete proxy;
    return 0;
}
```

Real Mangkhut Shop:
Hello Micky
Mangkhut 5 kg. 450 THB
10 kg. 750 THB

Proxy Mangkhut Shop:
Promotion: Buy 1 Get 1 Free
Hello lilly
Mangkhut 5 kg. 450 THB
10 kg. 750 THB

Promotion: Buy 2 Get 1 Free
Hello
Mangkhut 5 kg. 450 THB
10 kg. 750 THB

ควรใช้เมื่อไหร่

- Virtual proxy
 - Lazy initialization
 - เมื่อถึงเวลาใช้จริงค่อย **init**
- Protection proxy
 - Access control
 - Proxy จะส่ง **request** ก็ต่อเมื่อสอดคล้องกับเงื่อนไขบางอย่าง
- Remote proxy
 - เมื่อต้องการให้บริการอยู่บน **remote server**
 - Proxy จะส่ง **request** ผ่าน **network** โดยจัดการรายละเอียดต่างๆ บน **network** ให้

ควรใช้เมื่อไหร่

- Logging proxy
 - Logging request
 - เมื่อต้องการเก็บประวัติของการส่ง request ไปยังอีบเจกต์ที่ให้บริการ
- Caching proxy
 - Caching request results
 - เมื่อต้องการจะ cache ผลลัพธ์ที่ได้จากการ request และจัดการ life cycle ของ cache นี้
 - บาง request ก็ให้ผลลัพธ์แบบเดิมตลอด ซึ่ง proxy อาจใช้พารามิเตอร์ของ request เป็น cache key ก็ได้
- Smart reference
 - เพื่อที่จะสามารถเลิกอีบเจกต์ที่ให้บริการเมื่อไม่มีใครใช้ และ free system resources
 - ติดตามว่ามีใครแก้ไขอีบเจกต์ที่ให้บริการหรือไม่ อีบเจกต์ที่ไม่ถูกแก้ไข สามารถนำมาให้ client อื่นๆ ใช้ซ้ำได้

ข้อดี ข้อเสีย

- ข้อดี
 - สามารถควบคุมอ็อบเจกต์ที่ให้บริการได้โดยที่ client ไม่ต้องรู้อะไรมาก
 - สามารถจัดการ lifecycle ของอ็อบเจกต์ที่ให้บริการได้ในกรณีที่ client ไม่สนใจจะจัดการ
 - Proxy สามารถทำงานได้ถึงแม้ว่าอ็อบเจกต์ที่ให้บริการจะยังไม่พร้อม
 - Open/Closed Principle
 - สามารถสร้าง proxy ใหม่ได้โดยไม่ต้องมีการเปลี่ยนแปลงแก้ไขอ็อบเจกต์ที่ให้บริการหรือเปลี่ยนแปลง client
- ข้อเสีย
 - โดยมีความซับซ้อน
 - Response จากการให้บริการอาจมีความล่าช้า

การบ้าน

- ร้านข้าวมันไก่
- ทุกครั้งที่ลูกค้าสั่งข้าวมันไก่แบบออนไลน์ผ่าน proxy ให้แกรมน้ำส้ม 1 ขวด

อ้างอิง

- <https://refactoring.guru/design-patterns/proxy>

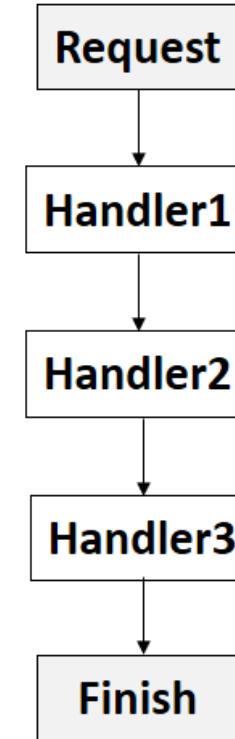
Catalog of Design Patterns

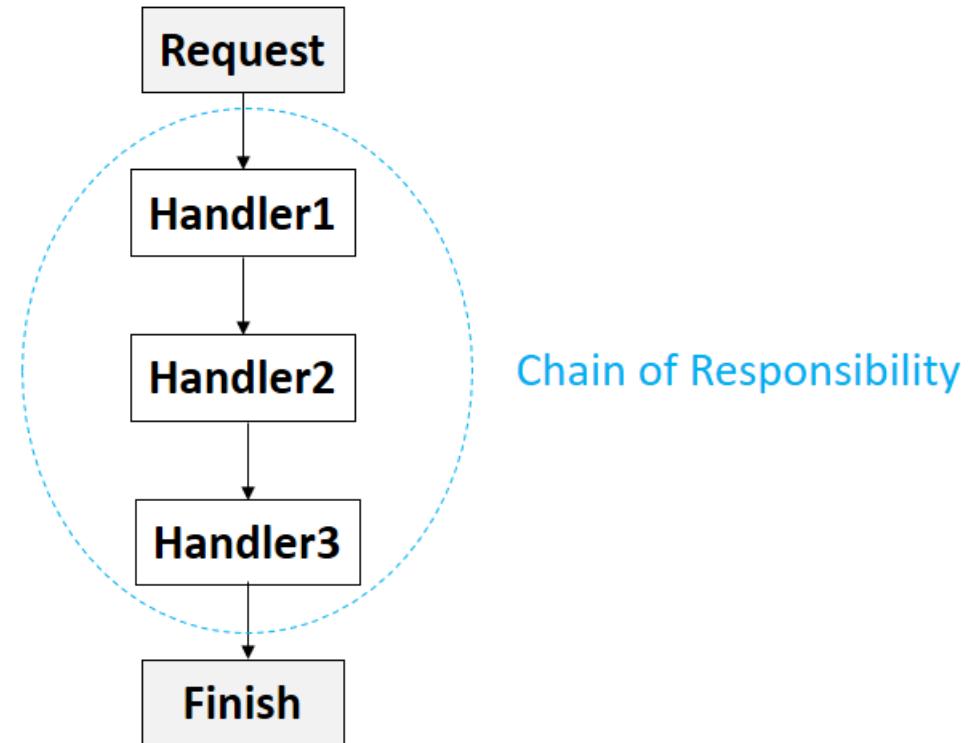
Behavioral Design Patterns

Chain of Responsibility

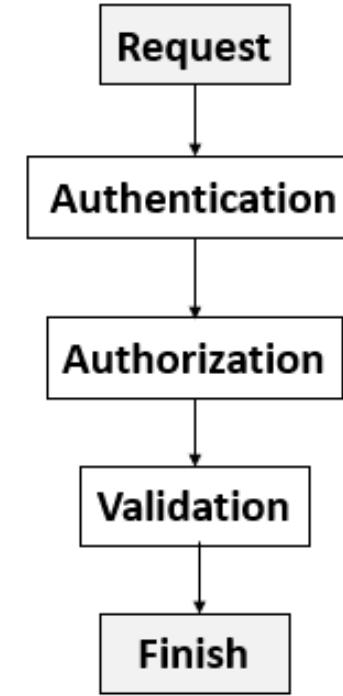
Chain of Responsibility (CoR) หรือ Chain of Command

- ส่ง request ไปยังห่วงโซ่ของ handler
- แต่ละ handler จะทำการตัดสินใจว่า
 - จะดำเนินการกับ request นั้นหรือไม่
 - จะส่ง request นั้นต่อไปยัง handler ที่อยู่ในห่วงโซ่ตัวถัดไปหรือไม่
- แต่ละ handler จะทำงานแค่อย่างเดียว

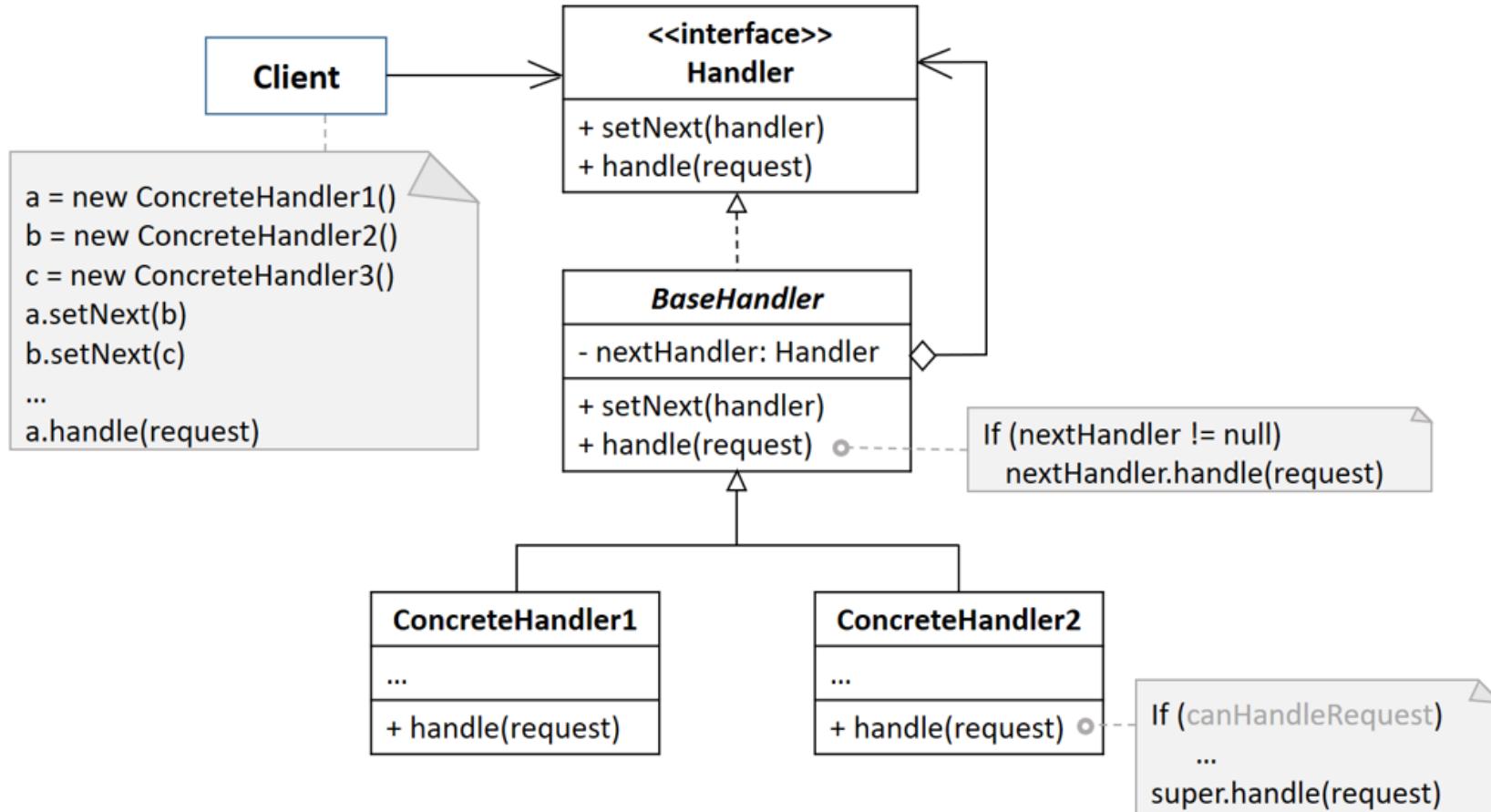


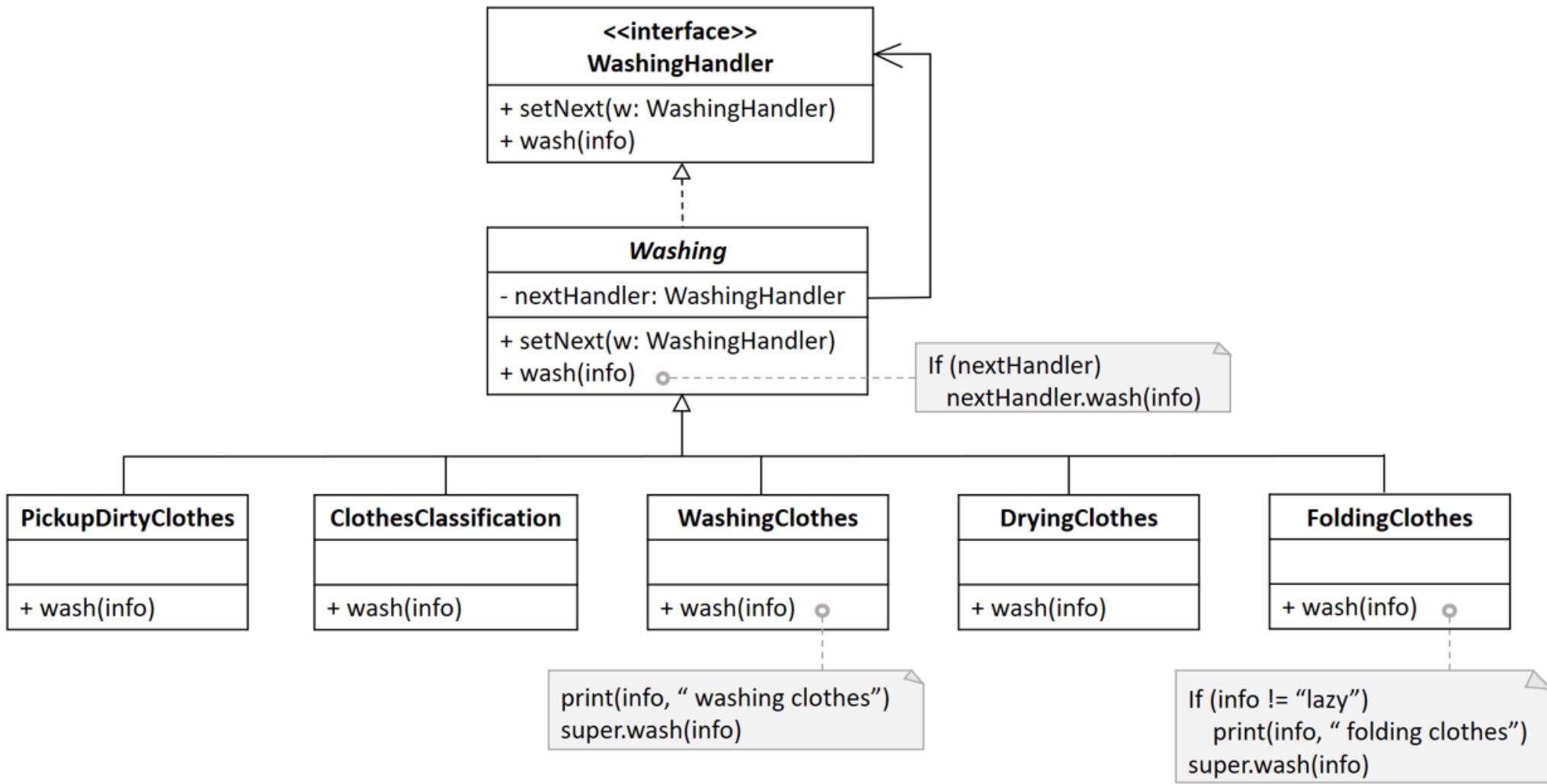


- การจัดการการเข้าถึงระบบ
 - Authentication
 - Authorization
 - Validation
- แทนที่จะใช้ `if` ในการตรวจสอบ ก็เปลี่ยนเป็นคลาสที่มีเมธอดเดียวในการตรวจสอบ
- แต่ละ `handler` มีแอทริบิวต์ที่เก็บ `reference` ที่ชี้ไปยัง `handler` ถัดไป



Structure





```
#include <iostream>
using namespace std;

class WashingHandler {
public:
    virtual WashingHandler *setNext(WashingHandler *handler) = 0;
    virtual void wash(string info) = 0;
};

class Washing: public WashingHandler {
    WashingHandler *nextHandler;

public:
    Washing() {
        nextHandler = 0;
    }
    WashingHandler *setNext(WashingHandler *handler) {
        nextHandler = handler;
        return nextHandler;
    }
    void wash(string info) {
        if (nextHandler) {
            nextHandler->wash(info);
        }
    }
};

class PickupDirtyClothes: public Washing {
public:
    void wash(string info) {
        if (info != "lazy")
            cout<<info<<"-> pickup dirty cloths\n";
        Washing::wash(info);
    }
};

class ClothesClassifier: public Washing {
public:
    void wash(string info) {
        if (info != "lazy")
            cout<<info<<"-> classify cloths\n";
        Washing::wash(info);
    }
};

class WashingClothes: public Washing {
public:
    void wash(string info) {
        cout<<info<<"-> washing cloths\n";
        Washing::wash(info);
    }
};
```

```

class DryingClothes: public Washing {
public:
    void wash(string info) {
        if (info != "do not dry")
            cout<<info<<"-> drying cloths\n";
        Washing::wash(info);
    }
};

class FoldingClothes: public Washing {
public:
    void wash(string info) {
        if (info != "lazy")
            cout<<info<<"-> folding cloths\n";
        Washing::wash(info);
    }
};

void client(WashingHandler &handler) {
    handler.wash("mom");
}

```

```

int main() {
    PickupDirtyClothes *h1 = new PickupDirtyClothes;
    ClothesClassifier *h2 = new ClothesClassifier;
    WashingClothes *h3 = new WashingClothes;
    DryingClothes *h4 = new DryingClothes;
    FoldingClothes *h5 = new FoldingClothes;

    h1->setNext(h2)->setNext(h3)->setNext(h4)->setNext(h5);
    client(*h1);

    cout<<endl;
    h3->wash("Micky");

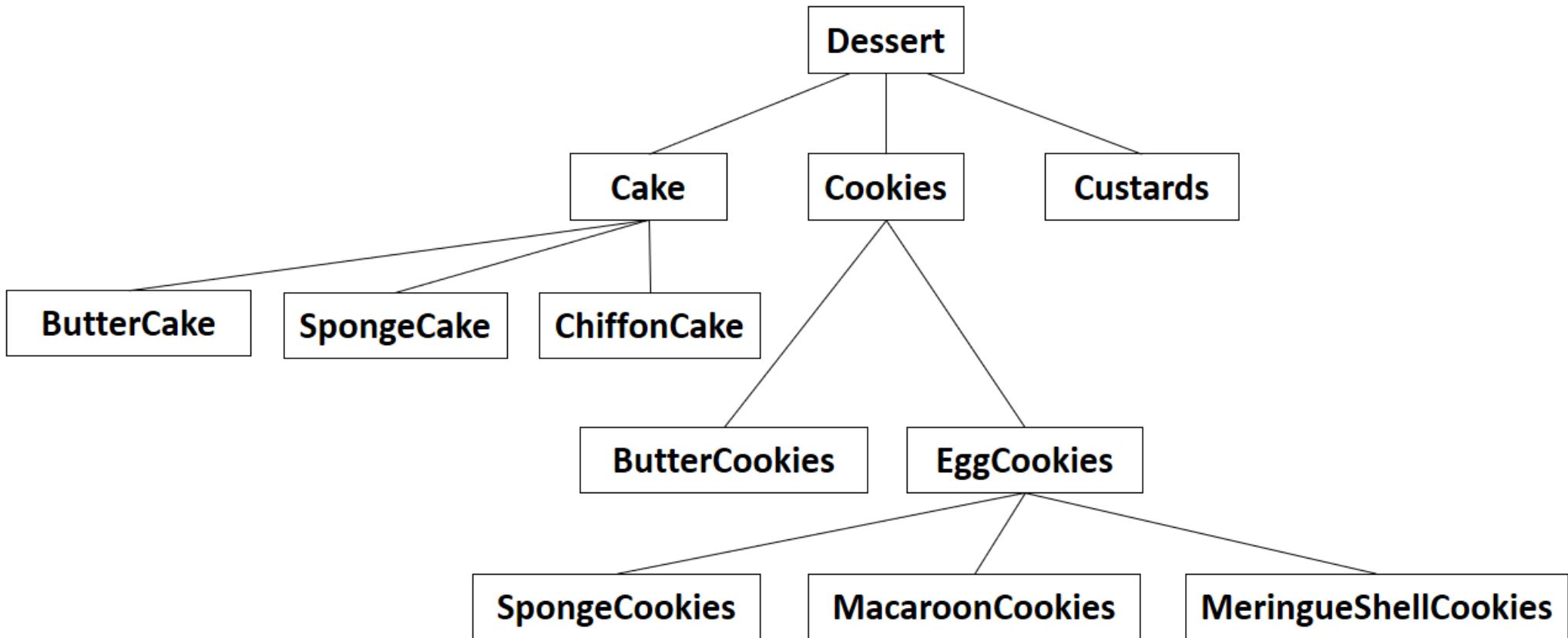
    cout<<endl;
    h1->wash("lazy");

    cout<<endl;
    h1->wash("do not dry");

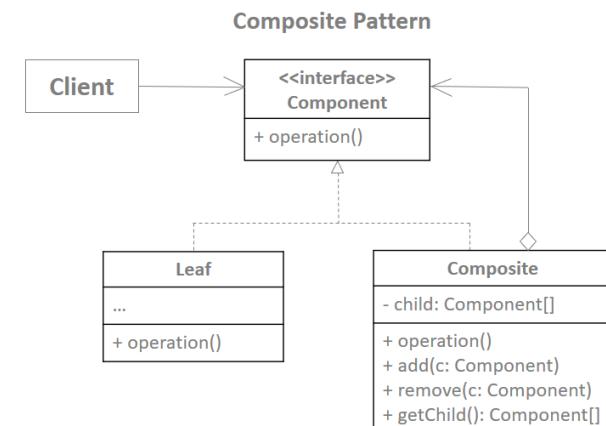
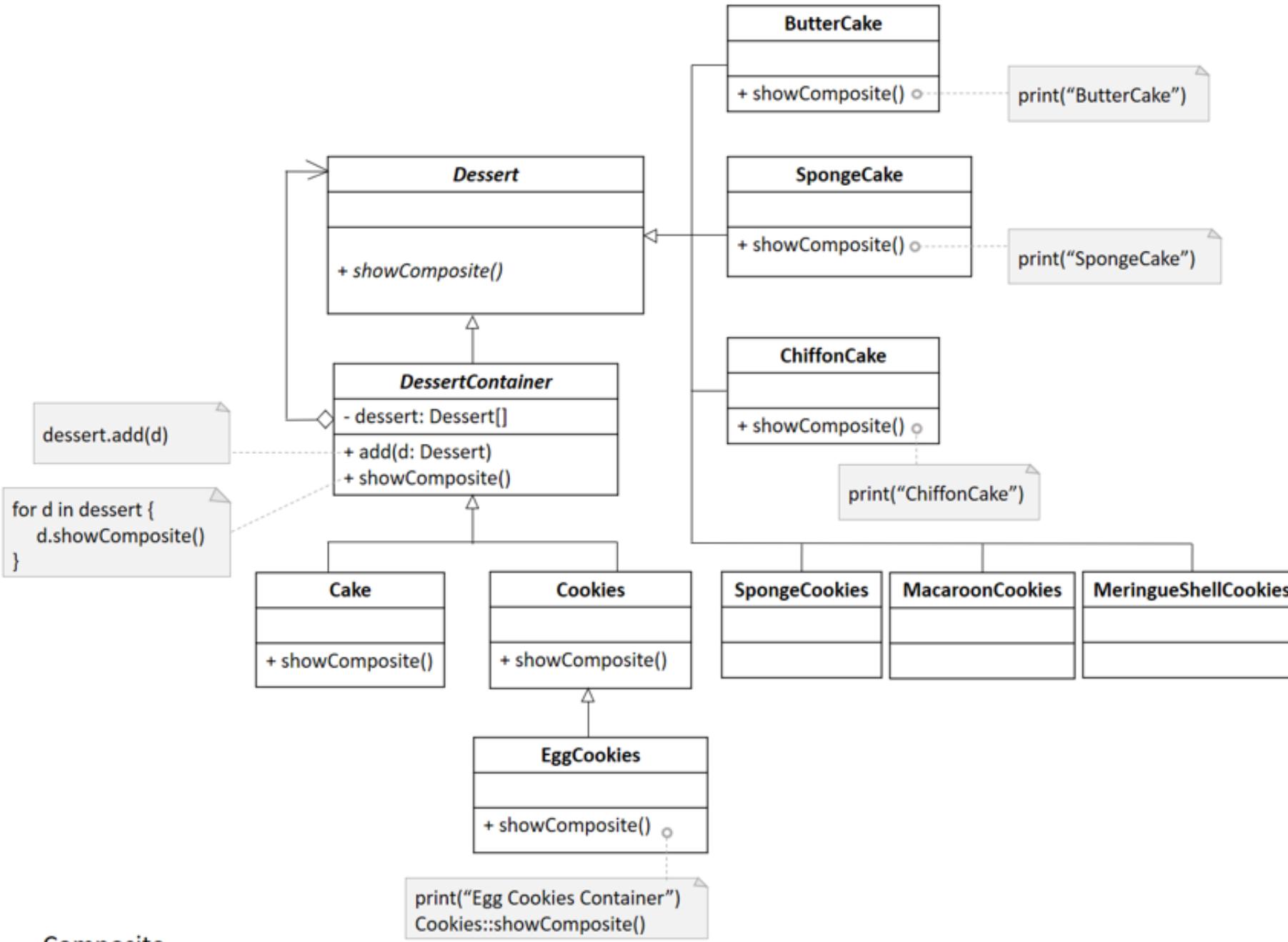
    delete h1;
    delete h2;
    delete h3;
    delete h4;
    delete h5;
    return 0;
}

