# Introduction to Design Patterns

## Head First Design Patterns


Prepared
By Wiwat V.

# Agenda

- What and Why Design Patterns!
- Strategy
- Observer
- Singleton
- Decorator
- Façade
- Adapter

# What is Design Pattern

- Gang of Four 1995 (GoF) คณะบุคคลสำคัญที่มีคนกล่าวถึงมาก

  “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

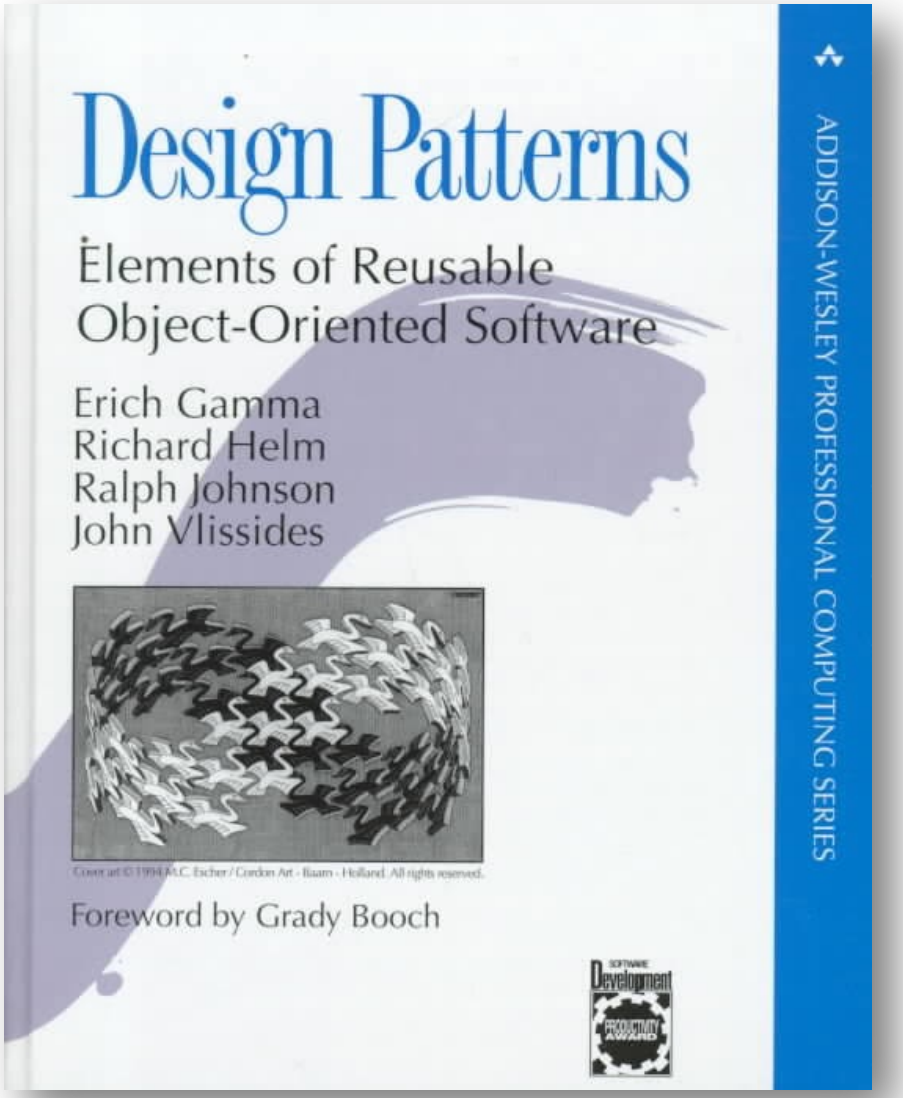
- Wikipedia definition ง่ายๆ

  “a design pattern is a general repeatable solution to a commonly occurring problem in software design”