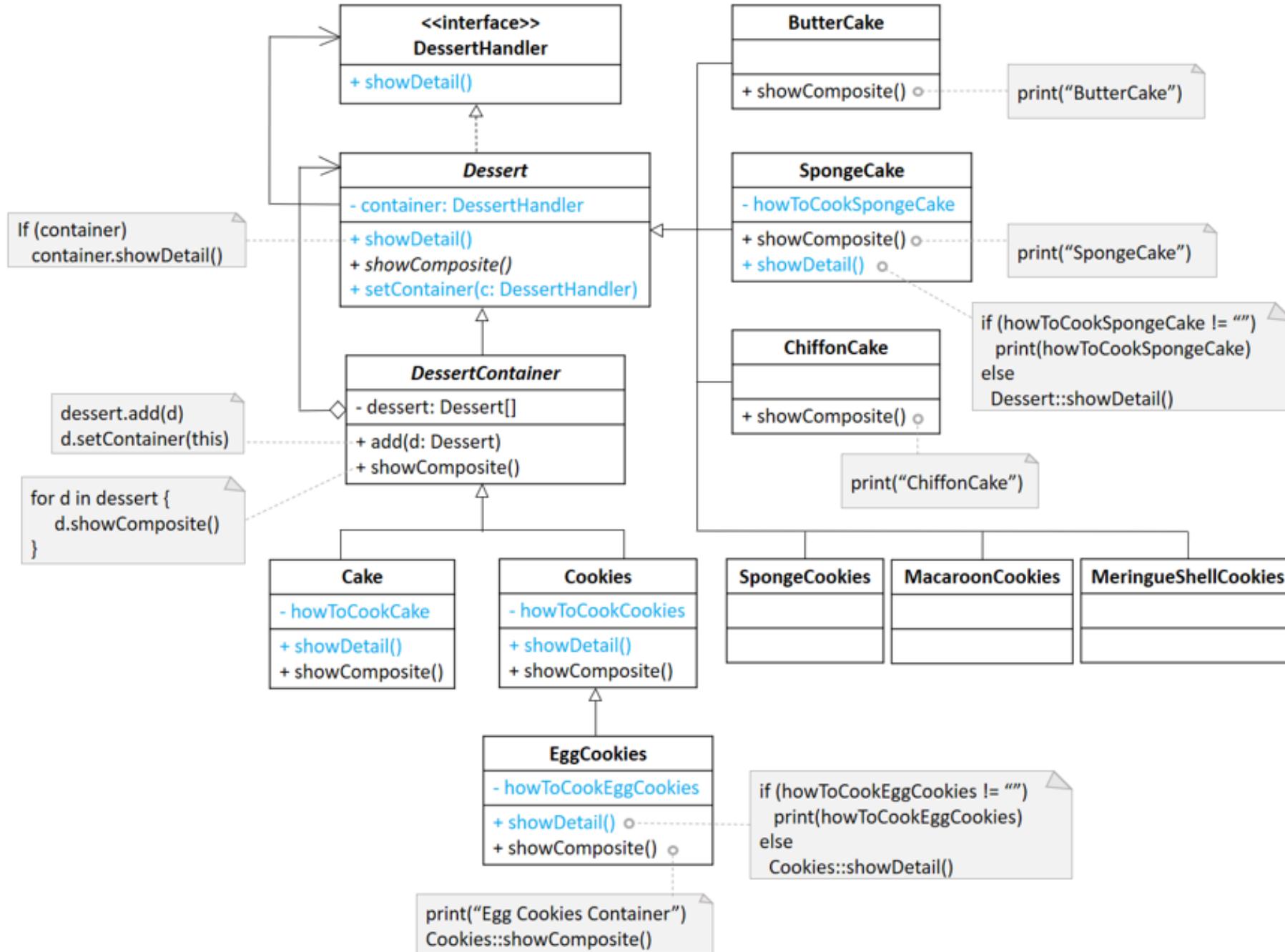
```

mom-> pickup dirty cloths
mom-> classify cloths
mom-> washing cloths
mom-> drying cloths
mom-> folding cloths
Micky-> washing cloths
Micky-> drying cloths
Micky-> folding cloths
lazy-> washing cloths
lazy-> drying cloths
do not dry-> pickup dirty cloths
do not dry-> classify cloths
do not dry-> washing cloths
do not dry-> folding cloths



- ให้แสดงคู่มือทำอาหารสำหรับแต่ละคลาส ถ้าคลาสไหนไม่มีก็ให้คลาสแม่แสดงแทน





```
#include<iostream>
#include<list>
using namespace std;

class DessertHandler {
public:
    virtual void showDetail() = 0;
};

class Dessert: public DessertHandler {
    DessertHandler *container; // next link for coR
public:
    Dessert() {
        container=0;
    }
    void setContainer(DessertHandler *c) {
        container = c;
    }
    virtual ~Dessert() {}
    void showDetail() { // for coR
        if (container)
            container->showDetail();
    }
    virtual void showComposite() = 0;
    virtual void add(Dessert *) {}
};
```

```
class ButterCake: public Dessert {
public:
    void showComposite() {
        cout<<"ButterCake"<<endl;
    }
};

class SpongeCake: public Dessert {
    string howToCookSpongeCake;
public:
    SpongeCake() {
        howToCookSpongeCake = "How to cook SPONGE CAKE";
    }
    void showComposite() {
        cout<<"SpongeCake"<<endl;
    }
    void showDetail() {
        if (howToCookSpongeCake != "")
            cout<<howToCookSpongeCake<<endl;
        else
            Dessert::showDetail();
    }
};

class ChiffonCake: public Dessert {
public:
    void showComposite() {
        cout<<"ChiffonCake"<<endl;
    }
};
```

```
class DessertContainer: public Dessert {
    list <Dessert*> dessert;
public:
    void add(Dessert *d) {
        dessert.push_back(d);
        d->setContainer(this);
    }
    void showComposite() {
        for ( list<Dessert*>::iterator it = dessert.begin(); it != dessert.end(); it++ ) {
            (*it)->showComposite();
        }
        cout<<endl;
    }
};
```

```
class Cake: public DessertContainer {
    string howToCookCake;
public:
    Cake() {
        howToCookCake = "How to cook CAKE.";
    }
    void showDetail() {
        if (howToCookCake != "")
            cout<<howToCookCake<<endl;
        else
            DessertContainer::showDetail();
    }
    void showComposite() {
        cout<<"\nCake container:"<<endl;
        DessertContainer::showComposite();
    }
};
```

```
class Cookies: public DessertContainer {
    string howToCookCookies;
public:
    Cookies() {
        howToCookCookies = "";
    }
    void showDetail() {
        if (howToCookCookies != "")
            cout<<howToCookCookies<<endl;
        else
            DessertContainer::showDetail();
    }
    void showComposite() {
        cout<<"\nCookies container:"<<endl;
        DessertContainer::showComposite();
    }
};
```

```
class EggCookies: public Cookies {
    string howToCookEggCookies;
public:
    EggCookies() {
        howToCookEggCookies = "How to cook EGG COOKIES";
    }
    void showDetail() {
        if (howToCookEggCookies != "")
            cout<<howToCookEggCookies<<endl;
        else
            Cookies::showDetail();
    }
    void showComposite() {
        cout<<"\nEgg Cookies container:"<<endl;
        Cookies::showComposite();
    }
};
```

```
class ButterCookies: public Dessert {
public:
    void showComposite() {
        cout<<"ButterCookies"<<endl;
    }
};

class SpongeCookies: public Dessert {
public:
    void showComposite() {
        cout<<"SpongeCookies"<<endl;
    }
};

class MacaroonCookies: public Dessert {
string howToCookMacaroonCookies;
public:
    MacaroonCookies() {
        howToCookMacaroonCookies = "how to cook macaroon cookies";
    }
    void showComposite() {
        cout<<"MacaroonCookies"<<endl;
    }
    void showDetail() {
        if (howToCookMacaroonCookies != "") {
            cout<<howToCookMacaroonCookies<<endl;
        } else {
            Dessert::showDetail();
        }
    }
};

class MeringueShellCookies: public Dessert {
public:
    void showComposite() {
        cout<<"MeringueShellCookies"<<endl;
    }
};
```

```
int main() {
    Dessert *dessertContainer = new DessertContainer;

    Dessert *b1 = new ButterCake;
    Dessert *s1 = new SpongeCake;
    Dessert *c1 = new ChiffonCake;

    Dessert *bc = new ButterCookies;
    Dessert *sc = new SpongeCookies;
    Dessert *mc = new MacaroonCookies;
    Dessert *msc = new MeringueShellCookies;

    DessertContainer *cake = new Cake;
    DessertContainer *cookies = new Cookies;
    DessertContainer *eggCookies = new EggCookies;

    dessertContainer->add(cake);
    dessertContainer->add(cookies);
    cookies->add(bc);
    cookies->add(eggCookies);

    cake->add(b1);
    cake->add(s1);
    cake->add(c1);

    eggCookies->add(sc);
    eggCookies->add(mc);
    eggCookies->add(msc);

    dessertContainer->showComposite();
//=====
}
```

```
cout<<"-- cake containter --"<<endl;
cake->showDetail();
cout<<endl;
cout<<"-- egg cookies containter --"<<endl;
eggCookies->showDetail();
cout<<endl;
cout<<"-- butter cake --"<<endl;
b1->showDetail();
cout<<endl;
cout<<"-- macaroon cookies --"<<endl;
mc->showDetail();
cout<<endl;
cout<<"-- sponge cake --"<<endl;
s1->showDetail();
cout<<endl;
cout<<"-- meringue shell cookies --"<<endl;
msc->showDetail();
cout<<endl;

delete b1;
delete s1;
delete c1;
delete bc;
delete sc;
delete mc;
delete msc;
delete cake;
delete cookies;
delete eggCookies;
delete dessertContainer;
```

Cake container:
ButterCake
SpongeCake
ChiffonCake

Cookies container:
ButterCookies

Egg Cookies container:

Cookies container:
SpongeCookies
MacaroonCookies
MeringueShellCookies

-- cake containter --
How to cook CAKE.

-- egg cookies containter --
How to cook EGG COOKIES

-- butter cake --
How to cook CAKE.

-- macaroon cookies --
how to cook macaroon cookies

-- sponge cake --
How to cook SPONGE CAKE

-- meringue shell cookies --
How to cook EGG COOKIES

ควรใช้เมื่อไหร่

- เมื่อต้องดำเนินการกับ **request** ในหลากหลายรูปแบบโดยที่ยังไม่รู้ว่า **request** นั้นเป็น **request** ชนิดไหนและยังไม่รู้ลำดับการทำงานมาก่อน
 - handler** แต่ละตัวจะทำการตัดสินใจว่าจะดำเนินการกับ **request** นั้นหรือไม่
- เมื่อต้องการใช้งาน **handler** ตามลำดับที่กำหนดไว้
- เมื่อ **handler** และลำดับการทำงาน มีการเปลี่ยนแปลงในช่วงเวลา **runtime**
 - ในคลาส **handler** ถ้าเรามี **setter** ในการกำหนดค่า **reference** จะทำให้สามารถเพิ่ม ลบ หรือเปลี่ยนแปลงลำดับของ **handler** ได้อย่างไดนามิกซ์

ข้อดี ข้อเสีย

- ข้อดี
 - สามารถควบคุมลำดับของการจัดการ request ได้
 - Single Responsibility Principle
 - Open/Closed Principle
 - สามารถสร้าง handler ใหม่ได้โดยไม่กระทบกับโค้ดเดิม
- ข้อเสีย
 - อาจมีบาง request ที่ไม่ถูกจัดการเลย

การบ้าน

- เล่นรถไฟ Hague ตีลังกา
 - ส่วนสูงต้องถึงเกณฑ์
 - ไม่เป็นโรคหัวใจ
 - มีสิ่งของสัมภาระ ต้องฝากไว้ก่อน
 - เล่นได้
 - ถ้าฝากของไว้ ต้องรับคืนด้วย

อ้างอิง

- <https://refactoring.guru/design-patterns/chain-of-responsibility>

Catalog of Design Patterns

Behavioral Design Patterns

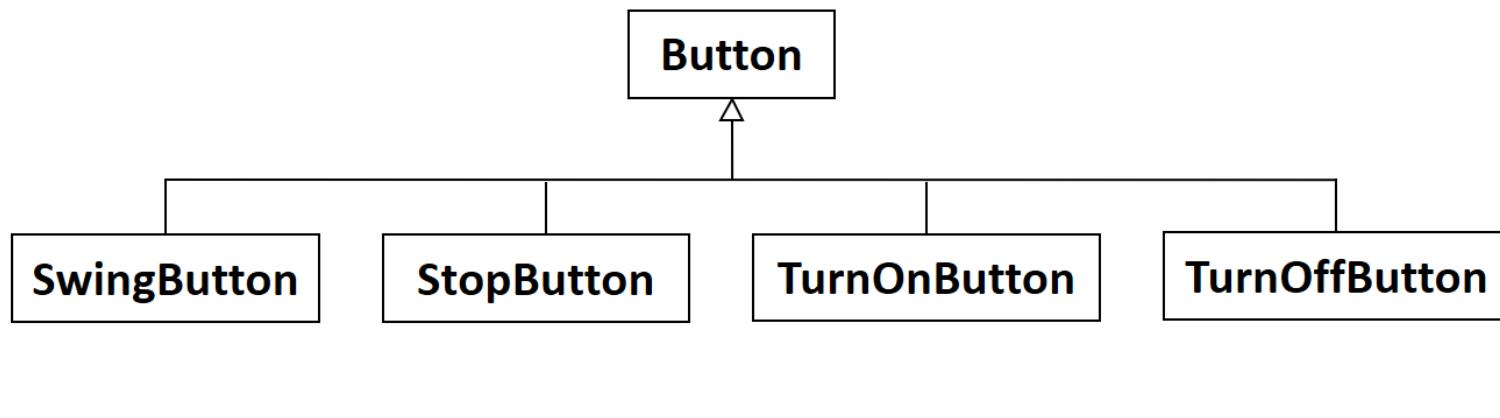
Command

Command หรือ Action หรือ Transaction

- เปลี่ยน **request** ไปเป็น **อ็อบเจกต์**
 - โดยที่ **อ็อบเจกต์** จะประกอบด้วยข้อมูลทุกอย่างเกี่ยวกับ **request** นั้น

- พัดลม

- ปุ่มเปิด
- ปุ่มปิด
- ปุ่มเริ่มสาย
- ปุ่มหยุดสาย



- สามารถสั่งงานได้หลายวิธี
 - เช่น การกดปุ่ม , ใช้มือท , สั่งงานผ่าน mobile app

SwingButton

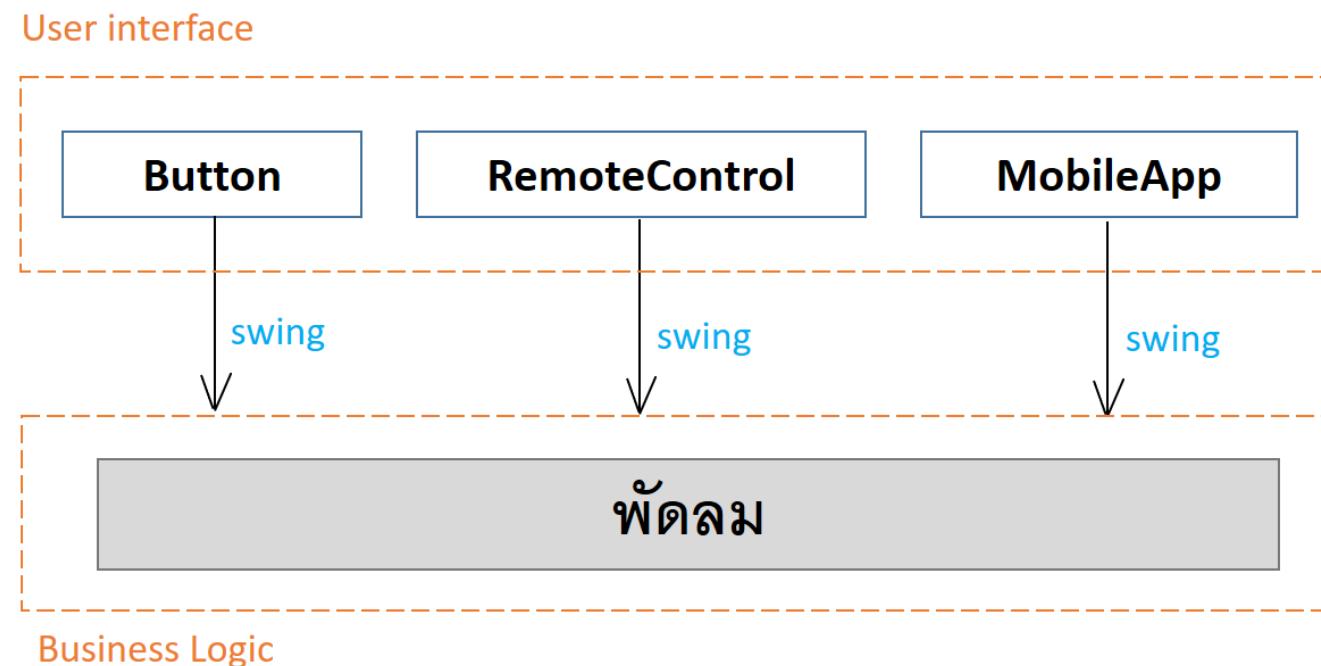
SwingRemoteControl

SwingMobileApp

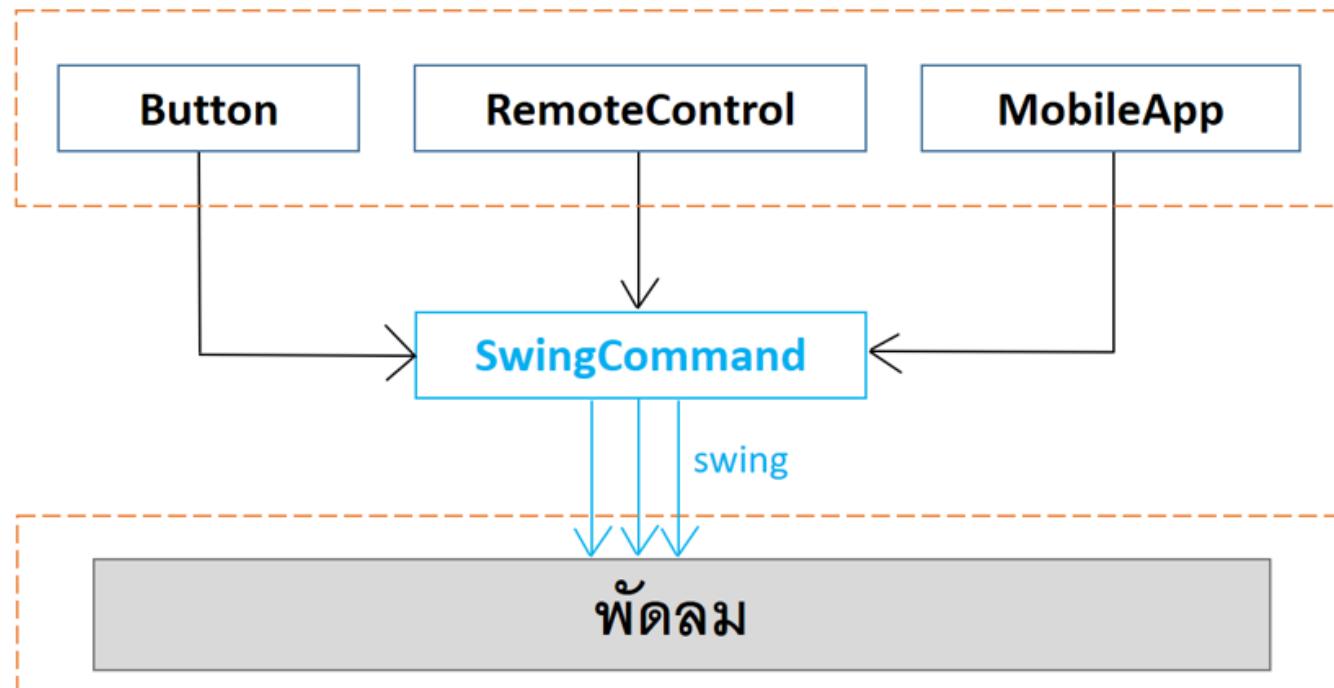
- ทุกวิธีใช้โค้ดเดียวกันในการสั่งงาน
- เกิดการ **duplicate** โค้ด

- ใช้หลัก principle of separation of concerns

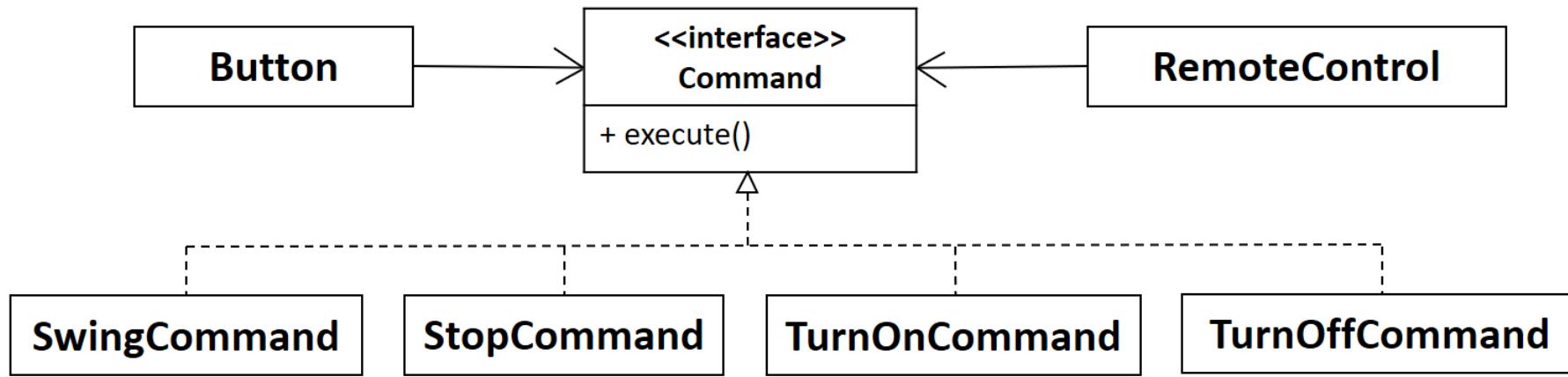
- แบ่งออกเป็น layer
- Layer สำหรับ user interface
- Layer สำหรับ business logic



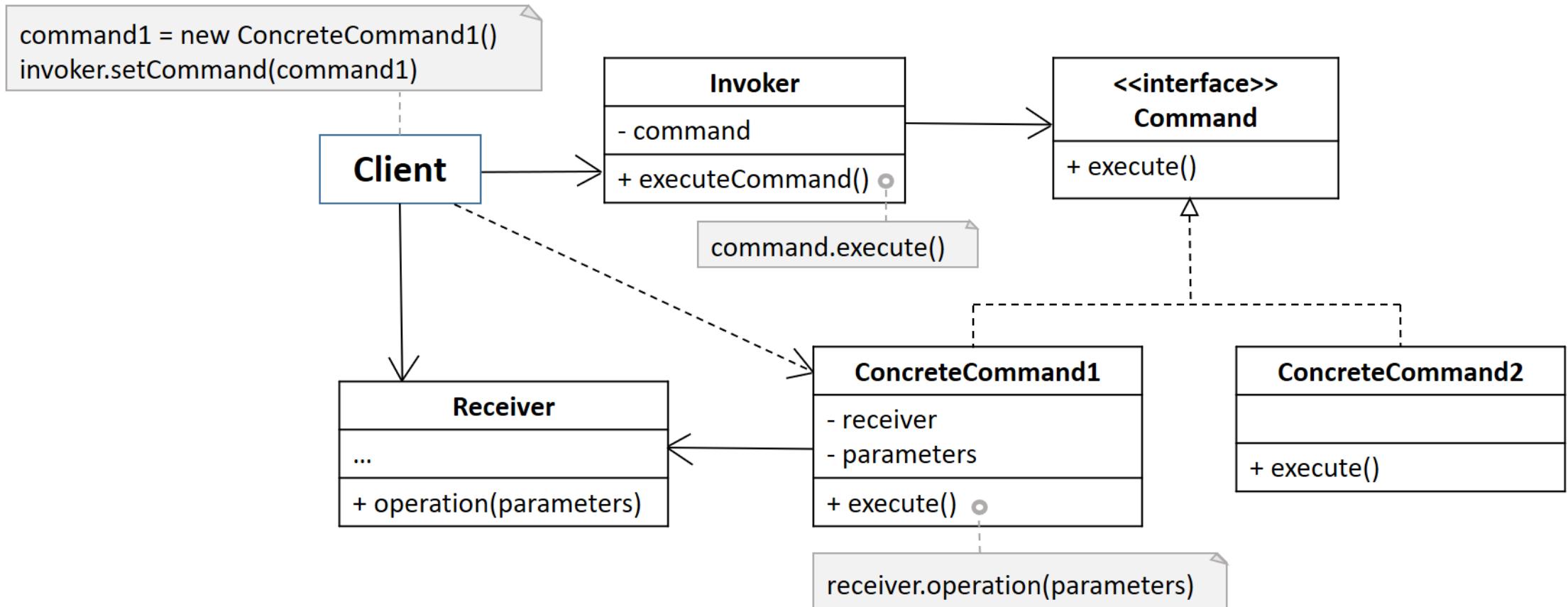
User interface



Business Logic



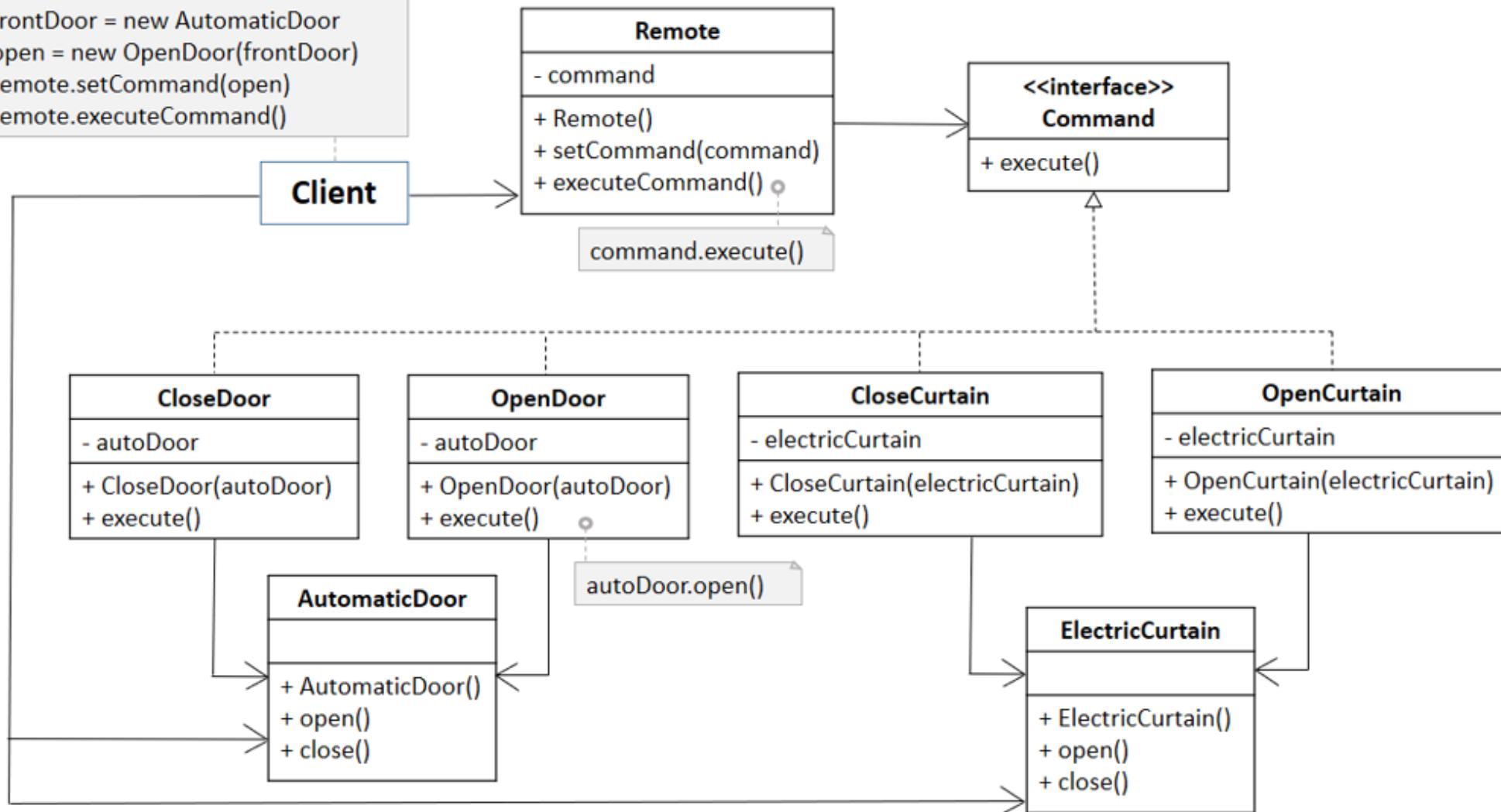
Structure



```

remote = new Remote
frontDoor = new AutomaticDoor
open = new OpenDoor(frontDoor)
remote.setCommand(open)
remote.executeCommand()

```



```
#include <iostream>
using namespace std;

class Command {
public:
    virtual ~Command() {
    }
    virtual void execute() = 0;
};

class HelloCommand: public Command {
private:
    string name;

public:
    HelloCommand(string n): name(n) {
    }
    void execute() {
        cout<<"Hello "<<name<<endl;
    }
};

class ElectricCurtain {
public:
    void open(string msg) {
        cout<<"open the curtain: "<<msg<<endl;
    }
    void close(string msg) {
        cout<<"close the curtain: "<<msg<<endl;
    }
};

class AutomaticDoor {
public:
    void open(string msg) {
        cout<<"open the door: "<<msg<<endl;
    }
    void close(string msg) {
        cout<<"close the door: "<<msg<<endl;
    }
};

class OpenCurtainCommand: public Command {
ElectricCurtain *curtain;
string msg;
public:
    OpenCurtainCommand(ElectricCurtain *c, string s): curtain(c), msg(s) {
    }

    void execute() {
        curtain->open(msg);
    }
};

class CloseCurtainCommand: public Command {
ElectricCurtain *curtain;
string msg;
public:
    CloseCurtainCommand(ElectricCurtain *c, string s) {
        curtain = c;
        msg = s;
    }

    void execute() {
        curtain->close(msg);
    }
};
```

```
class OpenDoorCommand: public Command {
    AutomaticDoor *door;
    string msg;
public:
    OpenDoorCommand(AutomaticDoor *d, string s): door(d), msg(s) {
    }

    void execute() {
        door->open(msg);
    }
};

class CloseDoorCommand: public Command {
    AutomaticDoor *door;
    string msg;
public:
    CloseDoorCommand(AutomaticDoor *d, string s): door(d), msg(s) {
    }

    void execute() {
        door->close(msg);
    }
};
```

```
class Remote {
    Command *command;
public:
    Remote() {
        command=0;
    }
    ~Remote() {
        delete command;
    }
    void setCommand(Command *c) {
        command = c;
    }
    void executeCommand() {
        command->execute();
    }
};
```

```
int main() {
    Remote *remote = new Remote;

    remote->setCommand(new HelloCommand("Somchai"));
    remote->executeCommand();
    cout<<endl;

    ElectricCurtain *curtain = new ElectricCurtain;
    AutomaticDoor *frontDoor = new AutomaticDoor;
    AutomaticDoor *bedroomDoor = new AutomaticDoor;

    remote->setCommand(new OpenDoorCommand(frontDoor, "Welcome home \\"Pat\\\""));
    remote->executeCommand();
    remote->setCommand(new CloseDoorCommand(frontDoor, "The front door is locked"));
    remote->executeCommand();
    cout<<endl;

    remote->setCommand(new OpenCurtainCommand(curtain, "Good morning"));
    remote->executeCommand();
    cout<<endl;

    remote->setCommand(new OpenDoorCommand(bedroomDoor, "Pat's bedroom door was opened"));
    remote->executeCommand();
    remote->setCommand(new CloseDoorCommand(bedroomDoor, "Pat's bedroom door was closed"));
    remote->executeCommand();
    cout<<endl;

    remote->setCommand(new CloseCurtainCommand(curtain, "Good night"));
    remote->executeCommand();
    cout<<endl;

    delete bedroomDoor;
    delete frontDoor;
    delete curtain;
    delete remote;

    return 0;
}
```

```
Hello Somchai
open the door: Welcome home "Pat"
close the door: The front door is locked
open the curtain: Good morning
open the door: Pat's bedroom door was opened
close the door: Pat's bedroom door was closed
close the curtain: Good night
```

```

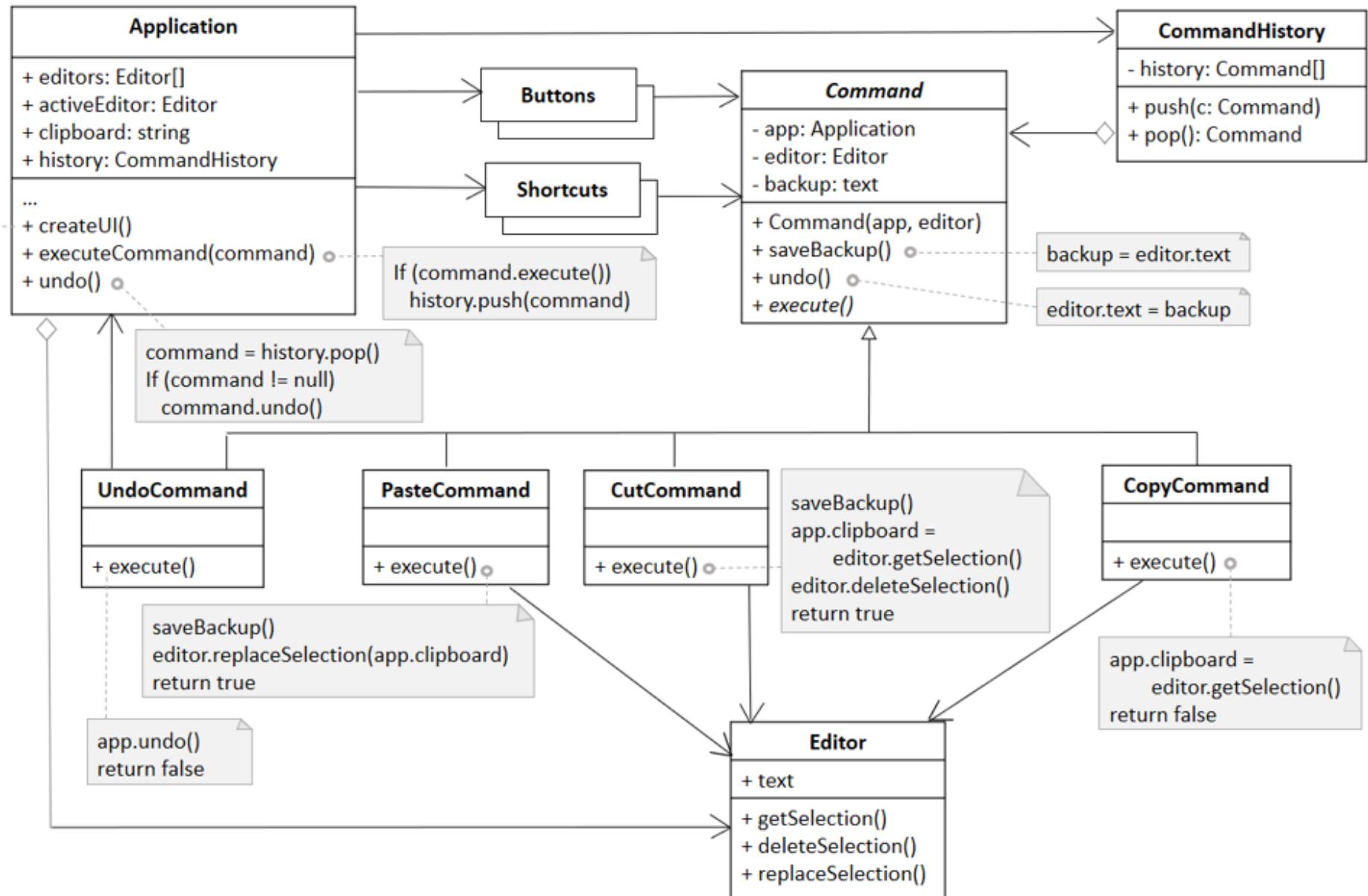
copy = function() { executeCommand(
  new copyCommand(this,activeEditor)) }
copyButton.setCommand(copy)
shortcuts.onKeyPress("Ctrl+C", copy)

cut = function() { executeCommand(
  new cutCommand(this,activeEditor)) }
cutButton.setCommand(cut)
shortcuts.onKeyPress("Ctrl+X", cut)

paste = function() { executeCommand(
  new pasteCommand(this,activeEditor)) }
pasteButton.setCommand(paste)
shortcuts.onKeyPress("Ctrl+V", paste)

undo = function() { executeCommand(
  new undoCommand(this,activeEditor)) }
undoButton.setCommand(undo)
shortcuts.onKeyPress("Ctrl+Z", undo)

```



ควรใช้เมื่อไหร่

- เมื่อต้องการจะใช้งานอ็อบเจกต์ในรูปแบบของ **operation**
 - เปลี่ยนการเรียกใช้เมธอดมาเป็นอ็อบเจกต์หนึ่งตัว
 - ทำให้สามารถส่ง **command** เป็นอาร์กูเมนต์ของเมธอด
 - เก็บ **command** นั้นไว้ในอ็อบเจกต์อื่นได้
 - เปลี่ยน **command** ได้ในช่วงเวลา晚 ไม่มี
- เมื่อต้องการจัดคิว **operation** หรือ จัดตารางการ **execute** หรือ ต้องการ **execute** จากระยะไกล
 - สามารถจัดอันดับของ **command** ได้
 - เปลี่ยน **command** ไปเป็น **string** ทำให้สามารถเขียนลงไฟล์หรือฐานข้อมูลได้ง่าย
 - สามารถคืนค่า **string** กลับไปเป็นอ็อบเจกต์ **command** ได้
 - ดังนั้น จึงสามารถที่จะ **delay** และจัดตารางการ **execute** ตัว **command** ได้
 - นอกจากรายละเอียดสามารถ จัดคิว **log** หรือส่ง **command** บน **network** ได้ด้วย
- เมื่อต้องการทำ **operation** ย้อนกลับ (**undo/redo**)

ข้อดี ข้อเสีย

- ข้อดี
 - Single Responsibility Principle
 - แยกคลาสที่เรียกว่า **operation** ออกจากคลาสที่ดำเนินการ **operation**
 - Open/Closed Principle
 - สามารถสร้าง **command** ใหม่ๆ เพิ่มได้โดยไม่กระทบกับโค้ดเดิม
 - สามารถทำ **undo/redo**
 - สามารถรวม **command** ง่ายๆ ไปเป็น **command** ที่ซับซ้อนได้
- ข้อเสีย
 - ได้ด้วยความซับซ้อน

การบ้าน

- พัฒนา

อ้างอิง

- <https://refactoring.guru/design-patterns/command>

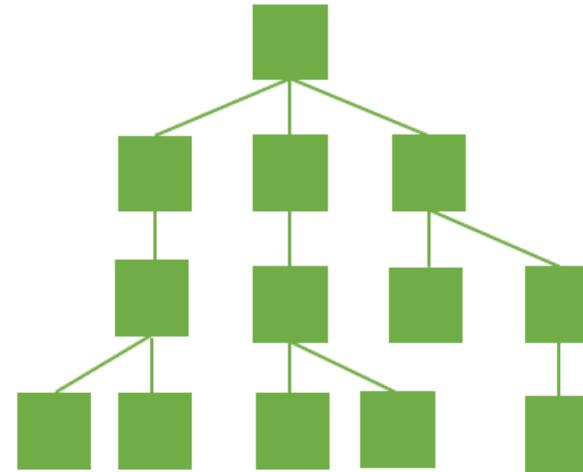
Catalog of Design Patterns

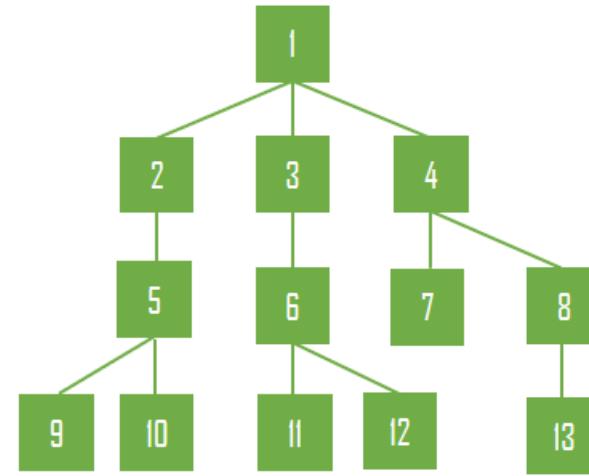
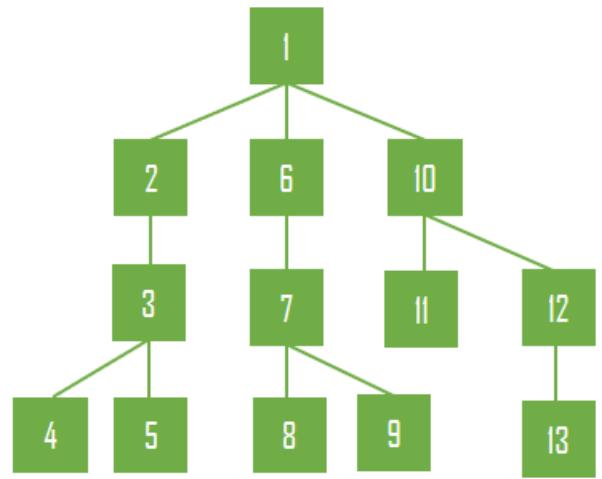
Behavioral Design Patterns

Iterator

Iterator

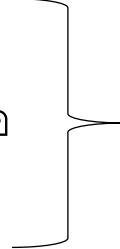
- การท่องเข้าไปยังสมาชิกของคอลเลคชัน โดยที่ไม่ต้องรู้ว่า รูปแบบการจัดเก็บข้อมูลของคอลเลคชันนั้นเป็นอย่างไร เช่นเป็น array หรือ stack หรือ tree เป็นต้น

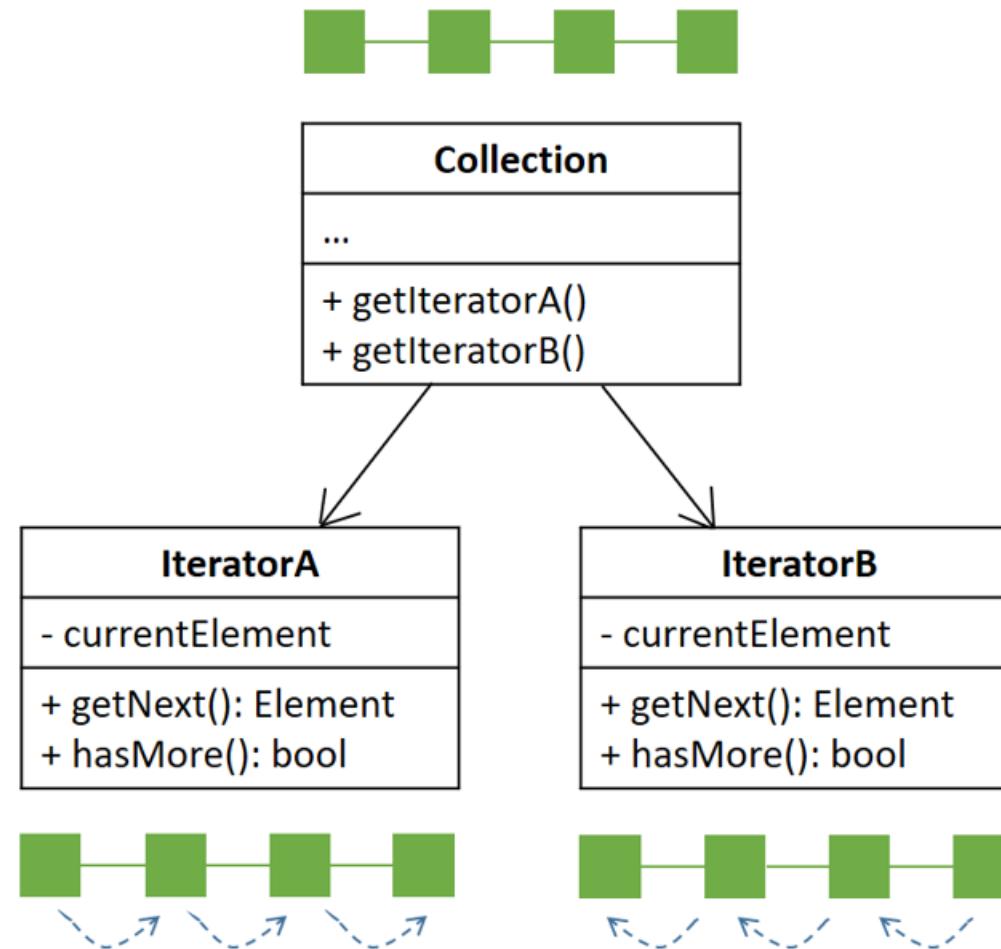




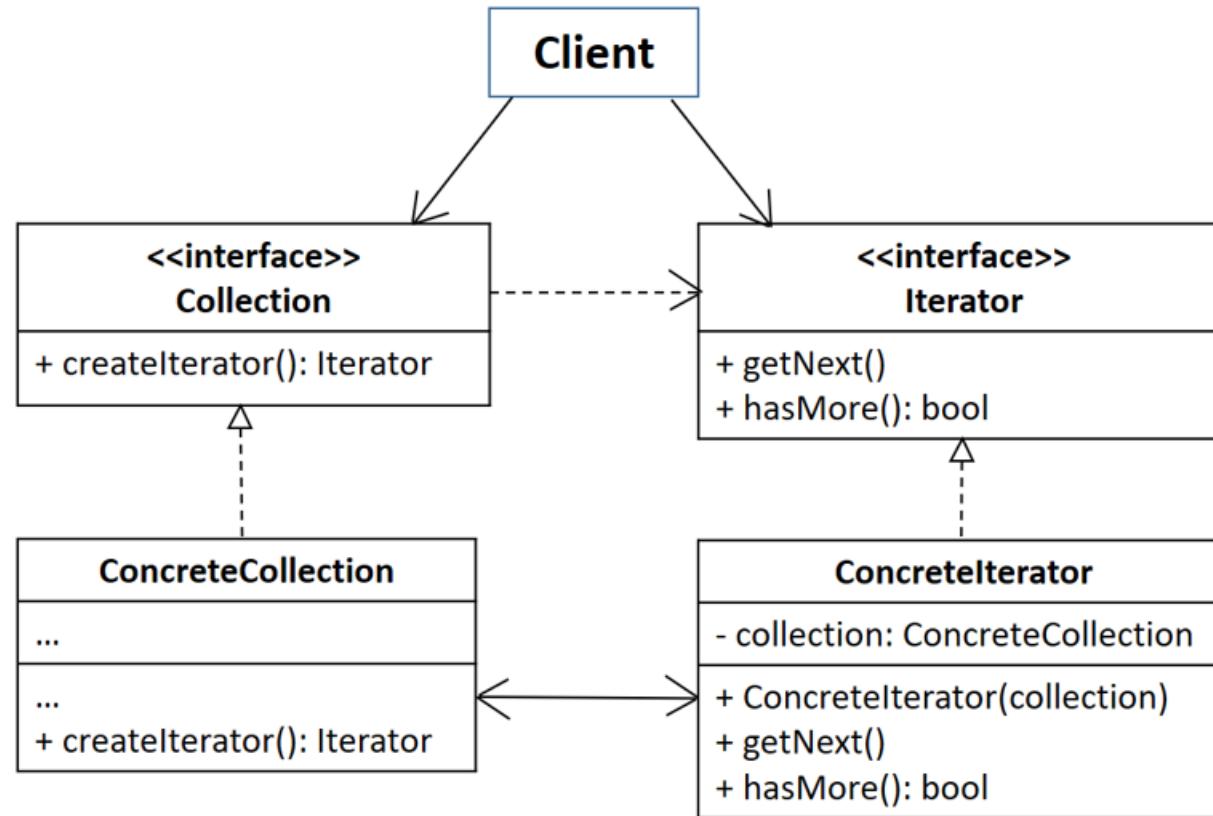
Iterator

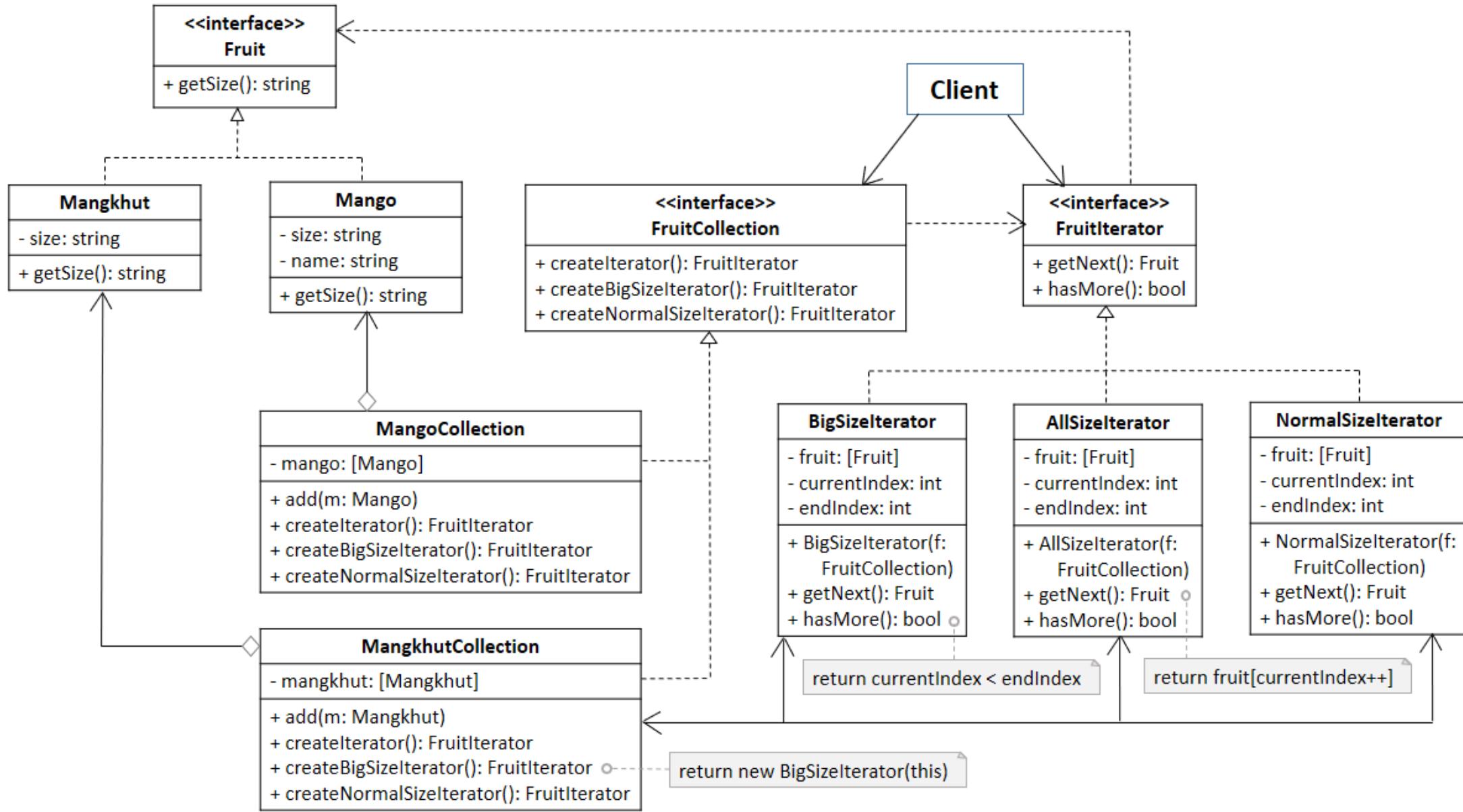
- แยกวิธีการท่องเข้าไปในคอลเลคชันออกมาเป็นอีกอีกชุดหนึ่ง โดยเรียกอีกชุดหนึ่งว่า **iterator**
 - อีกชุดหนึ่ง **Iterator** จะเก็บรายละเอียดทุกอย่างเกี่ยวกับการท่องเข้าไปในคอลเลคชัน เช่น ตำแหน่งปัจจุบัน และจำนวนสมาชิกที่เหลือที่จะต้องท่องไปจนกว่าจะหมด
 - ดังนั้น **iterator** หลายๆ ตัว สามารถท่องเข้าไปยัง **collection** เดียวกันได้พร้อมๆ กัน โดย **iterator** เหล่านี้จะเป็นอิสระต่อกัน ไม่ขึ้นต่อกัน

- ตัวอย่าง เที่ยวอยุธยา
 - เที่ยวเอง
 - ใช้ virtual guide app บนมือถือ
 - จ้าง local guide
- 
- นี่คือ **iterator** ที่จะท่องเข้าไปใน **collection** ที่ซึ่งประกอบด้วยจุดท่องเที่ยวในอยุธยา



Structure





```
#include <iostream>
using namespace std;

class Fruit {
public:
    virtual ~Fruit() {}
    virtual string getSize() = 0;
    virtual string getFruit() = 0;
};

class Mangkhut: public Fruit {
    string size;
public:
    Mangkhut(string s) {
        size = s;
    }
    string getFruit() {
        return "mangkhut size " + size;
    }
    string getSize() {
        return size;
    }
};

class Mango: public Fruit {
    string size;
    string name;
public:
    Mango(string n, string s) {
        size = s;
        name = n;
    }
    string getFruit() {
        return name + " mango size " + size;
    }
    string getSize() {
        return size;
    }
};
```