# Gang of Four (GoF)

Erich Gamma

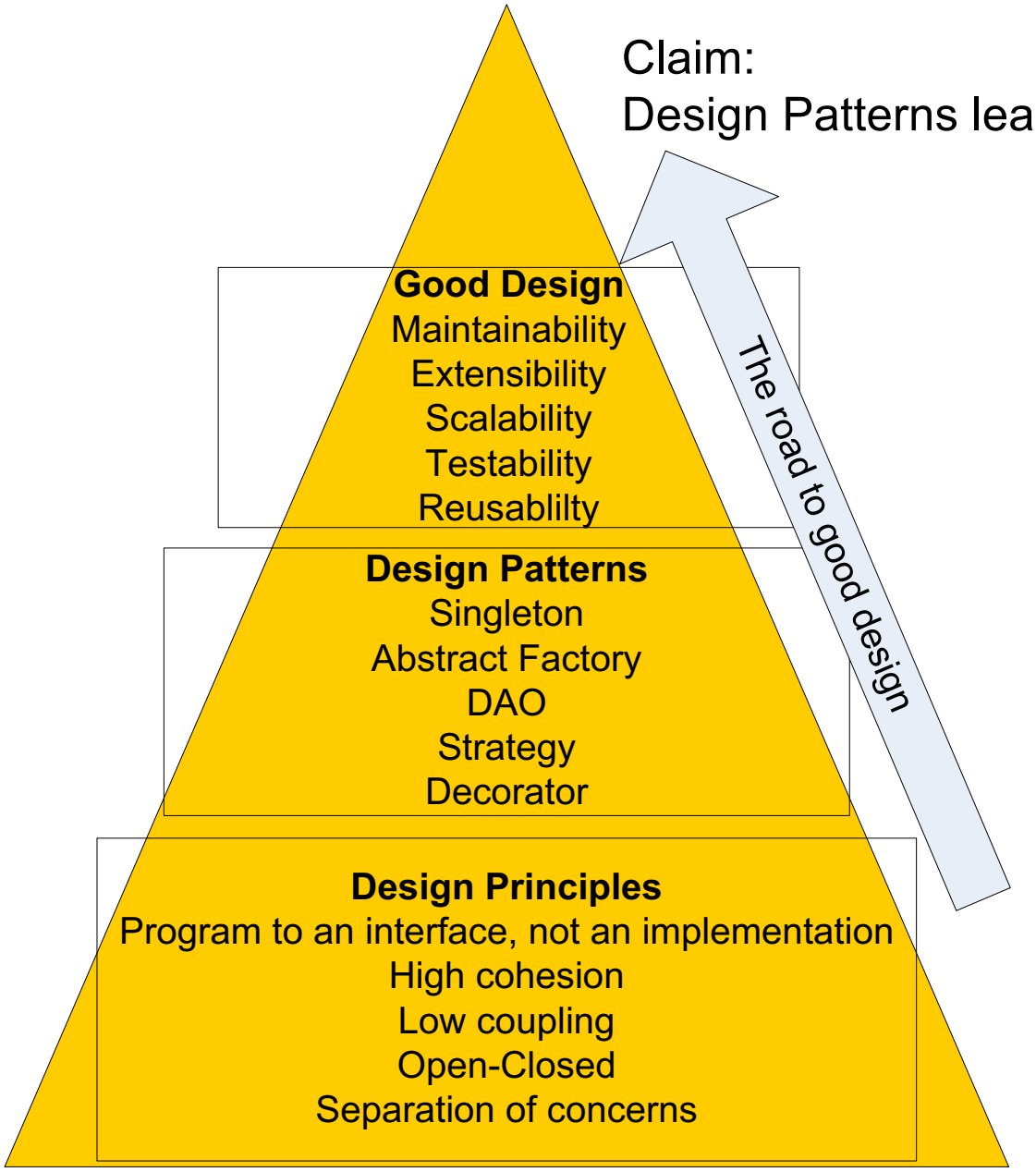
Ralph Johnson




Richard Helm

John Vlissides

# Why use Design Patterns?




Claim:
Design Patterns lead to good design

Anti-Patterns?

# Why use Design Patterns?

- Design Objectives
  - Good Design (the “ilities”)
    - High readability and maintainability
    - High extensibility
    - High scalability
    - High testability
    - High reusability

เช่น Strategy Pattern จะได้ Extensibility
Singleton Pattern จะได้ Integrity
เป็นต้น

# Elements of a Design Pattern

- A pattern has four essential elements (GoF)
  - Name
    - Describes the pattern
    - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
  - Problem
    - Describes when to apply the pattern
    - Answers - What is the pattern trying to solve?