```
class FruitIterator {
public:
    virtual ~FruitIterator() {}
    virtual Fruit* getNext() = 0;
    virtual bool hasMore() = 0;
};

class FruitCollection {
public:
    virtual ~FruitCollection() {}
    virtual FruitIterator* createIterator() = 0;
    virtual FruitIterator* createBigSizeIterator() = 0;
    virtual FruitIterator* createNormalSizeIterator() = 0;
    virtual Fruit* operator[](int) = 0;
};
```

```
class BigSizeIterator: public FruitIterator {
    Fruit *fruit[7];
    int currentIndex;
    int endIndex;
public:
    BigSizeIterator(FruitCollection *f) {
        int j=0;
        for (int i=0; i<7; i++)
            if ((*f)[i]->getSize() == "big")
                fruit[j++] = (*f)[i];
        endIndex=j;
        currentIndex=0;
    }
    Fruit* getNext() {
        return fruit[currentIndex++];
    }
    bool hasMore() {
        return currentIndex < endIndex;
    }
};
```

```
class NormalSizeIterator: public FruitIterator {
    Fruit *fruit[7];
    int currentIndex;
    int endIndex;
public:
    NormalSizeIterator(FruitCollection *f) {
        int j=0;
        for (int i=0; i<7; i++)
            if ((*f)[i]->getSize() == "normal")
                fruit[j++] = (*f)[i];
        endIndex=j;
        currentIndex=0;
    }
    Fruit* getNext() {
        return fruit[currentIndex++];
    }
    bool hasMore() {
        return currentIndex < endIndex;
    }
};
```

```
class AllSizeIterator: public FruitIterator {
    Fruit *fruit[7];
    int currentIndex;
    int endIndex;
public:
    AllSizeIterator(FruitCollection *f) {
        int j=0;
        for (int i=0; i<7; i++)
            fruit[j++] = (*f)[i];
        endIndex=j;
        currentIndex=0;
    }
    Fruit* getNext() {
        return fruit[currentIndex++];
    }
    bool hasMore() {
        return currentIndex < endIndex;
    }
};
```

```
class MangkhutCollection: public FruitCollection {
    int currentIndex;
    Mangkhut *mangkhutList[7];
public:
    MangkhutCollection() {
        currentIndex = 0;
        for (int i=0; i<7; i++)
            mangkhutList[i]=0;
    }
    ~MangkhutCollection() {
        for (int i=0; i<7; i++) {
            delete mangkhutList[i];
        }
    }
    void add(Mangkhut *m) {
        mangkhutList[currentIndex++] = m;
    }
    Mangkhut* operator[](int i) {
        return mangkhutList[i];
    }
    FruitIterator* createBigSizeIterator() {
        return new BigSizeIterator(this);
    }
    FruitIterator* createNormalSizeIterator() {
        return new NormalSizeIterator(this);
    }
    FruitIterator* createIterator() {
        return new AllSizeIterator(this);
    }
};
```

```
class MangoCollection: public FruitCollection {
    int currentIndex;
    Mango *mangoList[7];
public:
    MangoCollection() {
        currentIndex = 0;
        for (int i=0; i<7; i++)
            mangoList[i]=0;
    }
    ~MangoCollection() {
        for (int i=0; i<7; i++) {
            delete mangoList[i];
        }
    }
    void add(Mango *m) {
        mangoList[currentIndex++] = m;
    }
    Mango* operator[](int i) {
        return mangoList[i];
    }
    FruitIterator* createBigSizeIterator() {
        return new BigSizeIterator(this);
    }
    FruitIterator* createNormalSizeIterator() {
        return new NormalSizeIterator(this);
    }
    FruitIterator* createIterator() {
        return new AllSizeIterator(this);
    }
};
```

```

void showFruit(FruitIterator *it) {
    while (it->hasMore()) {
        cout<<it->getNext()->getFruit()<<endl;
    }
    delete it;
}

int main() {
    MangkhutCollection *m1 = new MangkhutCollection;
    m1->add(new Mangkhut("big"));
    m1->add(new Mangkhut("normal"));
    m1->add(new Mangkhut("big"));
    m1->add(new Mangkhut("big"));
    m1->add(new Mangkhut("normal"));
    m1->add(new Mangkhut("big"));
    m1->add(new Mangkhut("normal"));

    showFruit(m1->createBigSizeIterator());
    cout<<endl;

    showFruit(m1->createNormalSizeIterator());
    cout<<endl;

    showFruit(m1->createIterator());
    cout<<endl;

    MangoCollection *m2 = new MangoCollection;
    m2->add(new Mango("Namdokmai", "big"));
    m2->add(new Mango("Namdokmai", "normal"));
    m2->add(new Mango("Mun", "big"));
    m2->add(new Mango("Okrong", "big"));
    m2->add(new Mango("Okrong", "normal"));
    m2->add(new Mango("Bao", "normal"));
    m2->add(new Mango("Bao", "big"));
}

```

```

showFruit(m2->createBigSizeIterator());
cout<<endl;

showFruit(m2->createNormalSizeIterator());
cout<<endl;

showFruit(m2->createIterator());
cout<<endl;

delete m1;
delete m2;
return 0;
}

```

mangkhut	size	big
mangkhut	size	normal
mangkhut	size	normal
mangkhut	size	normal
mangkhut	size	big
mangkhut	size	normal
mangkhut	size	big
mangkhut	size	big
mangkhut	size	normal
mangkhut	size	big
Namdokmai	mango	size
Mun	mango	size
Okrong	mango	size
Bao	mango	size
Namdokmai	mango	size
Okrong	mango	size
Bao	mango	size
Namdokmai	mango	size
Namdokmai	mango	size
Mun	mango	size
Okrong	mango	size
Okrong	mango	size
Bao	mango	size
Bao	mango	size

ควรใช้เมื่อไหร่

- เมื่อคอลเลคชันมีโครงสร้างข้อมูลที่ซับซ้อน และเราต้องการซ่อนความซับซ้อนนั้นเอาไว้
- เมื่อต้องการลดความซ้ำซ้อนของโค้ดในการท่องเข้าไปในคอลเลคชัน
- เมื่อต้องการให้โค้ดสามารถท่องไปในโครงสร้างข้อมูลที่แตกต่างกันได้ หรือในกรณีที่ไม่ทราบชนิดของโครงสร้างข้อมูลมาก่อน
 - เนื่องจากมีการใช้ **interface** สำหรับ **collection** และ **iterator** ดังนั้นโค้ดจึงสามารถทำงานได้กับ **concrete collection** และ **concrete iterator** ที่หลากหลาย

ข้อดี ข้อเสีย

- ข้อดี
 - Single Responsibility Principle
 - มีการแยกอัลกอริทึมสำหรับการทำงานเข้าไปในคอลเลคชันเป็นคลาสแยกออกจาก
 - Open/Closed Principle
 - สามารถสร้าง collection และ iterator เพิ่มเติมได้โดยไม่กระทบกับโค้ดเดิม
 - สามารถท่องเข้าไปในคอลเลคชันเดียวกัน พร้อมๆ กันได้ เนื่องจาก iterator แต่ละตัวจะมี state เป็นของตัวเอง
- ข้อเสีย
 - โค้ดมีความซับซ้อน

การบัน

- เก็บข้อมูลเพื่อ存และครอบครัว
 - สามารถเข้าถึงข้อมูลได้หลายแบบดังนี้
 - เพื่อนสนิท
 - เพื่อนไม่สนิท
 - ครอบครัวสนิท
 - ครอบครัวไม่สนิท

อ้างอิง

- <https://refactoring.guru/design-patterns/iterator>

Catalog of Design Patterns

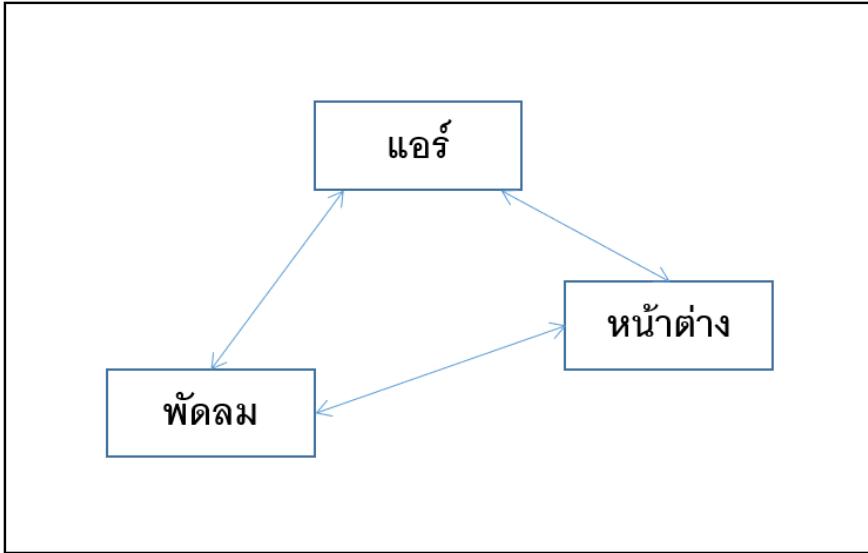
Behavioral Design Patterns

Mediator

Mediator หรือ Intermediary หรือ Controller

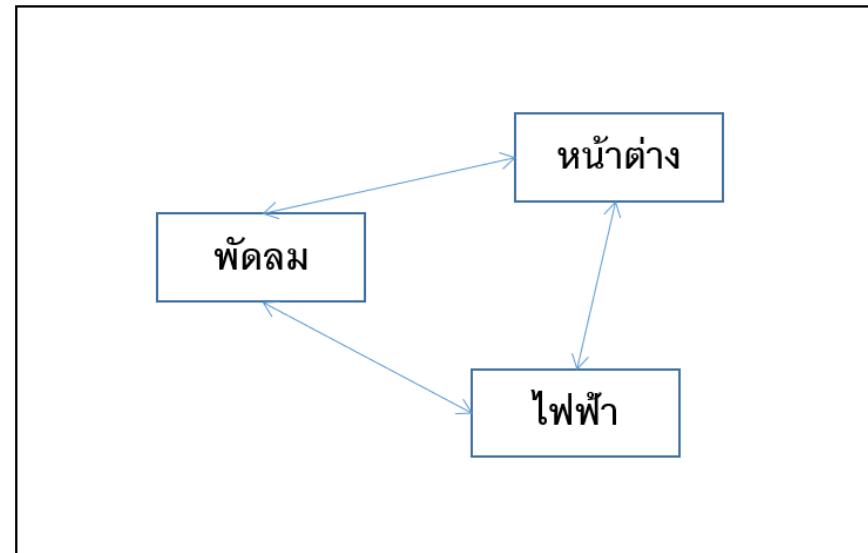
- ลดการขึ้นต่อกันระหว่างอ้อมบเจกต์
 - จำกัดการติดต่อสื่อสารกันระหว่างอ้อมบเจกต์ โดยให้ติดต่อกันผ่าน mediator เท่านั้น

ห้องนอน



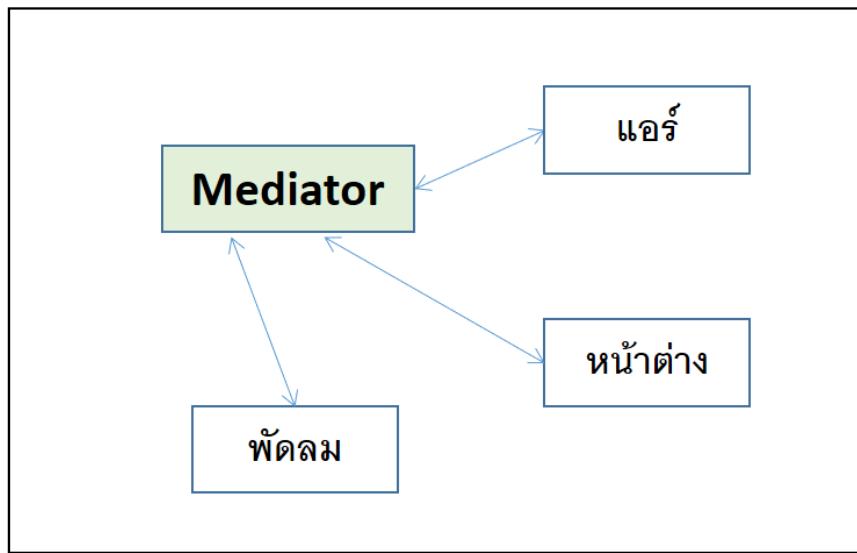
เปิดแอร์ ปิดหน้าต่าง ปิดพัดลม
ปิดแอร์ เปิดหน้าต่าง เปิดพัดลม
เปิดหน้าต่าง ปิดแอร์

ห้องน้ำ

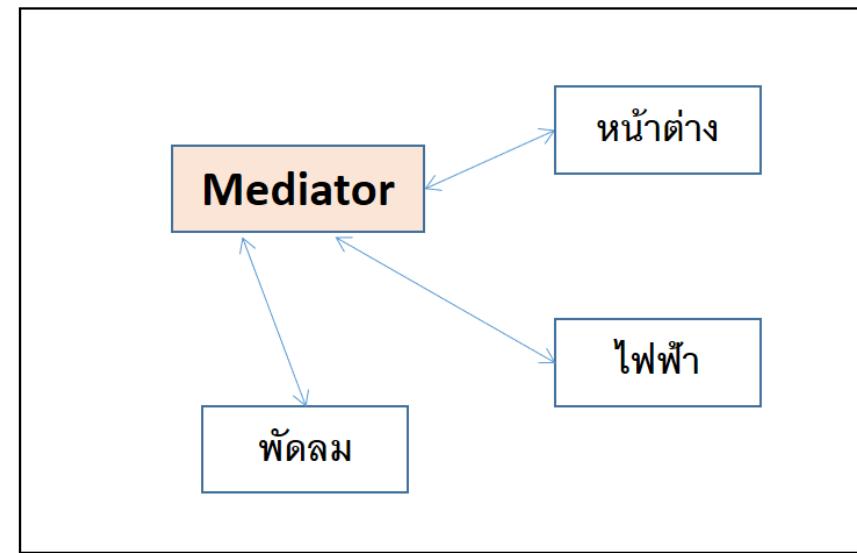


เปิดไฟ เปิดพัดลม เปิดหน้าต่าง
ปิดไฟ ปิดพัดลม ปิดหน้าต่าง
เปิดพัดลม เปิดหน้าต่าง
ปิดพัดลม ปิดหน้าต่าง

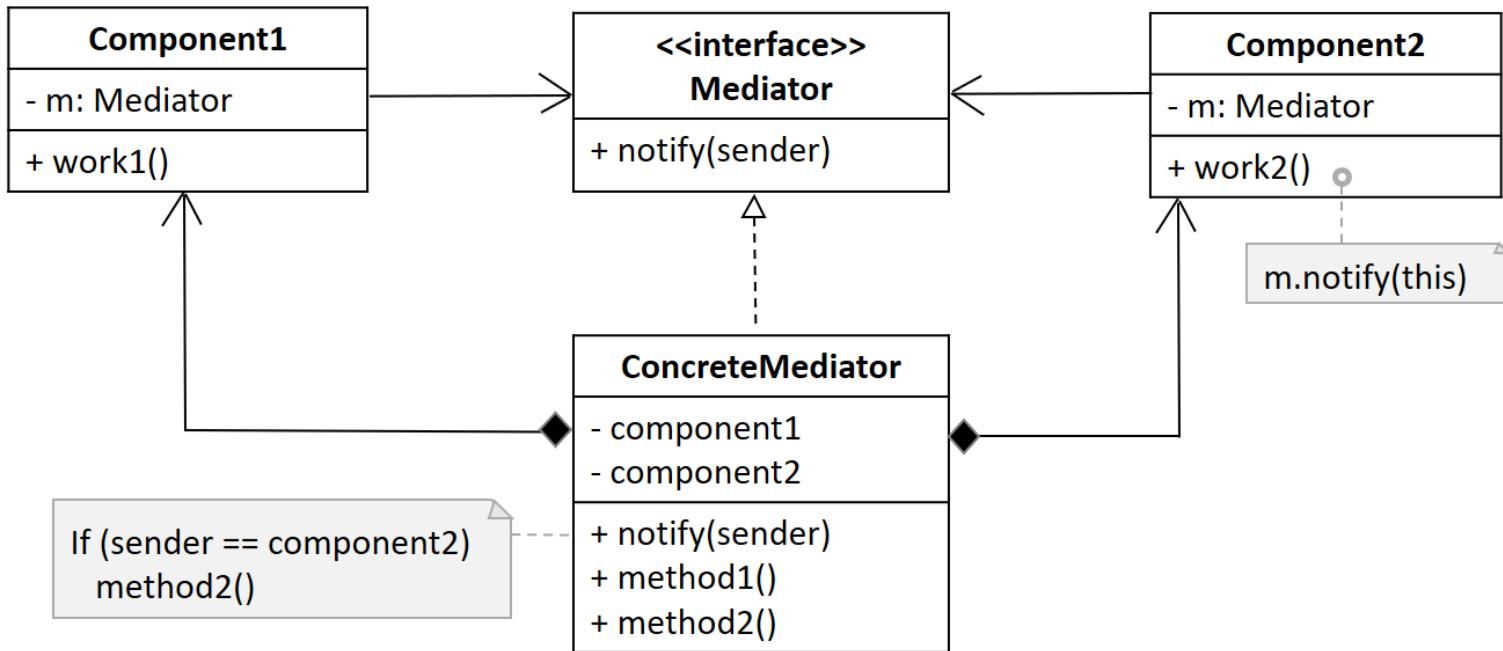
ห้องนอน



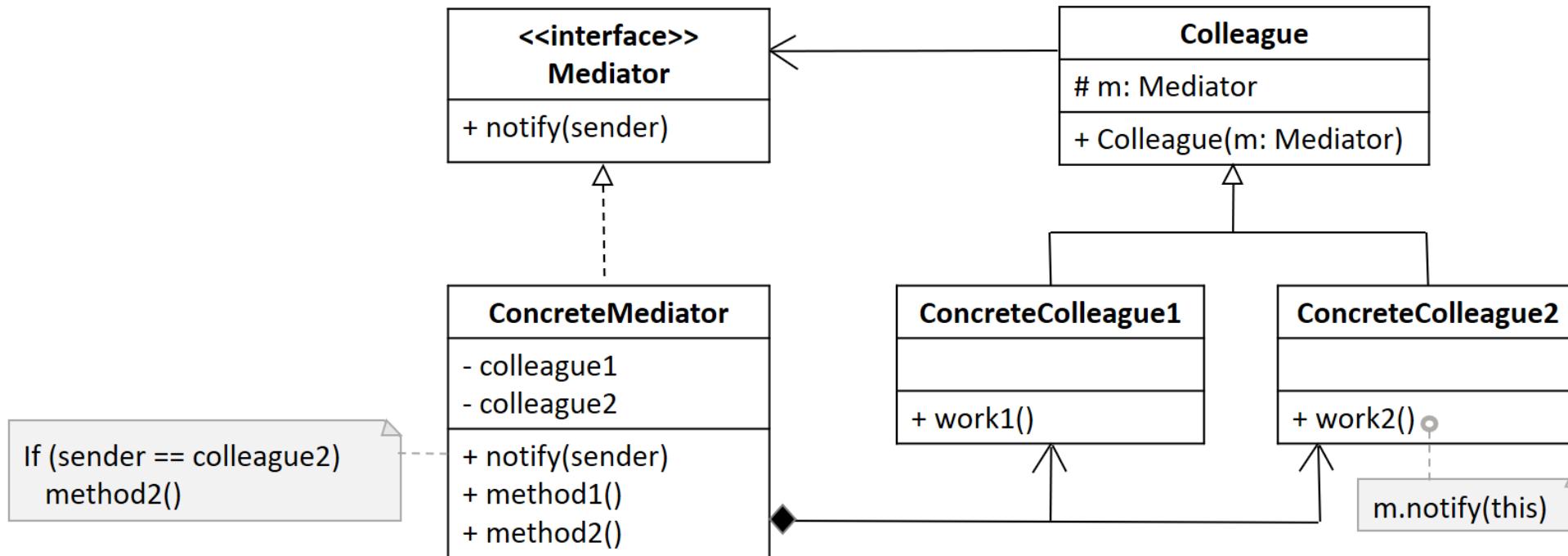
ห้องน้ำ

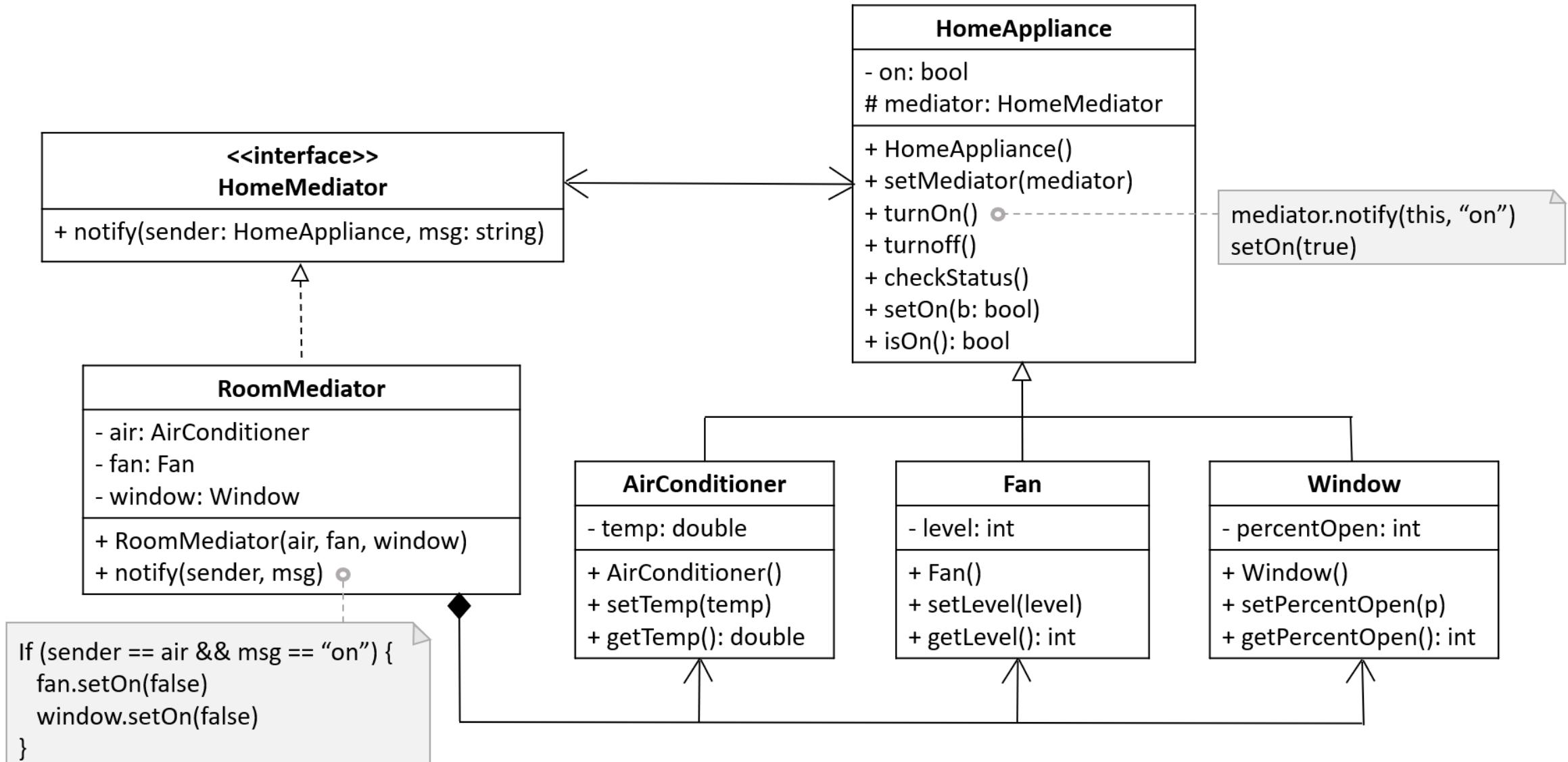


Structure



Structure





```
#include <iostream>
using namespace std;

class HomeAppliance;
class HomeMediator {
public:
    virtual void notify(HomeAppliance *sender, string msg) = 0;
};

class HomeAppliance {
    string name;
    bool on;
protected:
    HomeMediator *mediator;
    void setName(string n) {
        name = n;
    }
public:
    HomeAppliance() {
        on = false;
        mediator = 0;
    }
    void setMediator(HomeMediator *mediator) {
        this->mediator = mediator;
    }
    void turnOn() {
        setOn(true);
        mediator->notify(this, "on");
    }
    void turnOff() {
        setOn(false);
        mediator->notify(this, "off");
    }
}
```

```
void checkStatus() {
    mediator->notify(this, "check");
}
void setOn(bool b) {
    on = b;
}
bool isOn() {
    return on;
}
string getName() {
    return name;
}

class AirConditioner: public HomeAppliance {
    double temp;
public:
    AirConditioner() {
        setTemp(25);
        setName("Air");
    }
    void setTemp(double t) {
        temp = t;
    }
    double getTemp() {
        return temp;
    }
};
```

```
class Fan: public HomeAppliance {
    int level;
public:
    Fan() {
        setLevel(1);
        setName("Fan");
    }
    void setLevel(int t) {
        level = t;
    }
    int getLevel() {
        return level;
    }
};
```

```
class Window: public HomeAppliance {
    int percentOpen;
public:
    Window() {
        setPercentOpen(0);
        setName("Window");
    }
    void setPercentOpen(int t) {
        percentOpen = t;
    }
    int getPercentOpen() {
        return percentOpen;
    }
};
```

```
class RoomMediator: public HomeMediator {
    AirConditioner *air;
    Fan *fan;
    Window *window;

public:
    RoomMediator(AirConditioner *a, Fan *f, Window *w): air(a), fan(f), window(w) {
        this->air->setMediator(this);
        this->fan->setMediator(this);
        this->window->setMediator(this);
    }
    void notify(HomeAppliance *sender, string msg) {
        if (sender->getName() == "Air" && msg == "on") {
            cout<<"=> The air conditioner is on."<<endl;
            cout<<"=> The fan will be turned off and the window will be closed."<<endl;
            fan->setOn(false);
            window->setOn(false);
        }
        if (sender->getName() == "Window" && msg == "on") {
            window->setPercentOpen(50);
            cout << "=>The window is opened.\n=>The air conditioner will be turned off."<<endl;
            air->setOn(false);
        }
        if (sender->getName() == "Fan" && msg == "on") {
            cout << "=>The Fan is turn on."<<endl;
        }
    }
}
```