ควรจำ Name ไปใช้ในชีวิตการทำงาน ....เท่มาก

# Elements of a Design Pattern (cont.)

มี Generic Template ให้

- Solution
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences

ผลข้างเคียงที่ควรระวังครับ

  - Results of applying the pattern
  - Benefits and Costs
  - Subjective depending on concrete scenarios

ตัวอย่าง DP

Design Pattern's Name

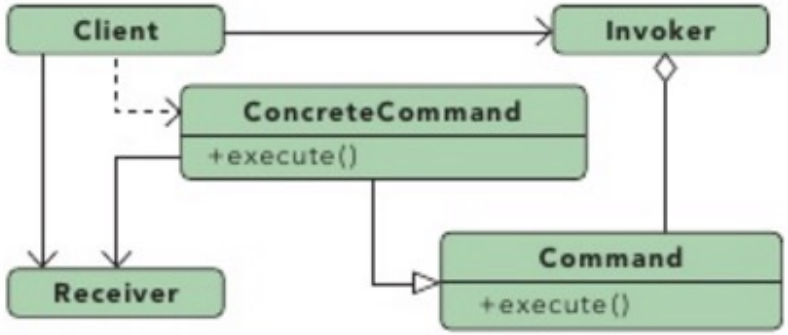
Solution

When to Use

…ขาด Consequences

## COMMAND

Object Behavioral




### Purpose

Encapsulates a request allowing it to be treated as an object. This allows the request to be handled in traditionally object based relationships such as queuing and callbacks.

### Use When

- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

### Example

Job queues are widely used to facilitate the asynchronous processing of algorithms. By utilizing the command pattern the functionality to be executed can be given to a job queue for processing without any need for the queue to have knowledge of the actual implementation it is invoking. The command object that is enqueued implements its particular algorithm within the confines of the interface the queue is expecting.

# Design Patterns Classification

A Pattern can be classified as

- Creational
- Structural
- Behavioral

# GoF: Design Patterns

**Creational:**

Abstract Factory

Builder

Factory Method

Prototype

★ **Singleton**

**Structural:**

★ Adapter

Bridge

**Composite**

★ Decorator

★ **Façade**

Flyweight

Proxy

**Behavioral:**

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

★ **Observer**

**State**

★ Strategy

Template Method

Visitor

# Pros/Cons of Design Patterns

- Pros
  - Add consistency to designs by solving similar problems the same way, independent of language
  - Add clarity to design and design communication by enabling a common vocabulary
  - Improve time to solution by providing templates which serve as foundations for good design
  - Improve reuse through composition

# Pros/Cons of Design Patterns

- Cons
  - Some patterns come with negative consequences (i.e. object proliferation, performance hits, additional layers)
  - Consequences are subjective depending on concrete scenarios
  - Patterns are subject to different interpretations, misinterpretations, and philosophies
  - Patterns can be overused and abused → Anti-Patterns

# Popular Design Patterns in This Class

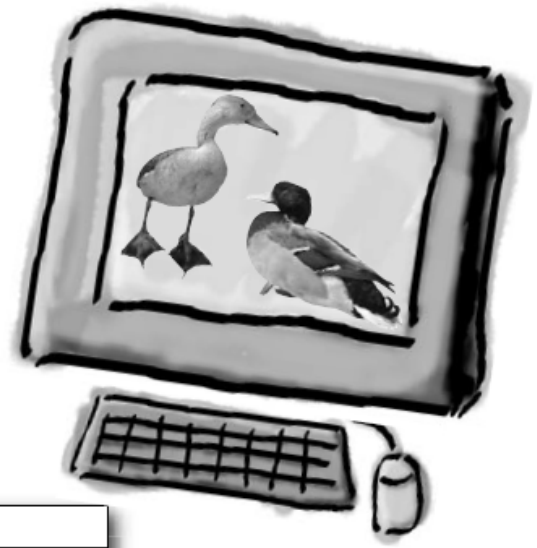
- Let's take a look
  - Strategy
  - Observer
  - Singleton
  - Decorator
  - Proxy
  - Façade
  - Adapter