```
if (msg == "off") {
    if (sender->getName() == "window")
        cout<<"=>The window is closed" << endl;
    else
        cout<<"=>The "<< sender->getName() << " is turn off." << endl;
}

if (msg == "check") {
    cout<<"ALL Status" << endl;
    if (air->isOn())
        cout<<"Air: turn on "<< air->getTemp() << " degree" << endl;
    else
        cout<<"Air: turn off" << endl;
    if (fan->isOn())
        cout<<"Fan: turn on, level "<< fan->getLevel() << endl;
    else
        cout<<"Fan: turn off" << endl;
    if (window->isOn())
        cout<<"Window: open "<< window->getPercentOpen() << " percent" << endl;
    else
        cout<<"Window: close" << endl;
}
};
```

```

void client() {
    AirConditioner *air = new AirConditioner;
    Fan *fan = new Fan;
    Window *window = new Window;

    RoomMediator *mediator = new RoomMediator(air, fan, window);

    air->turnOn();
    cout<<endl;
    air->checkStatus();

    cout<<endl<<"decrease temperature"<<endl;
    air->setTemp(22);
    cout<<endl;
    air->checkStatus();

    cout<<endl;
    window->turnOff();
    cout<<endl;
    window->checkStatus();

    cout<<endl;
    window->turnOn();
    cout<<endl;
    window->checkStatus();
}

int main() {
    client();
    return 0;
}

```

=> The air conditioner is on.
=> The fan will be turned off and the window will be closed.

ALL Status
Air: turn on 25 degree
Fan: turn off
Window: close

decrease temperature

ALL Status
Air: turn on 22 degree
Fan: turn off
Window: close

=>The Window is turn off.

ALL Status
Air: turn on 22 degree
Fan: turn off
Window: close

=>The window is opened.
=>The air conditioner will be turned off.

ALL Status
Air: turn off
Fan: turn off
Window: open 50 percent

=>The Fan is turn on.

ALL Status
Air: turn off
Fan: turn on, level 1
Window: open 50 percent

=>The Fan is turn off.

ALL Status
Air: turn off
Fan: turn off
Window: open 50 percent

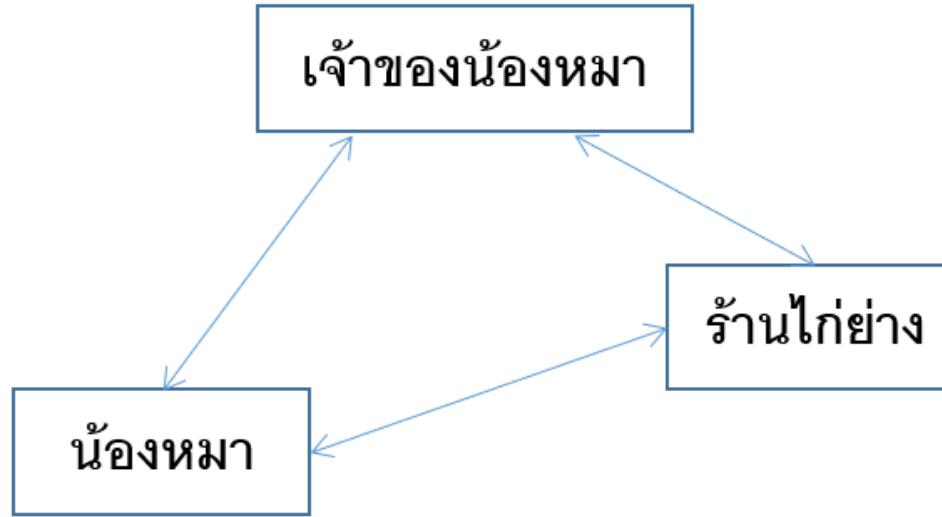
ควรใช้เมื่อไหร่

- เมื่อมีความยากลำบากในการแก้ไขเปลี่ยนแปลงคลาสเนื่องจากมีการผูกอยู่กับคลาสอื่นๆ
 - ความสัมพันธ์ระหว่างคลาสจะถูกแยกออกจากเป็นอีกคลาสนึง โดยใช้ **mediator**
- เมื่อไม่สามารถนำคอมโพเนนต์ต่างๆ ไป reuse ในโปรแกรมอื่นได้ เนื่องจากมีการขึ้นต่อ กัน
 - เมื่อ reuse คอมโพเนนต์ในแอพอื่นก็ทำการสร้าง mediator ใหม่
- เมื่อมีการสร้างคลาสลูกจำนวนมากเพื่อที่จะ reuse งานบางอย่าง
 - เนื่องจากความสัมพันธ์ถูกเก็บอยู่ใน mediator ดังนั้นจึงสามารถสร้าง mediator ใหม่ๆ ให้เก็บความสัมพันธ์แบบใหม่ๆ ได้

ข้อดี ข้อเสีย

- ข้อดี
 - Single Responsibility Principle
 - มีการแยกความสัมพันธ์ระหว่างคอมโพเนนต์ออกจากกัน
 - Open/Closed Principle
 - สามารถสร้าง mediator ใหม่ๆ ได้โดยไม่กระทบกับของเดิม
 - ลด coupling ระหว่างคอมโพเนนต์
 - สามารถ reuse แต่ละคอมโพเนนต์ได้ง่าย
- ข้อเสีย
 - God Object

การบ้าน



- น้องหมา **หิว** เจ้าของน้องหมาไป **ซื้อไก่ย่าง** ให้น้องหมา ร้านไก่ย่าง **ขายไก่** ได้ น้องหมา **ได้กินไก่ย่าง**
- เจ้าของน้องหมาไป **ซื้อไก่ย่าง** และ **แบ่งน้องหมากินทุกครั้ง**
- น้องหมา **ไปขอไก่ย่าง** ร้านขาย **ไก่ย่าง** ร้านขาย **ไก่ย่าง** **ไปเก็บเงินกับเจ้าของน้องหมา**
- ร้านขาย **ไก่ย่าง** **ให้ฟรี** **ไก่ย่างกับน้องหมา** น้องหมา **ได้กินไก่ฟรี**

อ้างอิง

- <https://refactoring.guru/design-patterns/mediator>

Catalog of Design Patterns

Behavioral Design Patterns

Memento

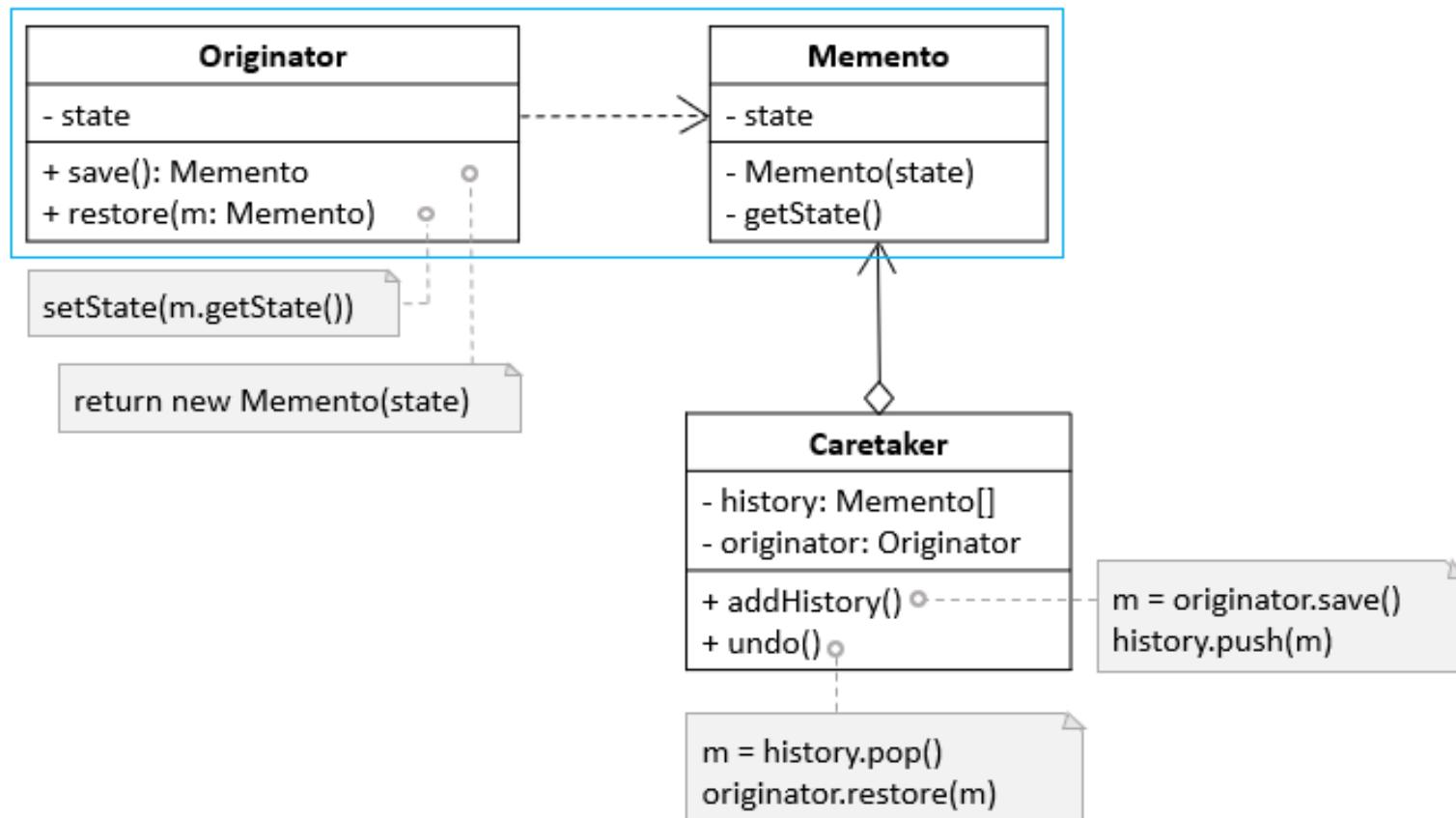
Memento หรือ Snapshot

- สามารถ **save** และ **restore** สถานะก่อนหน้าของอ็อบเจกต์ โดยไม่ต้องรู้รายละเอียดของการอิมเพลเม้นท์

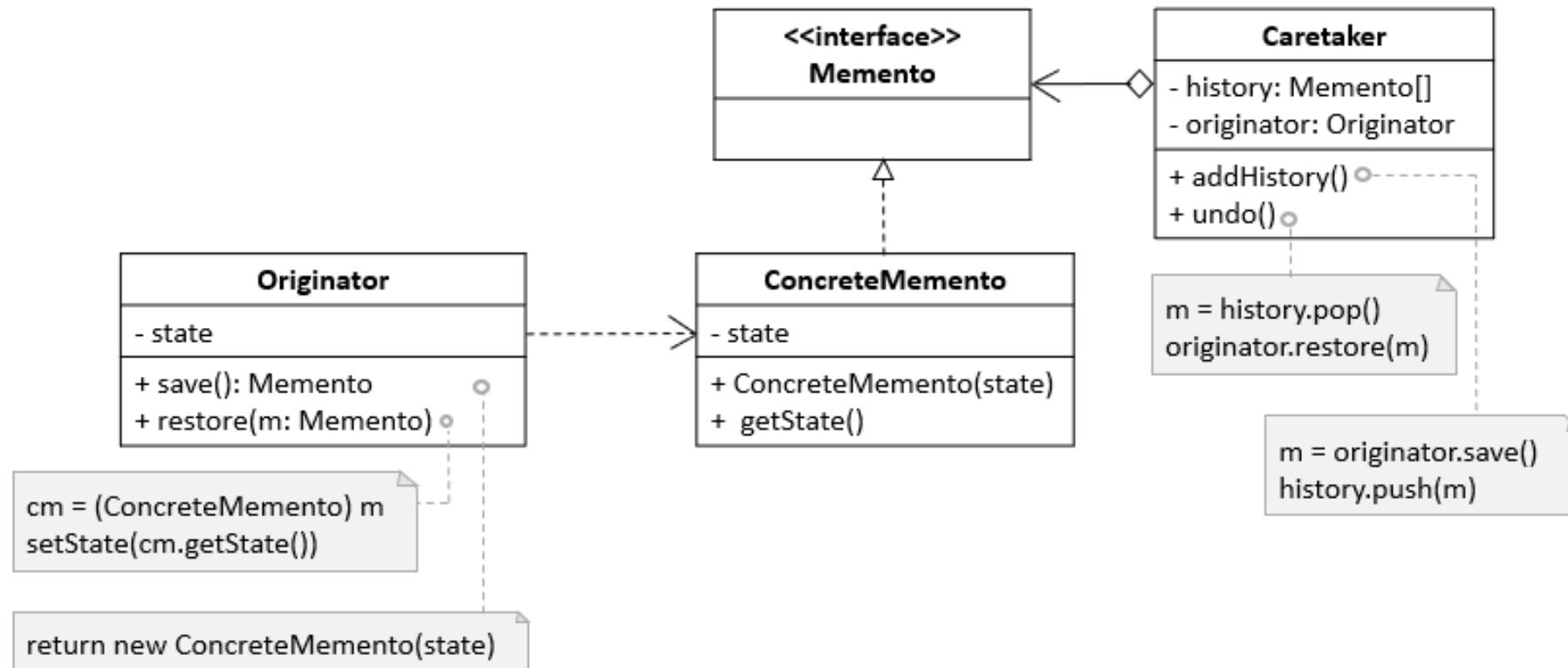
Memento

- ปัญหาในการเก็บข้อมูลสถานะของอ็อกบเจกต์
 - การเข้าถึงข้อมูล **private** ของแต่ละคลาส
 - การปรับเปลี่ยนรายละเอียดของคลาส เช่น เพิ่ม หรือ ลบ และทริบิวต์
 - ต้องเก็บข้อมูลอะไรบ้าง
 - เก็บอ็อกบเจกต์ของคลาสที่ประกอบด้วยทริบิวต์ที่ก็อปปีสถานะมา และทำเป็น **public** ?
 - ไม่เป็นไปตามกฎ **encapsulation**
- วิธีแก้
 - ให้เจ้าของหรือ **originator** เป็นคนสร้าง **snapshot** เอง แทนที่จะให้อ็อกบเจกต์อื่นมาทำ
 - เก็บตัวก็อปปี้ไว้ใน **memento** เนพาะเจ้าของเท่านั้นที่สามารถเข้าถึงได้
 - อ็อกบเจกต์อื่นๆ ติดต่อกับ **memento** ผ่าน **interface**
 - สามารถเก็บ **memento** ไว้ในอ็อกบเจกต์อื่นได้ เช่นเก็บใน **caretaker**
 - **caretaker** จะทำงานกับ **memento** ผ่าน **interface**

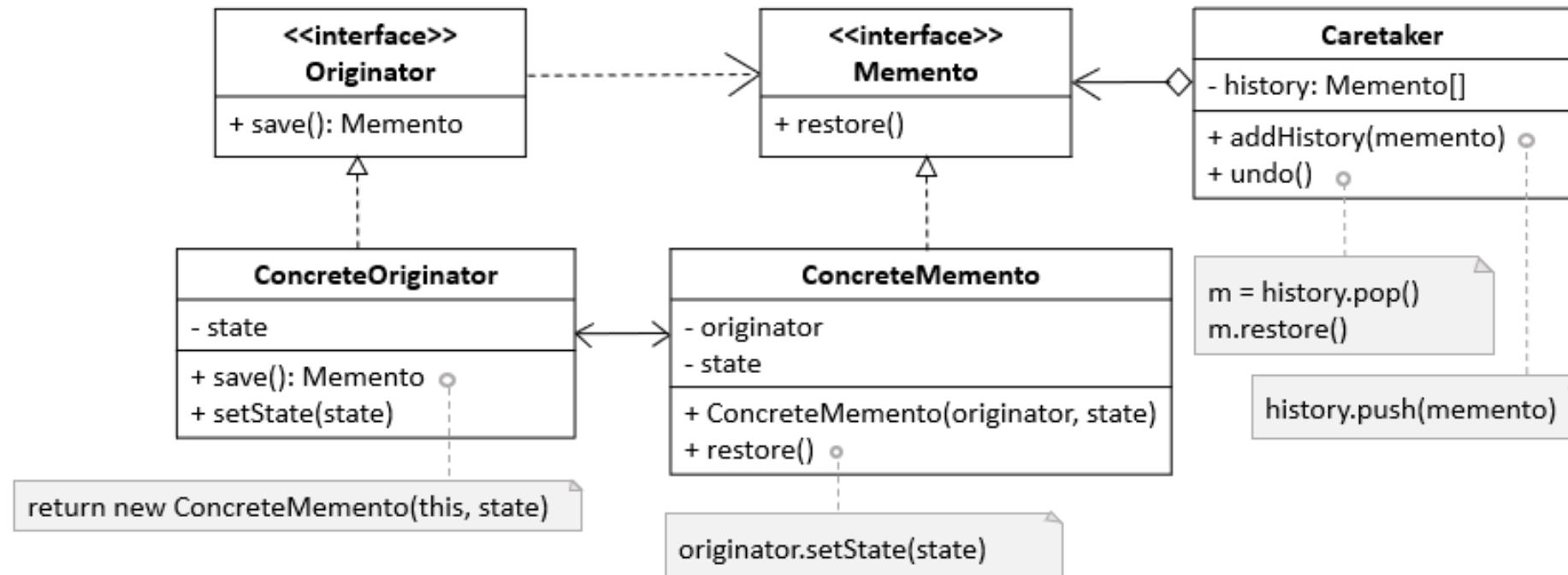
Structure 1 ໃຊ້ nested classes (C++, C#, Java)

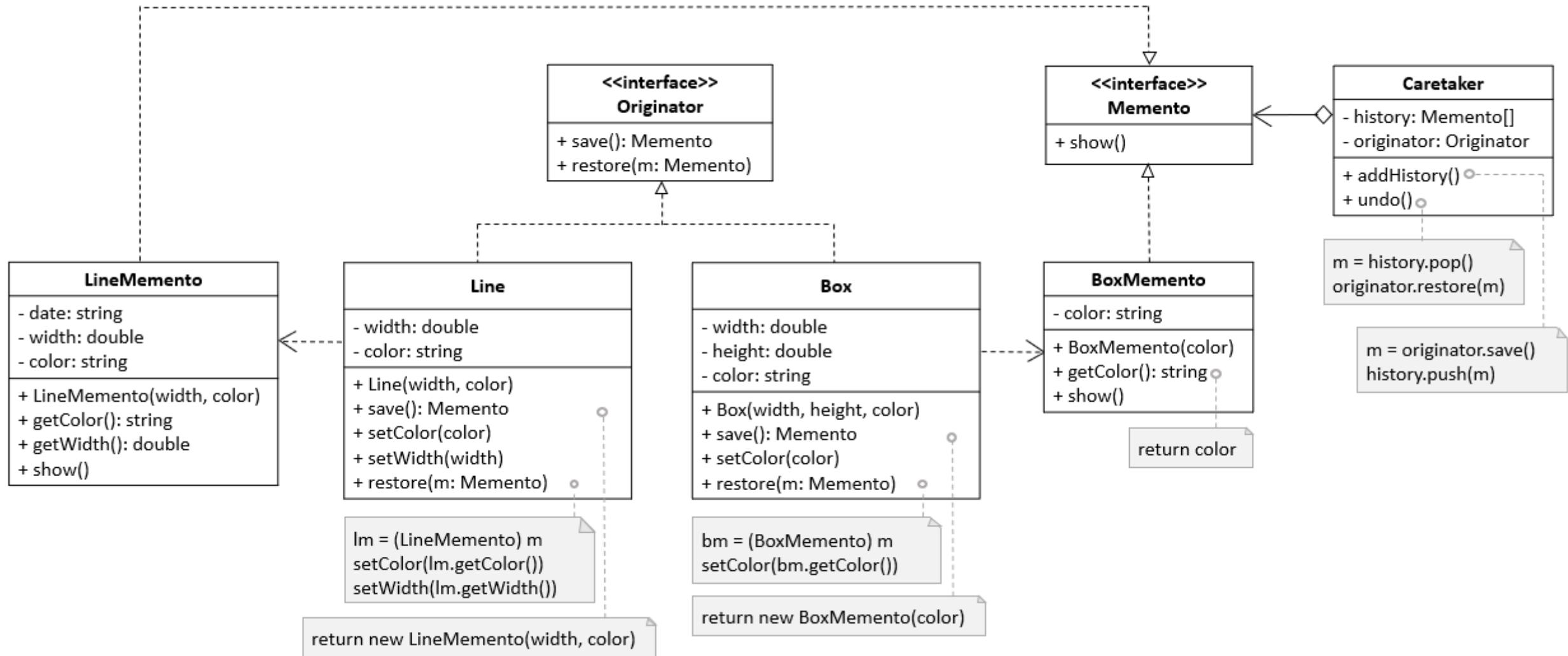


Structure 2 ໃຊ້ interface



Structure 3





```
#include <iostream>
#include <list>
#include <ctime>
using namespace std;

class Memento {
public:
    virtual void show() = 0;
};

class BoxMemento: public Memento {
string color;
public:
    BoxMemento(string s) {
        color = s;
    }
    string getColor() {
        return color;
    }
    void show() {
        cout<<"Box color: "<<color<<endl;
    }
};
```

```
class LineMemento: public Memento {
string date;
double width;
string color;
public:
    LineMemento(double w, string s) {
        time_t now = time(0);
        date = ctime(&now);
        width = w;
        color = s;
    }
    string getColor() {
        return color;
    }
    double getWidth() {
        return width;
    }
    void show() {
        cout<<"Line width: "<<width<< " , color: "<<color<<" => "<<date<<endl;
    }
};

class Originator {
public:
    virtual Memento* save() = 0;
    virtual void restore(Memento*) = 0;
};
```

```
class Box: public Originator {
    double width;
    double height;
    string color;
public:
    Box(double w, double h, string c) {
        width = w;
        height = h;
        color = c;
    }
    void setColor(string s) {
        color = s;
    }
    string getColor() {
        return color;
    }
    double getWidth() {
        return width;
    }
    double getHeight() {
        return height;
    }
    Memento *save() {
        return new BoxMemento(color);
    }
    void restore(Memento *m) {
        BoxMemento *bm = (BoxMemento*) m;
        cout<<"Restore "<<color<<" to "<<bm->getColor()<<" [box"<<width<<"x"<<height<<"]"<<endl;
        setColor(bm->getColor());
    }
};
```

```
class Line: public Originator {
    double width;
    string color;
public:
    Line(double w, string s) {
        width = w;
        color = s;
    }
    void setColor(string c) {
        color = c;
    }
    string getColor() {
        return color;
    }
    void setWidth(double w) {
        width = w;
    }
    double getWidth() {
        return width;
    }
    Memento *save() {
        return new LineMemento(width, color);
    }
    void restore(Memento *m) {
        LineMemento *lm = (LineMemento*) m;
        cout<<"Restore "<<color<<" to "<<lm->getColor()<<" and "<<width<<" to "<<lm->getWidth()<<endl;
        setColor(lm->getColor());
        setWidth(lm->getWidth());
    }
    void show() {
        cout<<"Now Line: "<<color<<" "<<width<<endl;
    }
};
```

```
class Caretaker {
    list<Memento*> history;
    Originator *originator;
public:
    Caretaker(Originator *o) {
        originator = o;
    }
    void addHistory() {
        history.push_back(originator->save());
    }
    void undo() {
        if (!history.size()) {
            return;
        }
        Memento *memento = history.back();
        originator->restore(memento);
        history.pop_back();
    }
    void showHistory() {
        cout<<"History"<<endl;
        for ( list<Memento*>::iterator it = history.begin(); it != history.end(); it++ ) {
            (*it)->show();
        }
    }
};
```

```
for (Memento* m : history)
    m->show();
```

```
void clientBox() {
    Box *box1 = new Box(3, 4, "Red");
    Caretaker *caretakerBox1 = new Caretaker(box1);
    caretakerBox1->addHistory();
    box1->setColor("Blue");
    caretakerBox1->addHistory();
    box1->setColor("Green");
    caretakerBox1->addHistory();
    box1->setColor("Black");

    cout<<"Now box1: "<<box1->getColor()<<endl;
    cout<<endl;
    caretakerBox1->showHistory();

    cout<<endl;
    caretakerBox1->undo();
    cout<<"Now box1: "<<box1->getColor()<<endl;

    delete box1;
    delete caretakerBox1;
}
```

Now box1: Black

History
Box color: Red
Box color: Blue
Box color: Green

Restore Black to Green [box3x4]
Now box1: Green

Restore Green to Blue [box3x4]
Now box1: Blue

Restore Blue to Red [box3x4]
Now box1: Red

Now box1: Red

=====

```

void clientLine() {
    Line *line = new Line(5, "Pink");
    Caretaker *caretakerLine = new Caretaker(line);
    caretakerLine->addHistory();
    line->setColor("Yellow");
    line->setWidth(6);
    caretakerLine->addHistory();
    line->setColor("white");
    line->setWidth(7);
    caretakerLine->addHistory();
    line->setColor("Orange");
    line->setWidth(8);

    line->show();
    cout<<endl;
    caretakerLine->showHistory();

    cout<<endl;
    caretakerLine->undo();
    line->show();

    cout<<endl;
    caretakerLine->undo();
    line->show();

    cout<<endl;
    caretakerLine->undo();
    line->show();

    cout<<endl;
    caretakerLine->undo();
    line->show();

    delete line;
    delete caretakerLine;
}

```

```

int main() {
    clientBox();
    cout<<"\n=====\\n" << endl;
    clientLine();

    return 0;
}

```

Now Line: Orange 8
 History
 Line width: 5 , color: Pink => Sun Apr 17 17:19:25 2022
 Line width: 6 , color: Yellow => Sun Apr 17 17:19:25 2022
 Line width: 7 , color: white => Sun Apr 17 17:19:25 2022
 Restore Orange to white and 8 to 7
 Now Line: white 7
 Restore white to Yellow and 7 to 6
 Now Line: Yellow 6
 Restore Yellow to Pink and 6 to 5
 Now Line: Pink 5
 Now Line: Pink 5

ควรใช้เมื่อไหร่

- เมื่อต้องการเก็บสถานะของอ็อบเจกต์ไว้ และสามารถเรียกคืนค่าได้ภายหลัง
- เมื่อการเข้าถึงแอทริบิวต์ของอ็อบเจกต์ตรงๆ ทำไม่ได้

ข้อดี ข้อเสีย

- ข้อดี
 - สามารถเก็บสถานะของอ็อบเจกต์ได้โดยไม่ทำลายกฎ **encapsulation**
 - สามารถใช้ **caretaker** ให้ดูแล **history** ของ **originator**
- ข้อเสีย
 - อาจใช้ **RAM** เยอะ ถ้ามีการเก็บข้อมูลบ่อย

การบันทึก

- มีระบบไฟอยู่ 1 เซ็ตประกอบด้วยไฟทั้งหมด 4 ดวง จึงเก็บสถานะของหลอดไฟทั้ง 4 เมื่อมีการเปลี่ยนแปลงค่าแต่ละครั้งในเซ็ต และสามารถ undo ได้ เช่น
 - ปิด ปิด ปิด ปิด
 - ปิด เปิด ปิด เปิด
 - เปิด ปิด ปิด เปิด

อ้างอิง

- <https://refactoring.guru/design-patterns/memento>

Catalog of Design Patterns

Behavioral Design Patterns

Observer

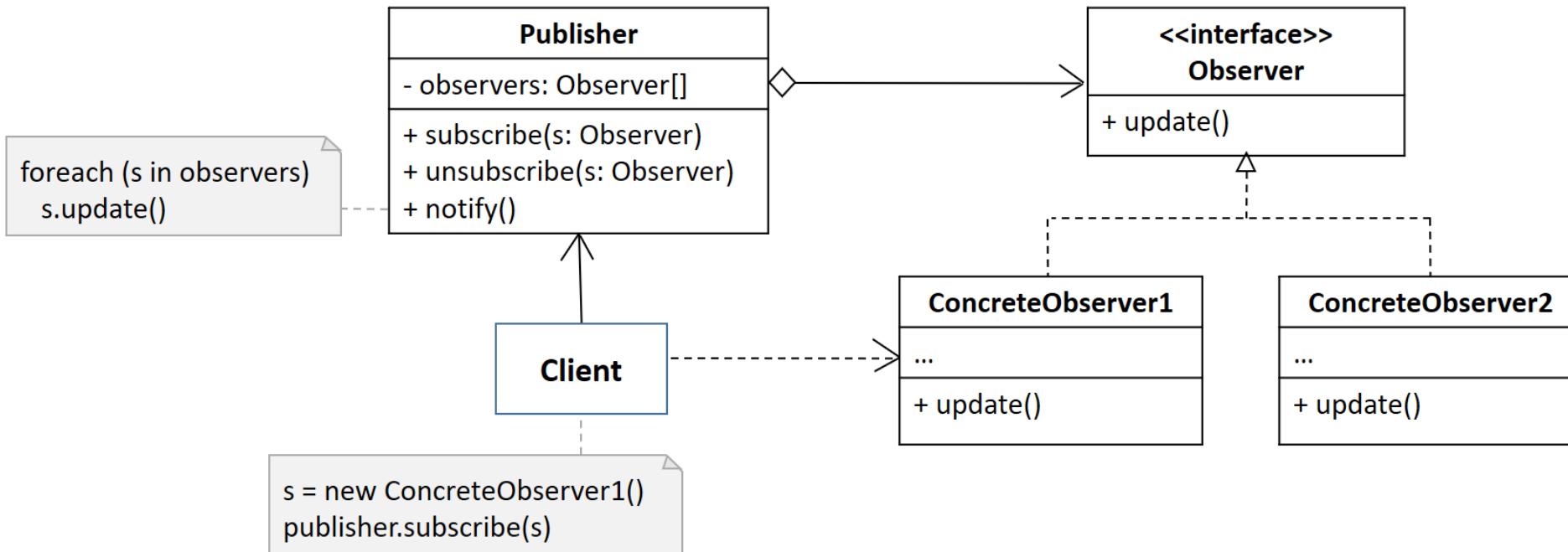
Observer หรือ Event-Subscriber หรือ Listener

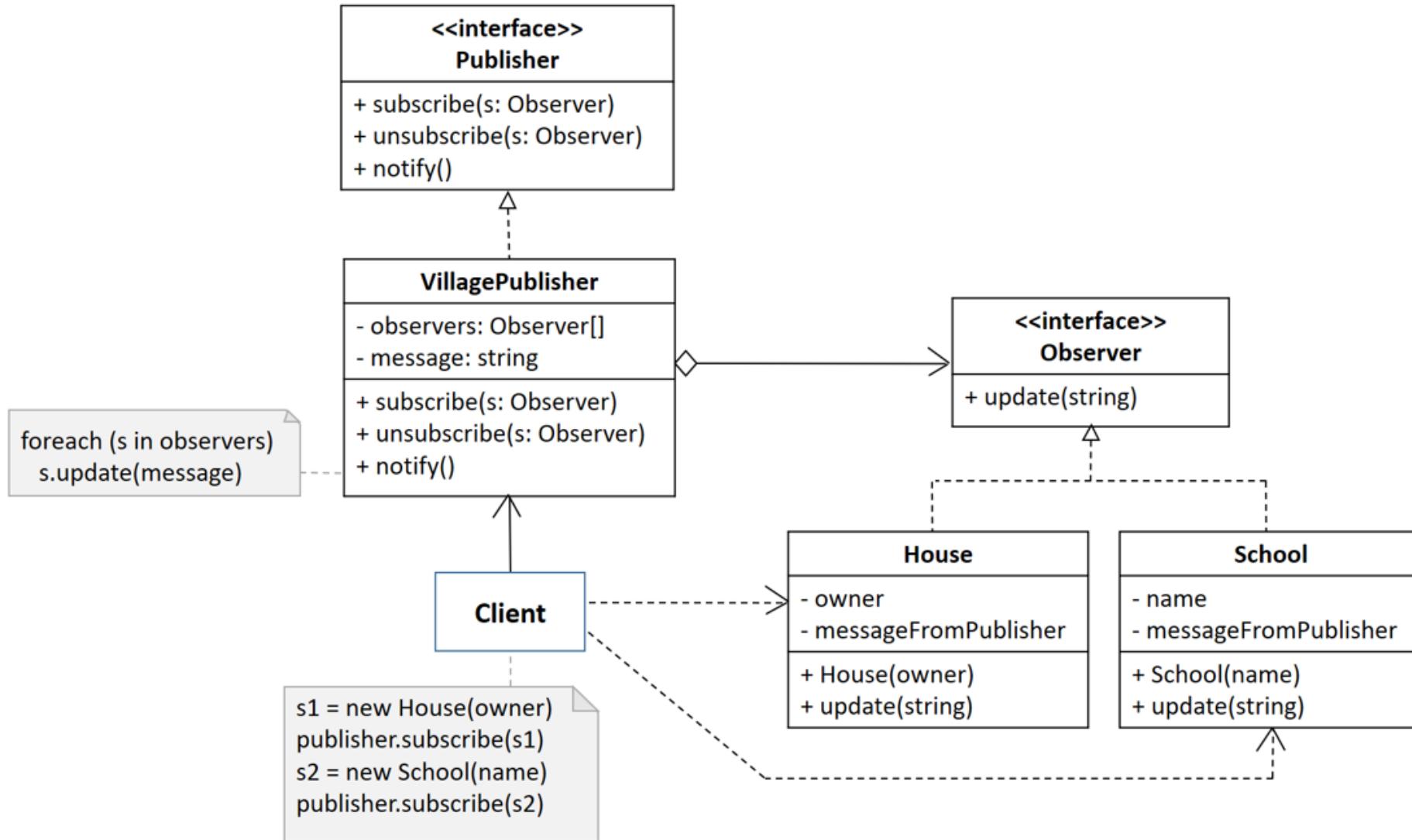
- สร้างกลไกการบอกรับสมาชิก เพื่อแจ้งเตือนสมาชิกถึงเหตุการณ์ที่เกิดขึ้นกับอ้อมูลเจ้าตัวที่สมาชิกกำลังให้ความสนใจ

Observer

- อ็อบเจกต์ไม่ต้องตามเช็คข้อมูลเอง
- สามารถส่งข้อมูลให้เฉพาะอ็อบเจกต์ที่ต้องการรับข้อมูลเท่านั้น
- คำศัพท์
 - อ็อบเจกต์ที่มี state ที่คนสนใจ จะเรียกอ็อบเจกต์นี้ว่า **Subject**
 - อ็อบเจกต์ที่ทำหน้าที่ส่งการแจ้งเตือนเมื่อ **Subject** มีการเปลี่ยนแปลง จะเรียกอ็อบเจกต์นี้ว่า **Publisher**
 - อ็อบเจกต์อื่นๆ ที่ต้องการติดตามการเปลี่ยนแปลง จะเรียกอ็อบเจกต์นี้ว่า **Subscriber** หรือ **Observer**

Structure





```
#include <iostream>
#include <list>
using namespace std;

class Observer {
public:
    virtual ~Observer() {};
    virtual void update(string messageFromPublisher) = 0;
};

class Publisher {
public:
    virtual ~Publisher() {};
    virtual void subscribe(Observer*) = 0;
    virtual void unsubscribe(Observer*) = 0;
    virtual void notify() = 0;
};
class VillagePublisher: public Publisher{
    list<Observer*> listObserver;
    string message;
public:
    void subscribe(Observer *s) {
        listObserver.push_back(s);
    }
    void unsubscribe(Observer *s) {
        listObserver.remove(s);
    }
    void notify() {
        for (list<Observer*>::iterator it = listObserver.begin(); it != listObserver.end(); it++)
            (*it)->update(message);
    }
    void setMessage(string message) {
        this->message = message;
        notify();
    }
};
```

```
class House: public Observer {
    string owner;
    string messageFromPublisher;
public:
    House(string a) {
        owner = a;
        messageFromPublisher = "";
    }
    void update(string message) {
        messageFromPublisher = message;
        showMessage();
    }
    void showMessage() {
        cout<<owner<<" house => "<<messageFromPublisher<<endl;
    }
};

class School: public Observer {
    string name;
    string messageFromPublisher;
public:
    School(string n) {
        name = n;
        messageFromPublisher = "";
    }
    void update(string message) {
        messageFromPublisher = message;
        showMessage();
    }
    void showMessage() {
        cout<<name<<" School => "<< messageFromPublisher<<endl;
    }
};
```

```
void client() {
    VillagePublisher *village = new VillagePublisher;
    House *maneeHouse = new House("Manee");
    village->subscribe(maneeHouse);
    House *somsriHouse = new House("Somsri");
    village->subscribe(somsriHouse);
    House *laddaHouse = new House("Ladda");
    village->subscribe(laddaHouse);
    School *lillySchool = new School("Lilly");
    village->subscribe(lillySchool);
    School *rosySchool = new School("Rosy");
    village->subscribe(rosySchool);
    House *jibHouse = new House("Jib");
    village->subscribe(jibHouse);

    village->setMessage("Village meeting tomorrow 9:30");
    cout<<endl;

    village->unsubscribe(maneeHouse);
    village->unsubscribe(laddaHouse);

    village->setMessage("Rain tomorrow morning");

    delete village;
    delete maneeHouse;
    delete somsriHouse;
    delete laddaHouse;
    delete lillySchool;
    delete rosySchool;
    delete jibHouse;
}
```

```
int main() {
    client();
    return 0;
}
```

Manee house => Village meeting tomorrow 9:30
Somsri house => Village meeting tomorrow 9:30
Ladda house => Village meeting tomorrow 9:30
Lilly School => Village meeting tomorrow 9:30
Rosy School => Village meeting tomorrow 9:30
Jib house => Village meeting tomorrow 9:30

Somsri house => Rain tomorrow morning
Lilly School => Rain tomorrow morning
Rosy School => Rain tomorrow morning
Jib house => Rain tomorrow morning

ควรใช้เมื่อไหร่

- เมื่ออ็อบเจกต์หนึ่งมีการเปลี่ยนแปลง อีกอ็อบเจกต์หนึ่งอาจต้องเปลี่ยนด้วย โดยไม่รู้ล่วงหน้าว่าเป็นอ็อบเจกต์ตัวไหน หรือ เป็นการเปลี่ยนแบบไอนามิก
- เมื่อมีอ็อบเจกต์ต้องการ **observe** อ็อบเจกต์ตัวอื่น แต่เป็นเพียงการ **observe** แค่ช่วงเวลาหนึ่ง หรือ เฉพาะกรณีใดกรณีหนึ่ง

ข้อดี ข้อเสีย

- ข้อดี
 - Open/Closed Principle
 - สามารถสร้างคลาส **observer** ตัวใหม่โดยไม่กระทบกับ โค้ดของ **publisher**
 - สามารถสร้างความสัมพันธ์ระหว่างอ็อปเจกต์ในช่วงเวลาอันใหม่
- ข้อเสีย
 - ไม่มีลำดับในการแจ้งเตือน

การบ้าน

- รับการแจ้งเตือนสภาพอากาศ เช่น อุณหภูมิสูงสุด อุณหภูมิต่ำสุด ค่า pm2.5 และสภาพอากาศ เช่น อากาศร้อน มีฝนฟ้าคะนอง ร้อยละ 10 ของพื้นที่ เป็นต้น
- แจ้งเตือนผ่าน มือถือ และ e-mail

อ้างอิง

- <https://refactoring.guru/design-patterns/observer>

Catalog of Design Patterns

Behavioral Design Patterns

State

State

- อ็อบเจกต์เปลี่ยนพฤติกรรมตามการเปลี่ยนสถานะของมัน

State

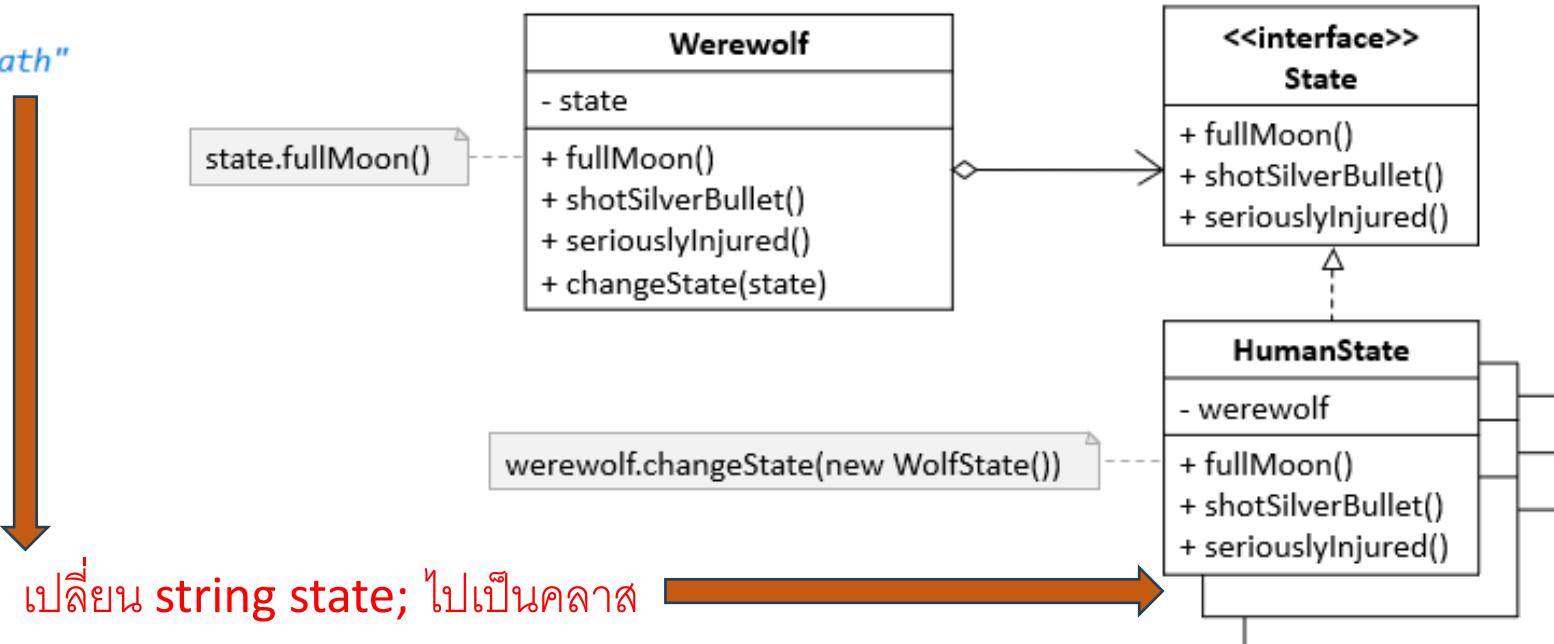
- แทนที่จะใช้ `if` หรือ `switch` ในการตรวจสอบ `state` ปัจจุบัน เพื่อให้อ็อบเจกต์ทำพฤติกรรมต่างๆ ตาม `state` นั้น ก็สร้างคลาสสำหรับแต่ละ `state` แทน รวมทั้งเงื่อนไขต่างๆ ก็อยู่ในแต่ละคลาสนี้ด้วย
- แต่ละ `state` อาจรับรู้ซึ้งกันและกัน และอาจมีการเปลี่ยนจาก `state` หนึ่งเป็นอีก `state` หนึ่ง

```
class Werewolf {
    string state; // "human", "wolf", "death"
public:
    Werewolf() {
        state="human";
    }
    void fullMoon() {
        if (state=="human")
            state = "wolf";
    }
    void shotSilverBullet() {
        state = "death";
    }
    void seriouslyInjured() {
        if (state=="wolf")
            state = "human";
    }
    string getState() {
        return state;
    }
};
```

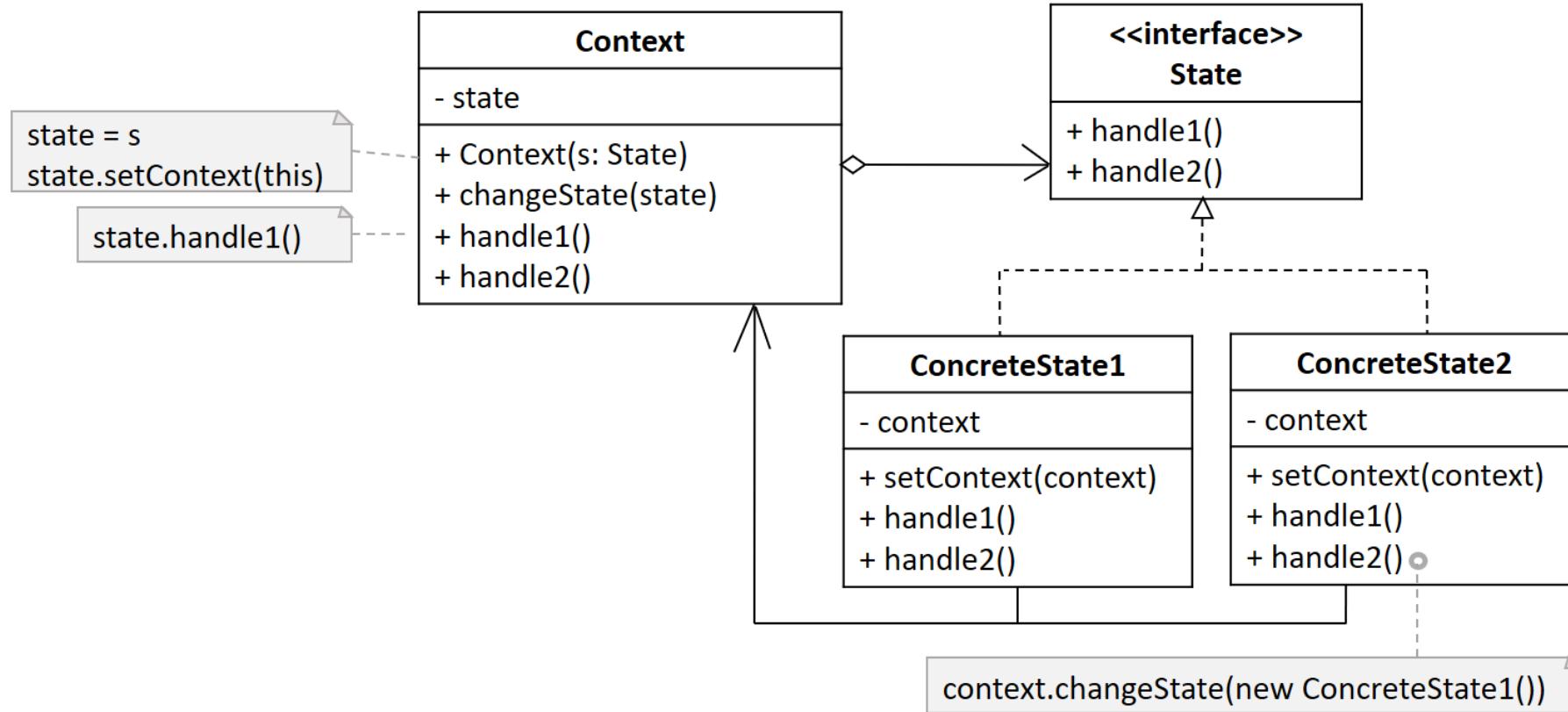
```

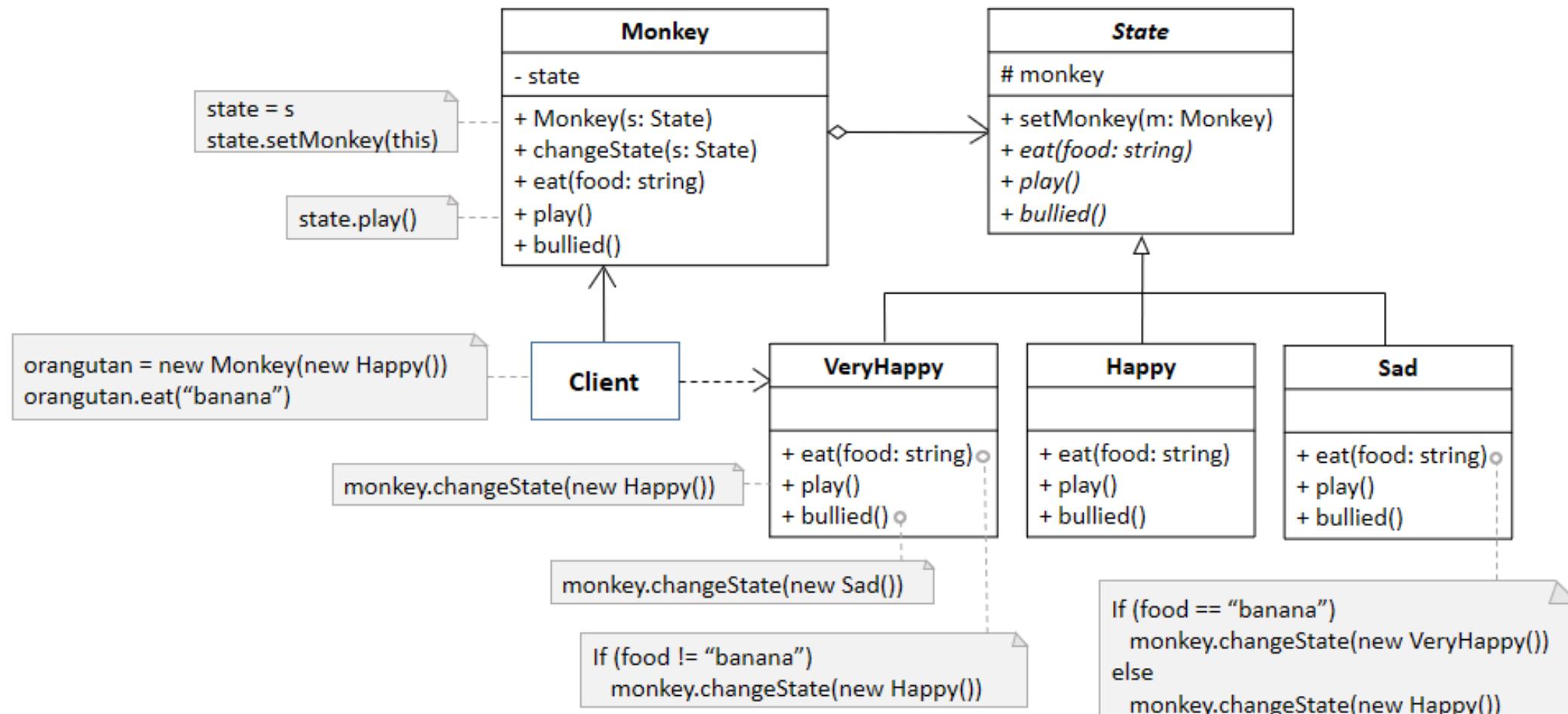
class Werewolf {
    string state; // "human", "wolf", "death"
public:
    Werewolf() {
        state="human";
    }
    void fullMoon() {
        if (state=="human")
            state = "wolf";
    }
    void shotSilverBullet() {
        state = "death";
    }
    void seriouslyInjured() {
        if (state=="wolf")
            state = "human";
    }
    string getState() {
        return state;
    }
};

```



Structure





```
#include <iostream>
using namespace std;

class Monkey;

class State {
protected:
    string name;
    Monkey *monkey;

public:
    virtual ~State() {}
    void setMonkey(Monkey *m) {
        monkey = m;
    }
    virtual void eat(string) = 0;
    virtual void play() = 0;
    virtual void bullied() = 0;
    string getName() {
        return name;
    }
};
```

```
class Monkey {
    State *state;
public:
    Monkey(State *s) {
        state = 0;
        changeState(s);
    }
    ~Monkey() {
        delete state;
    }
    void changeState(State *s) {
        cout<<"state: "<<s->getName()<<endl;
        if (state != 0)
            delete state;
        state = s;
        state->setMonkey(this);
    }
    void eat(string food) {
        state->eat(food);
    }
    void play() {
        state->play();
    }
    void bullied() {
        state->bullied();
    }
};
```

```
class VeryHappy: public State {
public:
    VeryHappy() {
        name="very happy";
    }
    void eat(string food);
    void play();
    void bullied();
};

class Happy: public State {
public:
    Happy() {
        name="happy";
    }
    void eat(string food) {
        if (food == "banana")
            monkey->changeState(new VeryHappy);
    }
    void play() {}
    void bullied();
};
```