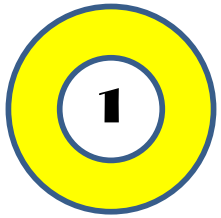
# โจทย์ข้อแรกคือ

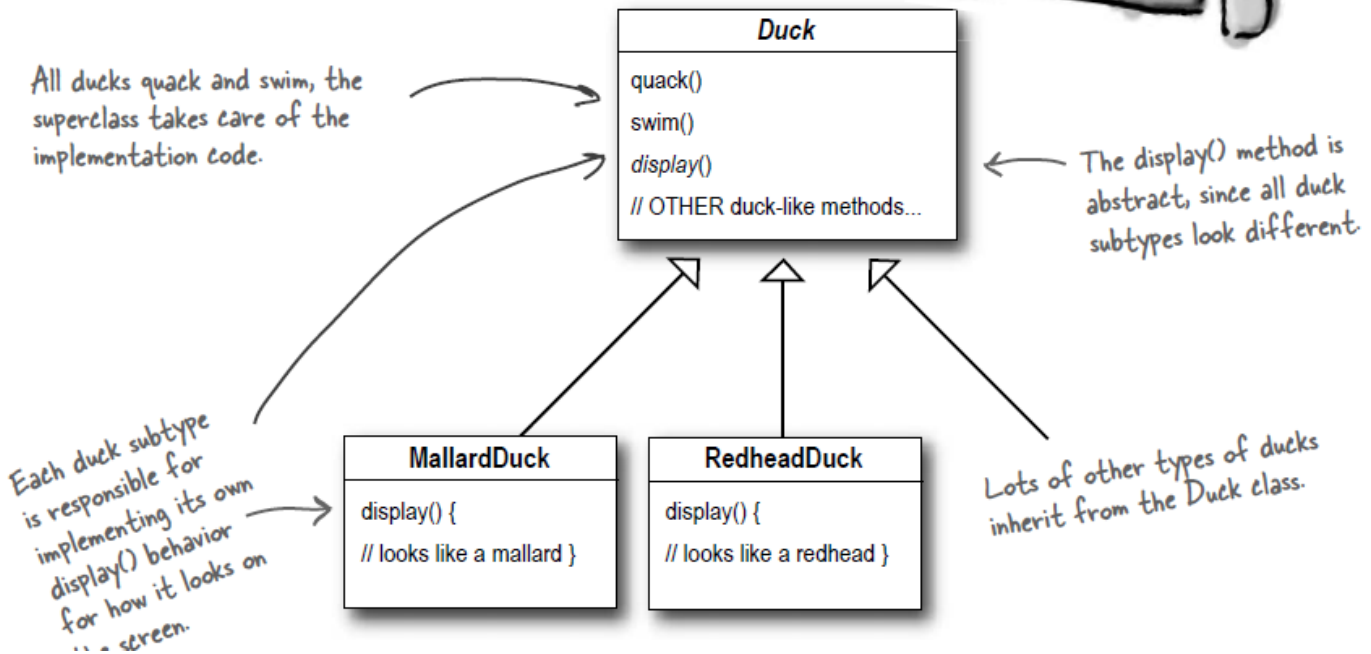
- บริษัทเกมสร้างเกม SimUDuck app โดยคุณ Joe เป็นผู้ออกแบบ
- มีเป็ดหลายชนิดในเกม โดยเป็ดทุกตัวมี Common behaviors คือ quack(), swim() แต่จะมี display() ต่างกัน
- Joe เรียน Object oriented design มาก็เลยออกแบบใช้ Inheritance ได้ตามนี้

## It started with a simple SimUDuck app

Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit.