```
class Sad: public State {
public:
    Sad() {
        name="sad";
    }
    void eat(string food) {
        if (food == "banana")
            monkey->changeState(new VeryHappy);
        else
            monkey->changeState(new Happy);
    }
    void play() {
        monkey->changeState(new Happy);
    }
    void bullied() {}
};

void VeryHappy::eat(string food) {
    if (food != "banana")
        monkey->changeState(new Happy);
}

void VeryHappy::play() {
    monkey->changeState(new Happy);
}

void VeryHappy::bullied() {
    monkey->changeState(new Sad);
}

void Happy::bullied() {
    monkey->changeState(new Sad);
}
```

```
void client() {
    Monkey *orangutan = new Monkey(new Happy);
    cout<<"\norangutan is eating banana" << endl;
    orangutan->eat("banana");

    cout<<"\norangutan is playing" << endl;
    orangutan->play();

    cout<<"\norangutan is bullied" << endl;
    orangutan->bullied();

    cout<<"\norangutan is eating leaf" << endl;
    orangutan->eat("leaf");

    cout<<"\norangutan is playing" << endl;
    orangutan->play();

    cout<<"\norangutan is eating coconut" << endl;
    orangutan->eat("coconut");

    delete orangutan;
}

int main() {
    client();
    return 0;
}
```

state: happy
orangutan is eating banana
state: very happy
orangutan is playing
state: happy
orangutan is bullied
state: sad
orangutan is eating leaf
state: happy
orangutan is playing
orangutan is eating coconut

ควรใช้เมื่อไหร่

- เมื่อต้องการแยกตัวกันโดยขึ้นอยู่กับสถานะปัจจุบัน และมี state เป็นจำนวนมาก และมีการแก้ไขโค้ดที่ขึ้นอยู่กับ state บ่อยๆ
 - การสร้าง state แยกออกจากเป็นคลาสทำให้ง่ายต่อการเพิ่มและเปลี่ยนแปลง state โดยไม่มีผลกระทบต่องานทำให้ลดค่าใช้จ่ายในการบำรุงรักษา
- เมื่อการปรับเปลี่ยนพฤติกรรมของคลาสขึ้นกับเงื่อนไขจำนวนมากและขึ้นกับค่าปัจจุบันของแอทริบิวต์ของคลาส
 - แปลงเงื่อนไขต่างๆ ไปเป็นเมธอด
- เมื่อมีการเขียนโค้ดซ้ำซ้อนกัน
 - โดยทำเป็น abstract base class

ข้อดี ข้อเสีย

- ข้อดี
 - Single Responsibility Principle
 - แยกแต่ละ state ออกเป็นแต่ละคลาส
 - Open/Closed Principle
 - สามารถสร้าง state ใหม่ได้โดยไม่กระทบกับของเดิม
 - โค้ดของ context เขียนง่ายขึ้น เพราะลดเงื่อนไขต่างๆ ลง
- ข้อเสีย
 - ถ้ามีแค่ไม่กี่ state หรือ state ไม่ค่อยมีการเปลี่ยนแปลง ก็ไม่ควรใช้ pattern นี้ เพราะว่ามันจะซับซ้อนมากเกินไป

การบ้าน

- Werewolf

อ้างอิง

- <https://refactoring.guru/design-patterns/state>

Catalog of Design Patterns

Behavioral Design Patterns

Strategy

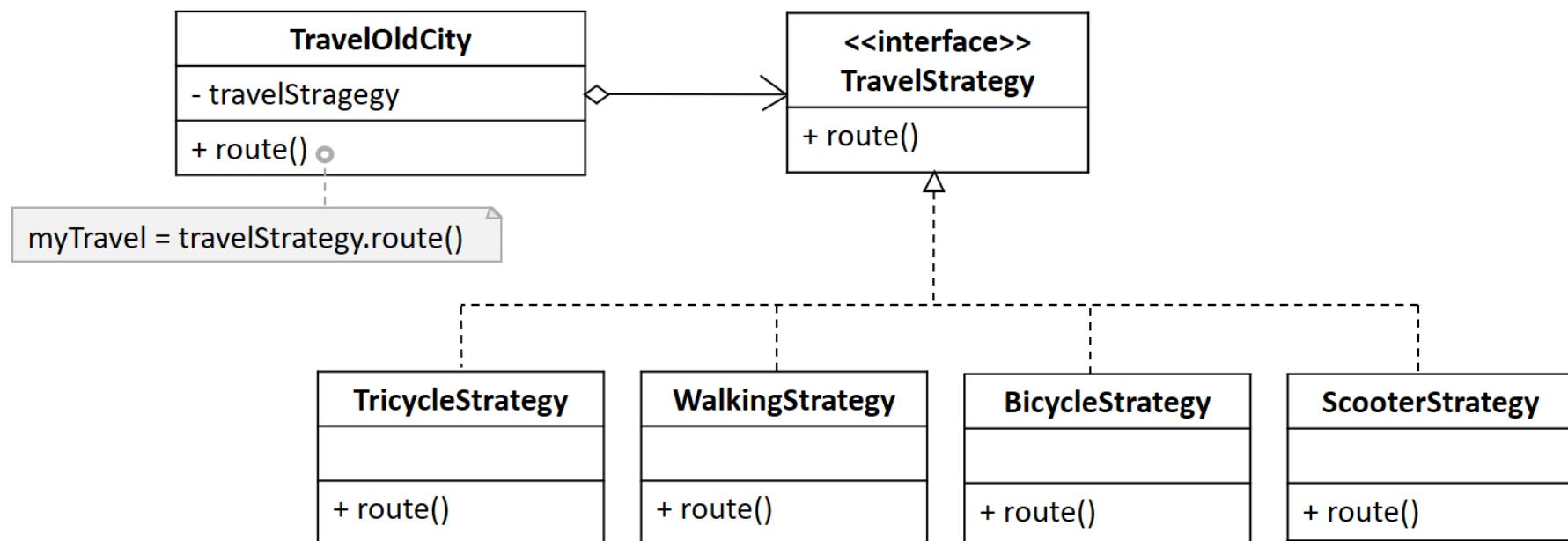
Strategy

- สร้างกลุ่มของอัลกอริทึมขึ้นมา โดยเอาอัลกอริทึมแต่ละตัวในกลุ่มนั้นไปไว้ในคลาสเดียวกัน และทำให้ออบเจกต์ของคลาสเหล่านั้นสามารถสลับแลกเปลี่ยนกันได้

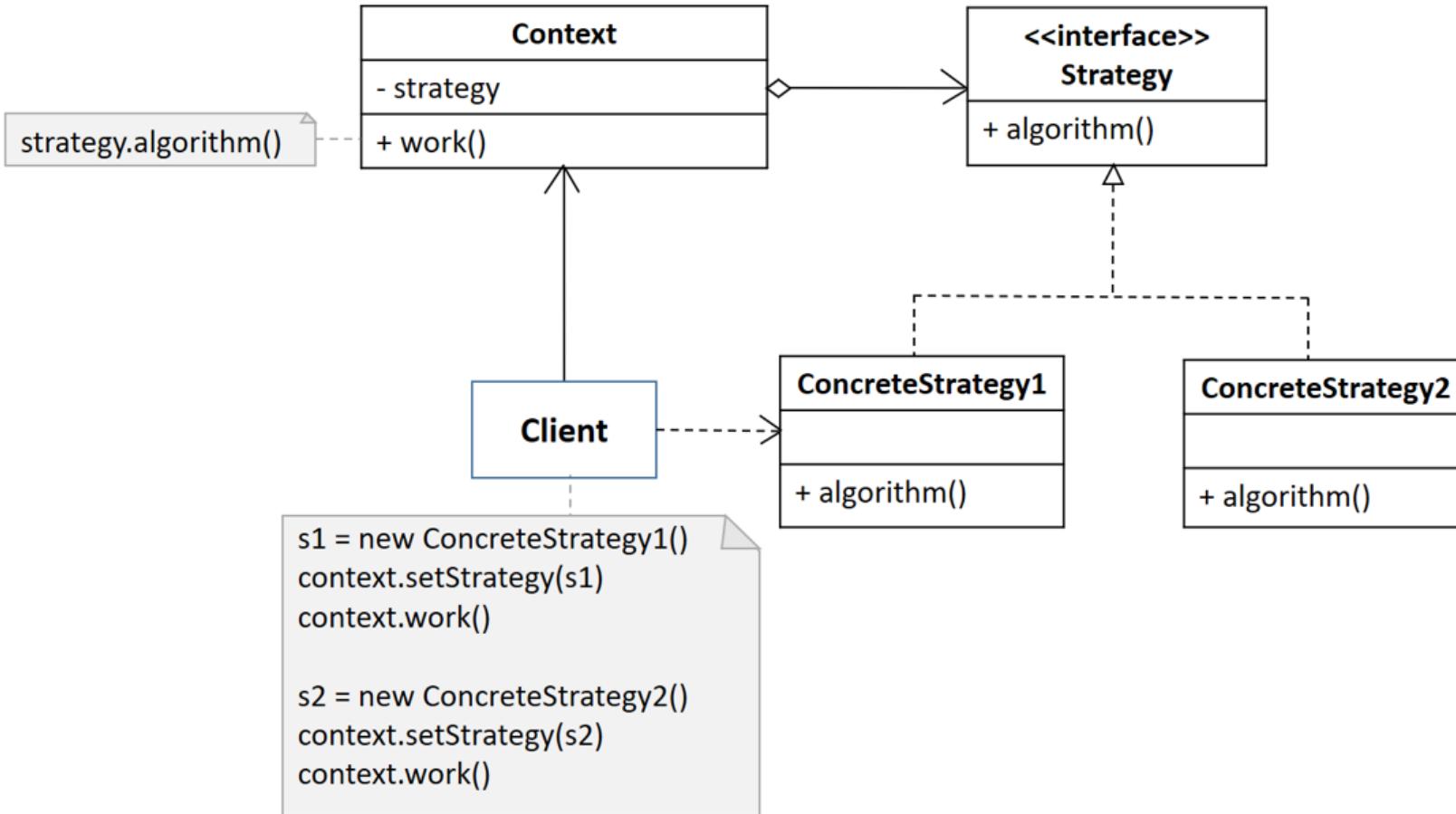
- อัลกอริทึม สำหรับการท่องเที่ยวในตัวเมืองเก่า
 - โดยรถสามล้อ
 - โดยการเดิน
 - โดยรถจักรยาน
 - โดยสกู๊ตเตอร์
- แยกอัลกอริทึมเหล่านี้ไปไว้ในคลาส เราเรียกคลาสเหล่านี้ว่า **strategy**
- เราเรียก **original class** ว่า **context** ซึ่งจะมี **reference** ซึ่งเปลี่ยนไปใน **strategy** เหล่านั้น

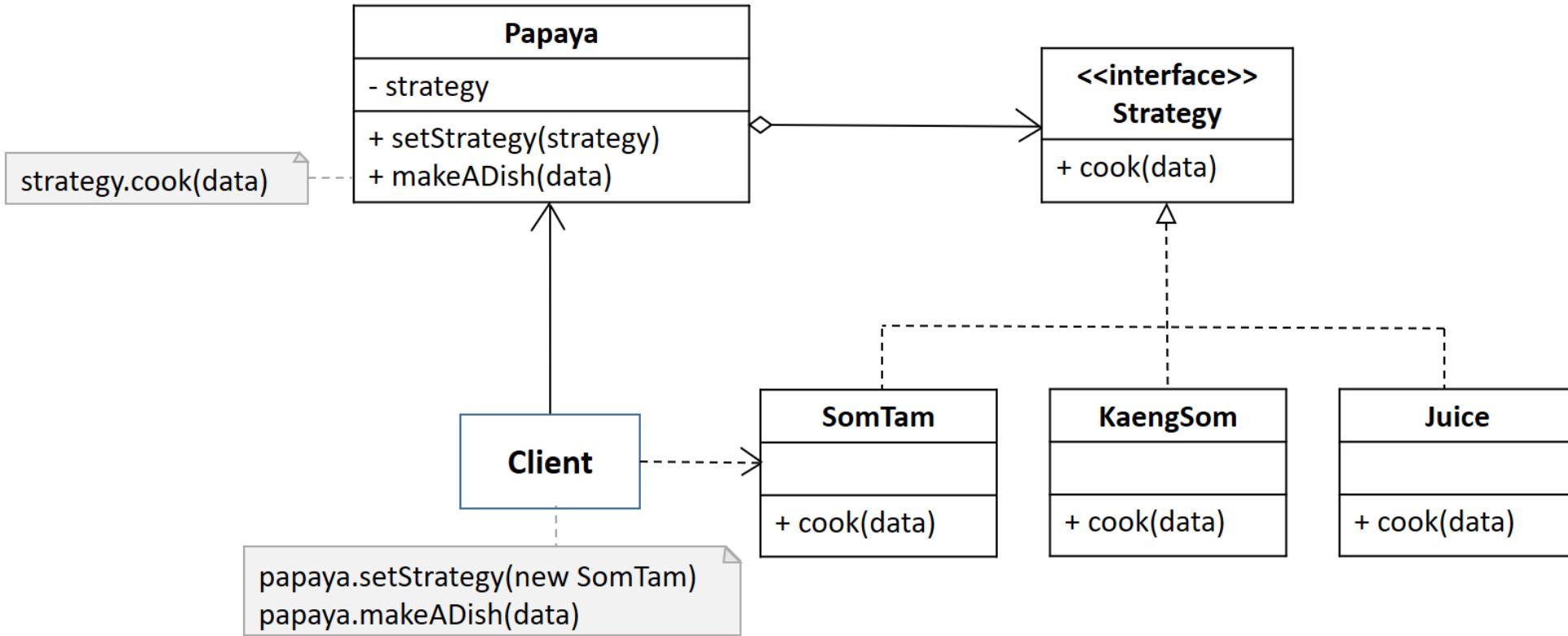
- อัลกอริทึม สำหรับการท่องเที่ยวในตัวเมืองเก่า

- โดยรถสามล้อ
- โดยการเดิน
- โดยรถจักรยาน
- โดยสกู๊ตเตอร์



Structure





```
#include <iostream>
using namespace std;

class Strategy {
public:
    virtual ~Strategy() {}
    virtual string cook(string) = 0;
};

class Papaya {
    Strategy *strategy;
public:
    Papaya(Strategy *s=0) {
        strategy = s;
    }
    ~Papaya() {
        delete strategy;
    }
    void setStrategy(Strategy *s) {
        delete strategy;
        strategy = s;
    }
    void makeADish(string data) {
        cout<<" "<<strategy->cook(data)<<endl;
    }
};

class KaengSom: public Strategy {
public:
    string cook(string data) {
        .....
        return "Kaeng Som "+data;
    }
};

class SomTam: public Strategy {
public:
    string cook(string data) {
        .....
        return "Som Tam "+data;
    }
};

class Juice: public Strategy {
public:
    string cook(string data) {
        .....
        return "Papaya Juice "+data;
    }
};
```

```
void client() {
    Papaya *p = new Papaya();
    cout<<"Papaya menu: "<<endl;
    p->setStrategy(new SomTam);
    p->makeADish("Thai");
    p->makeADish("Pu");
    p->makeADish("Pu Pla Ra");

    p->setStrategy(new KaengSom);
    p->makeADish("Kung");
    p->makeADish("Pla Chon");
    p->makeADish("South");

    p->setStrategy(new Juice);
    p->makeADish("with milk");
    p->makeADish("milk shake");

    delete p;
}

int main() {
    client();
    return 0;
}
```

Papaya menu:
Som Tam Thai
Som Tam Pu
Som Tam Pu Pla Ra
Kaeng Som Kung
Kaeng Som Pla Chon
Kaeng Som South
Papaya Juice with milk
Papaya Juice milk shake

ควรใช้เมื่อไหร่

- เมื่อ อ็อบเจกต์ต้องการใช้อัลกอริทึมหลายแบบและสามารถสลับการใช้งานอัลกอริทึมได้ในช่วงเวลา r น ใหม่
- เมื่อมีคลาสหลายคลาสที่แตกต่างกันแค่อัลกอริทึมที่ใช้
 - ลดความซ้ำซ้อนของโค้ด
- แยก **business logic** ของคลาสออกจากส่วนของการอิมเพลเม้นต์รายละเอียดของอัลกอริทึม

ข้อดี ข้อเสีย

- ข้อดี
 - สามารถ слับการเรียกใช้อัลกอริทึมได้ในช่วงเวลา ran ใหม่
 - แยกส่วนของการอิมเพลเม้นต์รายละเอียดของอัลกอริทึมออกจากโครงสร้างที่เรียกใช้อัลกอริทึมนั้น
 - ใช้ composition แทนการใช้ inheritance
 - Open/Closed Principle
 - สามารถสร้าง strategy ใหม่ได้โดยไม่ต้องแก้ไข context
- ข้อเสีย
 - โค้ดมีความซับซ้อน
 - จะต้องรู้ถึงความแตกต่างระหว่าง strategy ต่างๆ เพื่อจะได้เรียกใช้ได้อย่างเหมาะสม

การบ้าน

- ปลา มีวิธีในการทำอาหารหลากหลายแบบ
 - ทอดน้ำปลา
 - นึ่งซีอิ๊ว
 - ย่างเกลือ
 - ลุยกะทูล
 - ลาบ
 - เค็ก
- ให้ระบุชนิดของปลาที่ใช้ทำอาหารด้วย เช่น ปลากระพง ปลานิล ปลาช่อน ปลาทับทิม เป็นต้น

อ้างอิง

- <https://refactoring.guru/design-patterns/strategy>

Catalog of Design Patterns

Behavioral Design Patterns

Template Method

Template Method

- สร้างโครงสร้างของอัลกอริทึมไว้ในคลาสแม่ และให้คลาสลูกสามารถoverride บางชิ้นตอนของอัลกอริทึมได้โดยไม่กระทบกับโครงสร้างหลัก

- เกี่ยนอัลกอริทึมให้เป็นลำดับขั้นตอนชัดเจน
 - แปลงแต่ละขั้นตอนนั้นให้เป็นแต่ละเมธอด ซึ่งแต่ละเมธอดอาจทำเป็น abstract ก็ได้หรือทำเป็น default ไว้ก็ได้
 - สร้าง template method ขึ้นมา โดยให้ไปเรียกใช้เมธอดที่สร้างไว้แล้วนั้นตามลำดับของอัลกอริทึม
-
- ปั๊งยาง (หนุปั๊ง ไก่ปั๊ง ปลาหมึกปั๊ง)
 - ทำความสะอาดเนื้อสัตว์
 - หันเนื้อสัตว์
 - ใส่เครื่องปูรุ่งหมักเนื้อสัตว์
 - ปั๊ง

- ปิ้งย่าง (หมูปิ้ง ไก่ปิ้ง ปลาหมึกปิ้ง)

- ทำความสะอาดเนื้อสัตว์
- หั่นเนื้อสัตว์
- ใส่เครื่องปุงหมักเนื้อสัตว์
- ย่าง

- Steps

- abstract steps

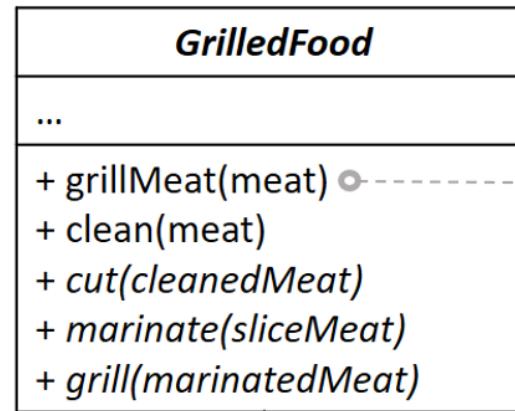
- ต้องอิมพลิเมนท์ทุกคลาส

- optional steps

- สามารถไม่เรียกได้

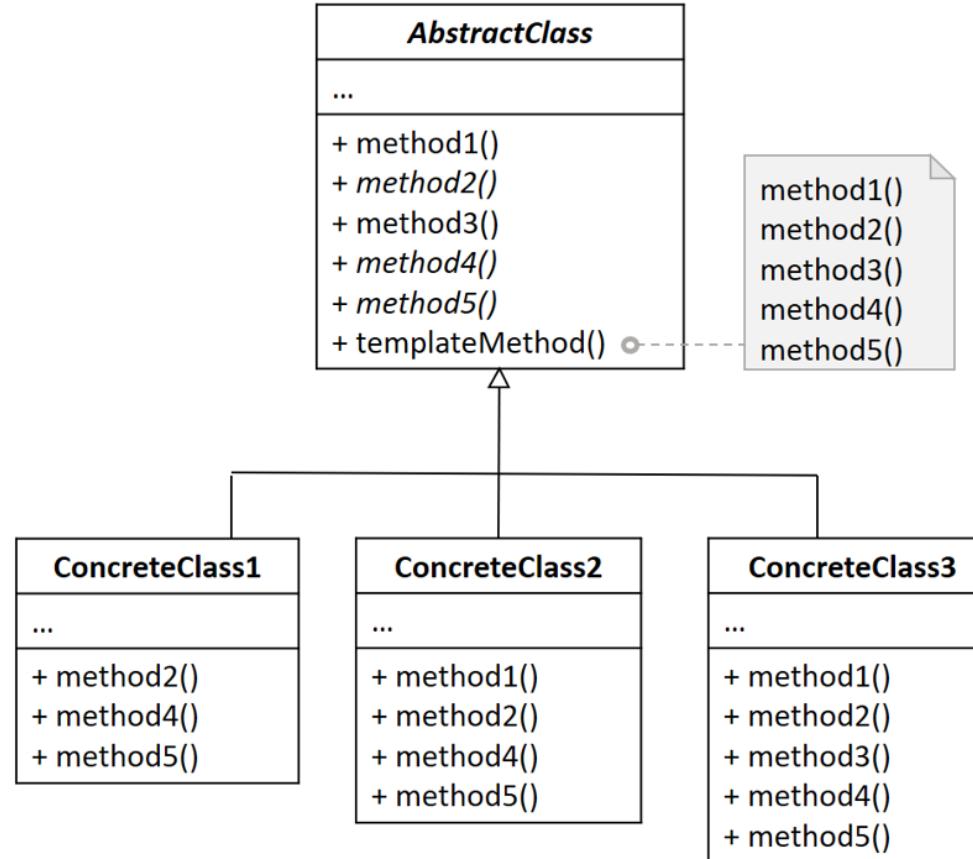
- hook เป็น optional step ที่ไม่ได้ให้ทำอะไร (empty body)

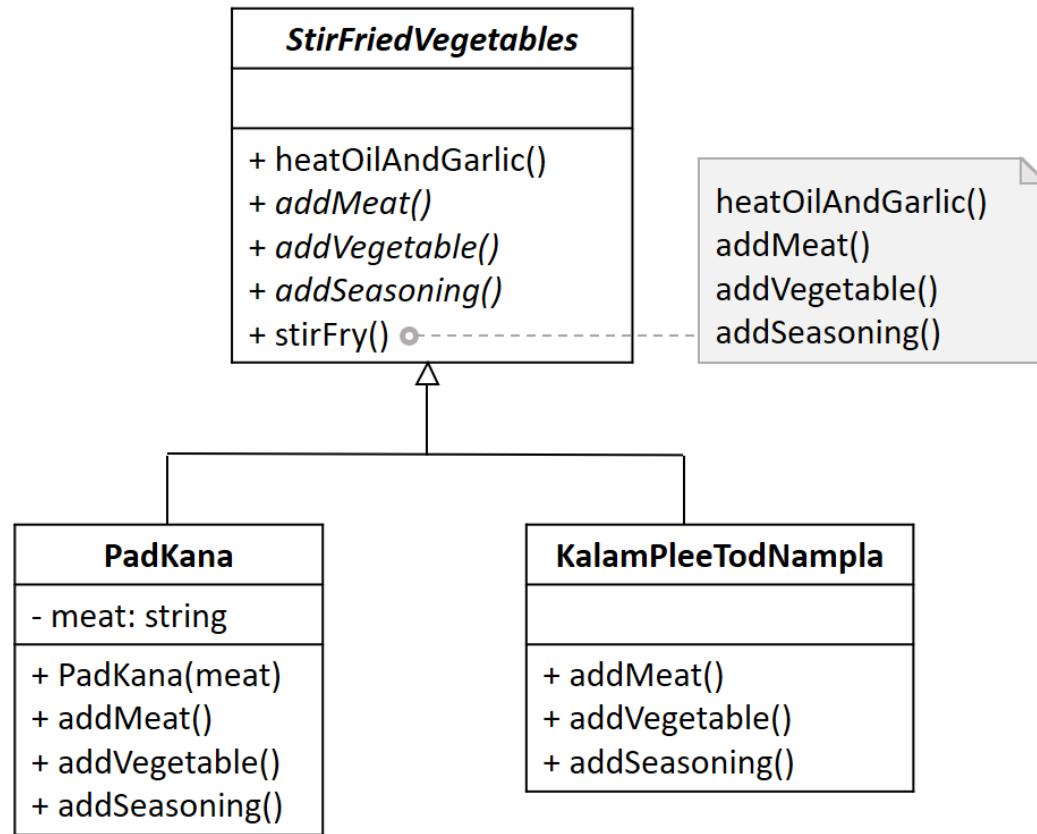
- มักจะวางไว้ก่อนและหลัง step ที่สำคัญๆ เพื่อให้คลาสลูกสามารถเพิ่มเติมอะไรได้



`cleanedMeat = clean(meat)`
`slicedMeat = cut(cleanedMeat)`
`marinatedMeat = marinate(sliceMeat)`
`grilledMeat = grill(marinatedMeat)`

Structure





```
#include <iostream>
using namespace std;

class StirFriedVegetables {
public:
    void stirFry() {
        heatOilAndGarlic();
        addMeat();
        addVegetable();
        addSeasoning();
    }

protected:
    void heatOilAndGarlic() {
        cout<<"  heat oil and minced garlic"<<endl;
    }
    virtual void addMeat() = 0;
    virtual void addVegetable() = 0;
    virtual void addSeasoning() = 0;
};
```

```
class PadKana: public StirFriedVegetables {
    string meat;
public:
    PadKana(string m) {
        meat = m;
    }
    void setMeat(string m) {
        meat = m;
    }
protected:
    void addMeat() {
        cout<<"  add " << meat << endl;
    }
    void addVegetable() {
        cout<<"  add Kana" << endl;
    }
    void addSeasoning() {
        cout<<"  add seasoning" << endl;
    }
};
```

```
class KalamPleeTodNampla: public StirFriedVegetables {
protected:
    void addMeat() {}
    void addVegetable() {
        cout<<"  add kalam plee"<<endl;
    }
    void addSeasoning() {
        cout<<"  add nampla"<<endl;
    }
};

void client(StirFriedVegetables *v) {
    v->stirFry();
}

int main() {
    PadKana *kana = new PadKana("moo krob");
    cout<<"Kana Moo Krob"<<endl;
    client(kana);
    cout<<endl;
    cout<<"Kana Pla Salid"<<endl;
    kana->setMeat("pla salid");
    client(kana);
    cout<<endl;

    cout<<"Kalam Plee Tod Nampla"<<endl;
    KalamPleeTodNampla *kalamPlee = new KalamPleeTodNampla;
    client(kalamPlee);

    delete kalamPlee;
    delete kana;

    return 0;
}
```

Kana Moo Krob
heat oil and minced garlic
add moo krob
add Kana
add seasoning

Kana Pla Salid
heat oil and minced garlic
add pla salid
add Kana
add seasoning

Kalam Plee Tod Nampla
heat oil and minced garlic
add kalam plee
add nampla

ควรใช้เมื่อไหร่

- เมื่อต้องการให้คลาสลูกสามารถแก้ไขบางข้อตอนของอัลกอริทึมได้
- เมื่อมีหลายคลาสที่มีการใช้อัลกอริทึมที่เกือบจะเหมือนกันโดยแตกต่างกันเพียงเล็กน้อยเท่านั้น

ข้อดี ข้อเสีย

- ข้อดี
 - คลาสลูกสามารถอิเ沃ร์ไรด์ได้แค่บางส่วนของอัลกอริทึม ทำให้มีผลกระทบกับส่วนอื่นของอัลกอริทึมน้อย
 - สามารถนำได้ด้วยตัวเอง ไปใช้ในคลาสแม่
- ข้อเสีย
 - การทำงานอาจถูกจำกัดโดยโครงสร้างของอัลกอริทึม
 - ละเมิดกฎ **Liskov Substitution Principle** (อ็อบเจกต์ของคลาสลูกต้องสามารถแทนที่อ็อบเจกต์ของคลาสแม่ได้)
 - ยิ่งมีขั้นตอนมาก ยิ่งดูแลจัดการ **template method** ได้ยาก

ກາຣບ້ານ

- ກ່ວຍເຕື່ອງ[†]
 - ລວກເສັ້ນ (ເລັກ ໃຫຍ່ງ ໄມ້ ບະໜີ)
 - ລວກຜັກ
 - ໄສເນື້ອທີ່ປຽງໄວແລ້ວ (ໜູ້ຕຸ້ນ ເນື້ອຕຸ້ນ ເປີດຕຸ້ນ ໄກ່ຕຸ້ນ)
 - ໄສນໍ້າຫຼຸບ
 - ໄສເຄື່ອງປຽງ

อ้างอิง

- <https://refactoring.guru/design-patterns/template-method>

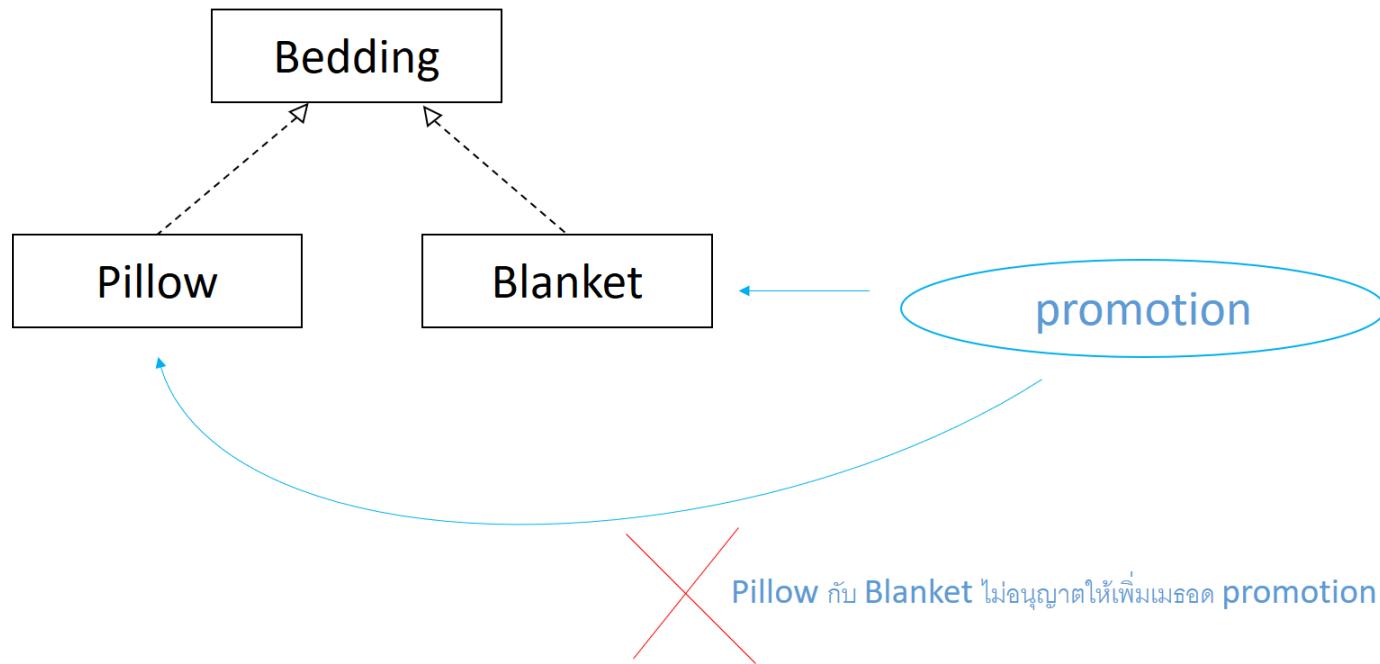
Catalog of Design Patterns

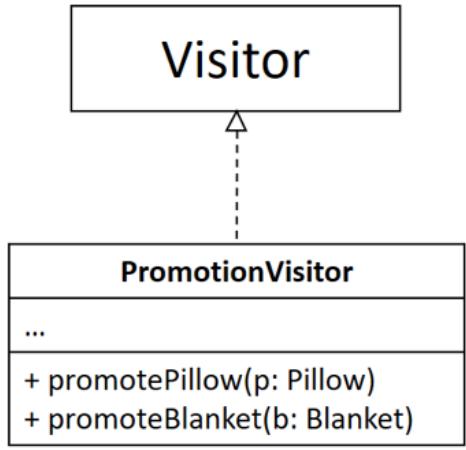
Behavioral Design Patterns

Visitor

Visitor

- แยกอัลกอริทึมออกจากอัลกอริทึมที่ใช้อัลกอริทึมนั้น

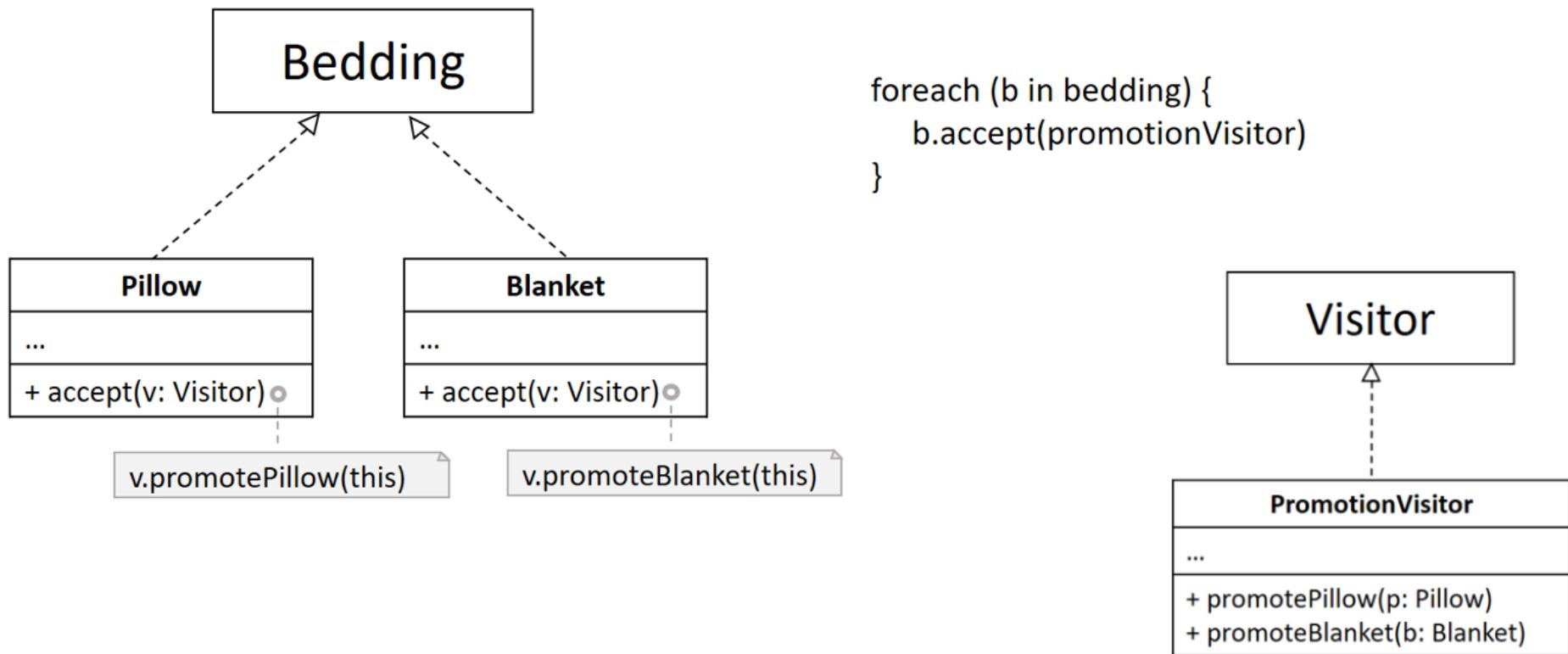




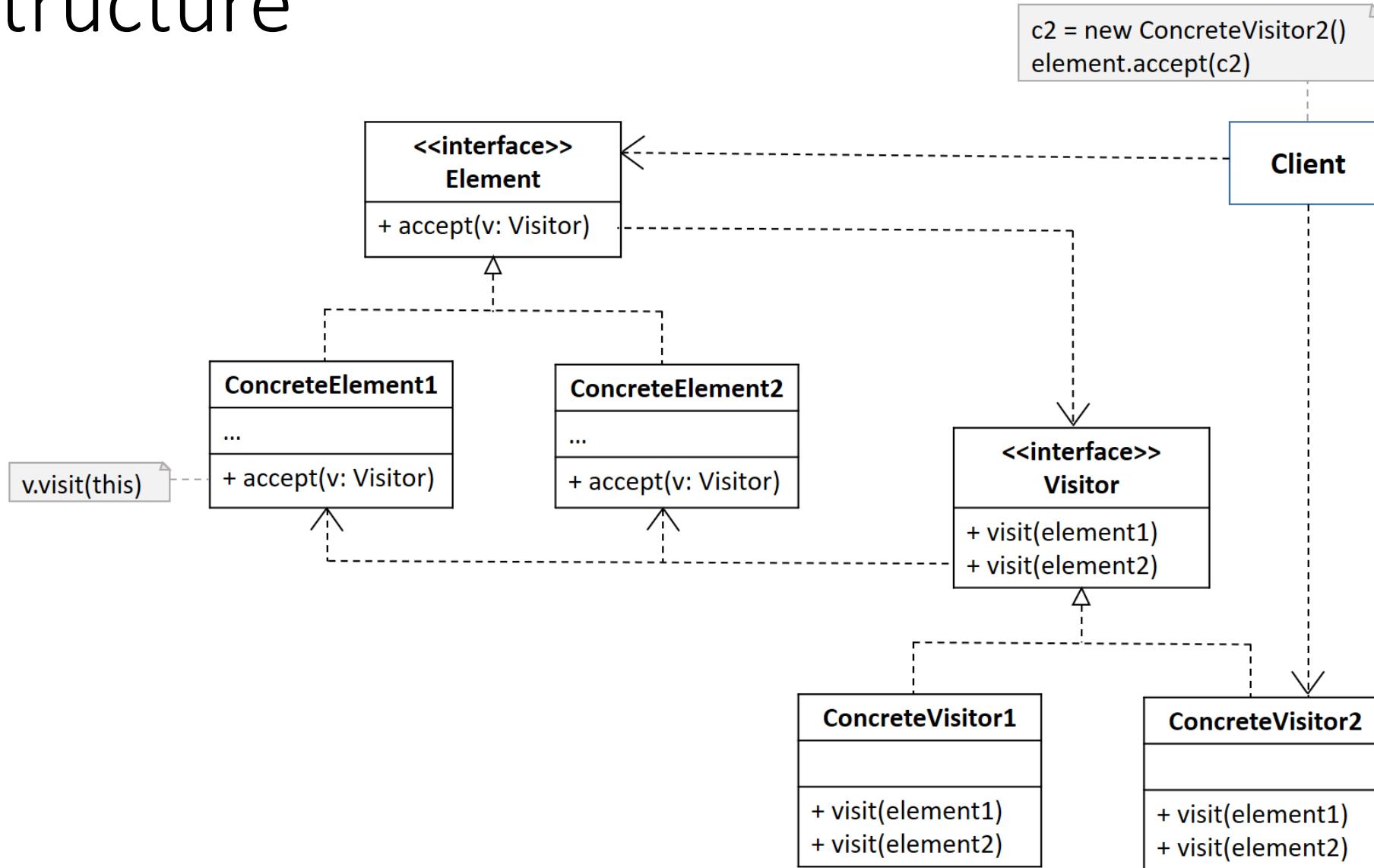
นำ promotion ไปไว้ในคลาสใหม่
แต่ก็ทำ polymorphism ไม่ได้
ต้องเช็คทีละคลาส

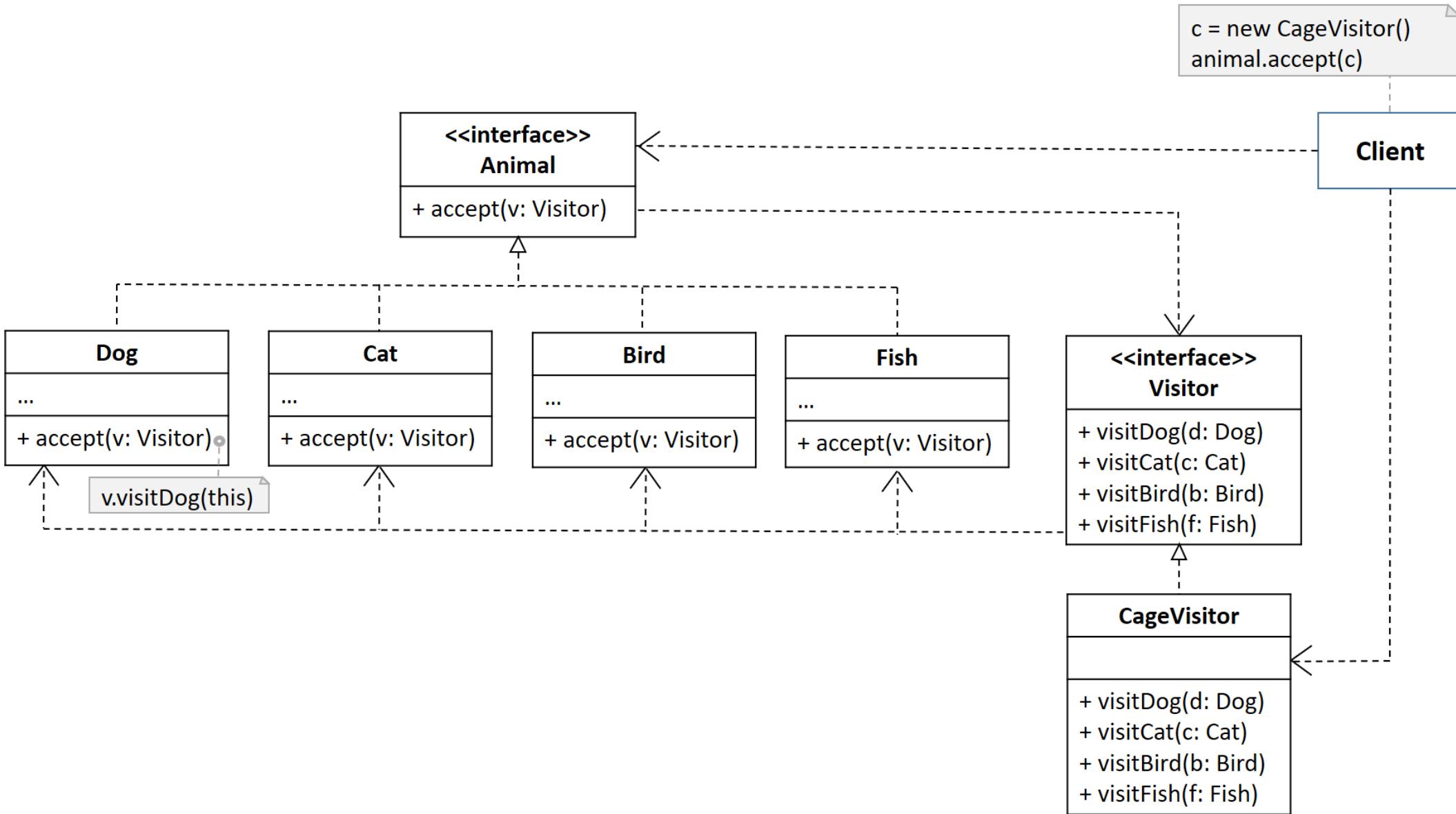
```
foreach (b in bedding) {
    if (b instanceof Pillow)
        promotionVisitor.promotePillow((Pillow) b)
    if (b instanceof Blanket)
        promotionVisitor.promoteBlanket((Blanket) b)
}
```

แทนที่จะให้ client เป็นคนเลือกใช้ method
ก็ให้ออปเจกต์ที่ถูกส่งเป็นอาร์กูเมนต์ไปยัง visitor เป็นคนทำแทน



Structure





```
#include <list>
#include <iostream>
using namespace std;

class Dog;
class Cat;
class Bird;
class Fish;

class Visitor {
public:
    virtual void visitDog(Dog*) = 0;
    virtual void visitCat(Cat*) = 0;
    virtual void visitBird(Bird*) = 0;
    virtual void visitFish(Fish*) = 0;
};

class Animal {
public:
    virtual ~Animal() {}
    virtual void accept(Visitor *v) = 0;
};

class Dog: public Animal {
    string breed;
    int age;
public:
    Dog(string b, int a) {
        breed = b;
        age = a;
    }
    void accept(Visitor *v) {
        v->visitDog(this);
    }
    string getBreed() {
        return breed;
    }
    int getAge() {
        return age;
    }
};

class Cat: public Animal {
    int age;
public:
    Cat(int a) {
        age = a;
    }
    void accept(Visitor *v) {
        v->visitCat(this);
    }
    int getAge() {
        return age;
    }
};
```

```
class Bird: public Animal {
    string breed;
public:
    Bird(string b) {
        breed = b;
    }
    void accept(Visitor *v) {
        v->visitBird(this);
    }
    string getBreed() {
        return breed;
    }
};

class Fish: public Animal {
    string breed;
public:
    Fish(string b) {
        breed = b;
    }
    void accept(Visitor *v) {
        v->visitFish(this);
    }
    string getBreed() {
        return breed;
    }
};
```

```
class CageVisitor: public Visitor {
public:
    void visitDog(Dog *d) {
        cout<<" build a cage for a "<<d->getAge()<<"-year-old "<<d->getBreed()<<" dog"<<endl;
    }
    void visitCat(Cat *c) {
        cout<<" build a cage for a "<<c->getAge()<<"-year-old cat"<<endl;
    }
    void visitBird(Bird *b) {
        cout<<" build a cage for "<<b->getBreed()<<endl;
    }
    void visitFish(Fish *f) {
        cout<<" build a basin for the "<<f->getBreed()<<endl;
    }
};

void client(list<Animal *> animals, Visitor *v) {
    cout<<"Using CageVisitor"<<endl;
    for (list<Animal*>::iterator it = animals.begin(); it != animals.end(); it++) {
        (*it)->accept(v);
    }
}
```

```
for (Animal* a : animals)
    a->accept(v);
```

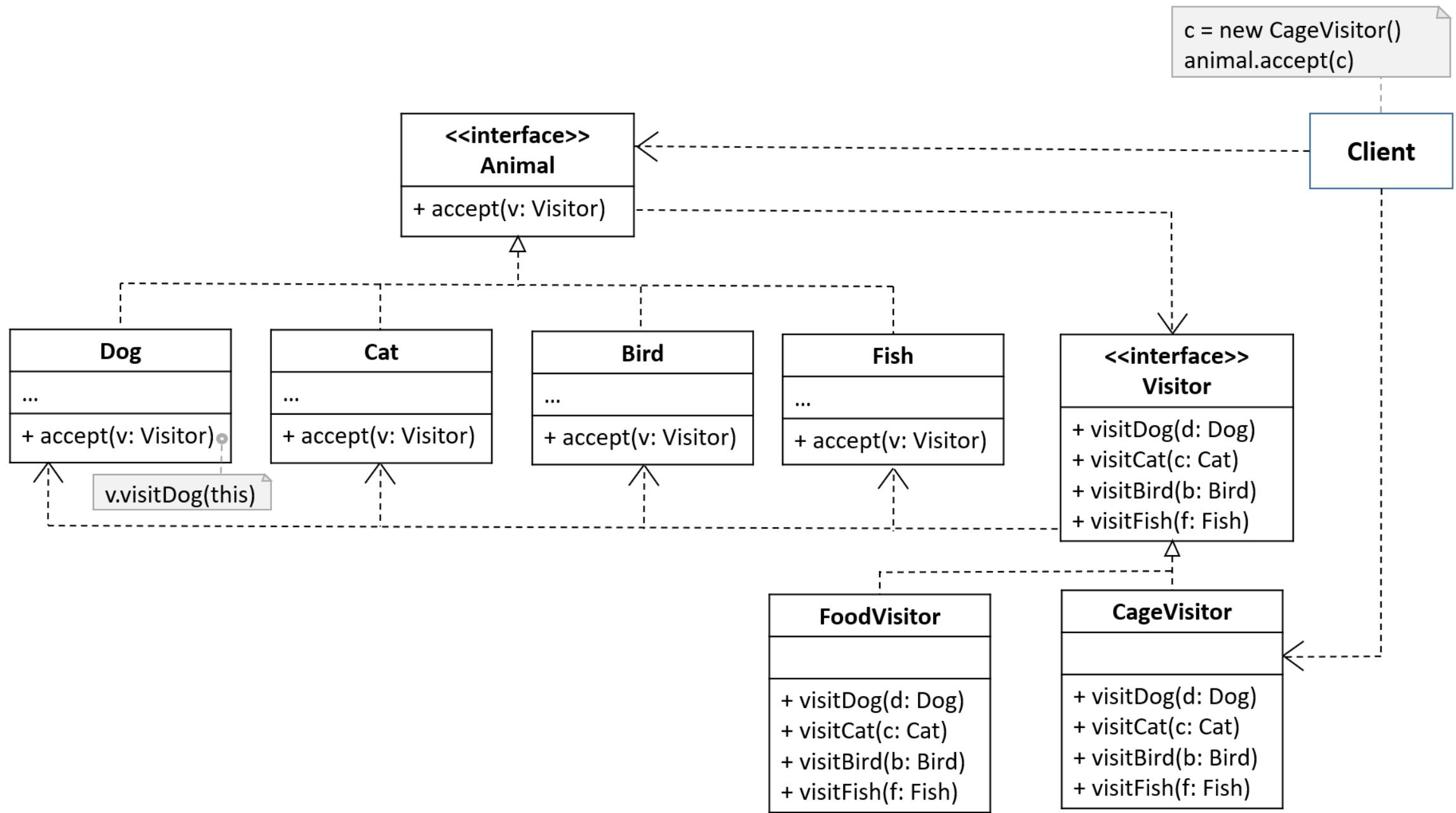
```
int main() {
    list<Animal*> animals;
    animals.push_back(new Dog("labrador", 3));
    animals.push_back(new Cat(2));
    animals.push_back(new Bird("parrots"));
    animals.push_back(new Fish("carp fish"));

    CageVisitor *cageVisitor = new CageVisitor;
    client(animals, cageVisitor);

    for (list<Animal*>::iterator it = animals.begin(); it != animals.end(); it++) {
        delete (*it);
    }
    delete cageVisitor;

    return 0;
}
```

```
Using CageVisitor
    build a cage for a 3-year-old labrador dog
    build a cage for a 2-year-old cat
    build a cage for parrots
    build a basin for the carp fish
```



ควรใช้เมื่อไหร่

- เมื่อต้องการดำเนินการกับ **element** ทุกตัว ในกลุ่มที่มีโครงสร้างที่ซับซ้อน
- เมื่อต้องจัดการกับงานที่ไม่ใช่งานหลัก
 - งานที่ไม่ใช่งานหลักของคลาสก็เอาไปไว้ใน **visitor**
- เมื่องานนั้นหมายความกับคลาสแค่บางคลาสที่อยู่ในลำดับชั้น
 - แยกงานนั้นออกมาระบบที่ **visitor** และให้เฉพาะคลาสที่ **accept** เท่านั้นที่จะใช้งานได้

ข้อดี ข้อเสีย

- ข้อดี
 - Open/Closed Principle
 - สามารถเพิ่มงานใหม่ได้โดยไม่ต้องแกะไขคลาส
 - Single Responsibility Principle
 - เก็บงานที่เหมือนกันไว้ในคลาสเดียวกัน
 - สามารถใช้ร่วมข้อมูลเมื่อต้องทำงานกับออบเจกต์ที่แตกต่างกัน
- ข้อเสีย
 - ต้องปรับปรุง visitor ทุกครั้งที่มีการเพิ่มหรือลบคลาสออกจากลำดับชั้นของ element
 - อาจเข้าถึงข้อมูล private ไม่ได้

การบ้าน

- คลาสหลัก TV, Fridge, Fan, WashingMachine
- Visitor
 - แสดงค่าข้อมูลเหล่านี้
 - TV ให้บอกว่าเป็น smart TV หรือไม่
 - Fridge ให้บอกว่ามีขนาดกี่คิว
 - Fan ให้บอกว่าเป็นพัดลมตั้งตีะหรือพัดลมตั้งพื้น
 - WashingMachine ให้บอกว่าเป็นแบบฝาน้ำหรือฝาบน

อ้างอิง

- <https://refactoring.guru/design-patterns/visitor>