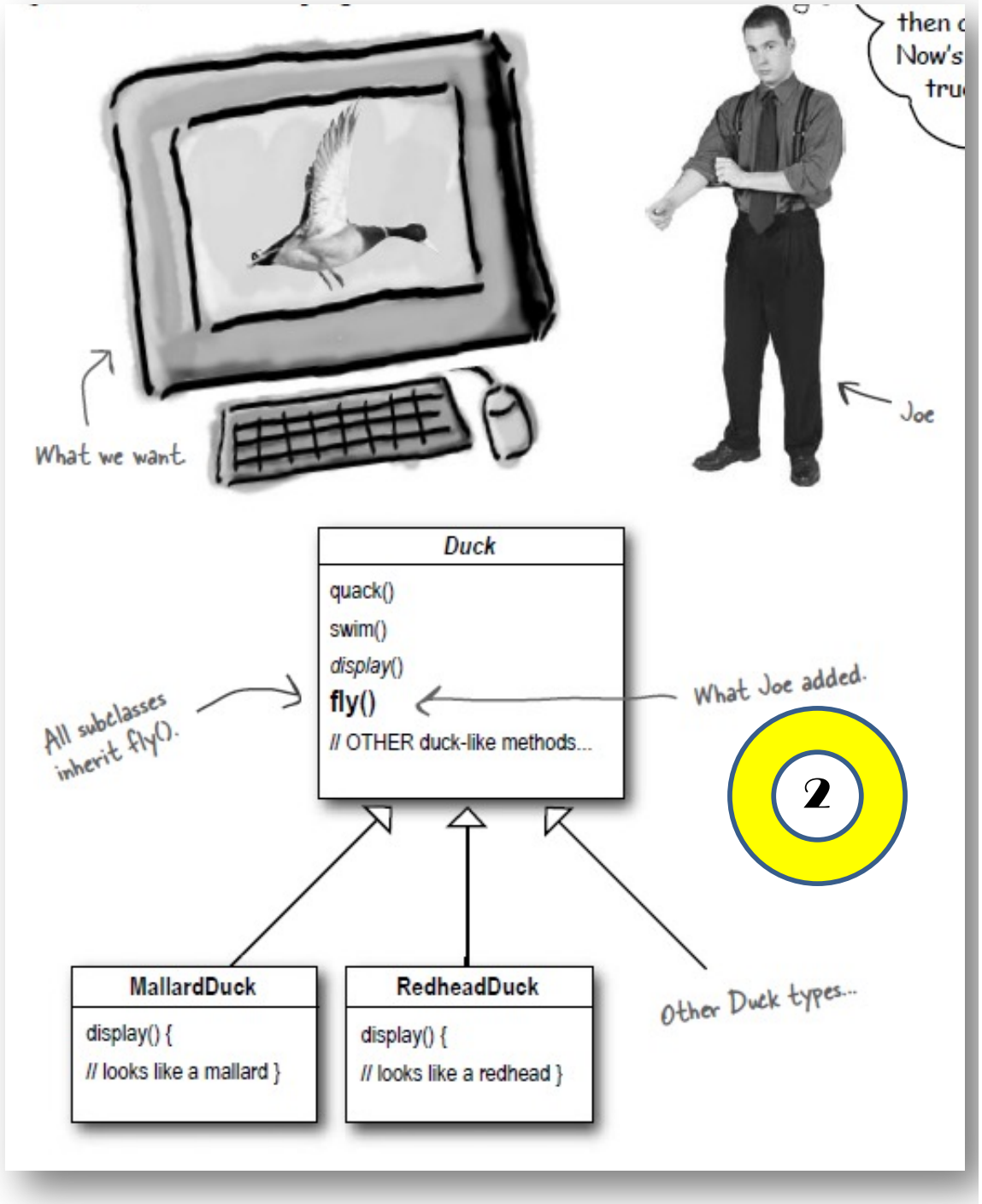




# หัวหน้าสั่งเพิ่มให้เป็ดบิน แข่งกับเกมอื่นๆ

- Joe สามารถเพิ่ม behavior ให้เป็ดได้ที่ supper class เช่น fly() เป็ดทุกตัวก็บินได้ทันที
- Joe ภูมิใจกับความรู้เรื่อง Inheritance ที่ร่ำเรียนมาอย่างมาก สามารถแก้ปัญหาโดยมีการเขียน fly() ครั้งเดียว และเป็ดทุกตัวที่เป็น subclass ก็บินได้เหมือนกันหมดจากการถ่ายทอด เกิดการ reusability



Ann โทรมาหาต่อว่า Joe

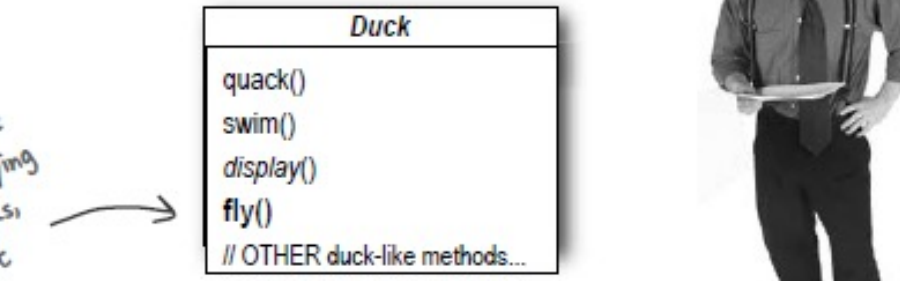
## What happened?

Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

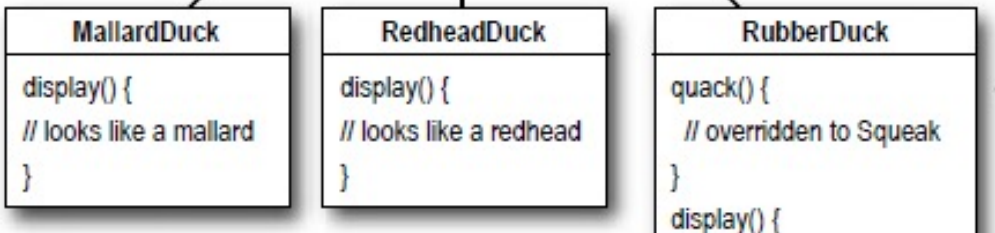
*A localized update to the code caused a non-local side effect (flying rubber ducks)!*


What he thought was a great use of inheritance for the purpose of reuse hasn't turned out so well when it comes to maintenance.

By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.




3

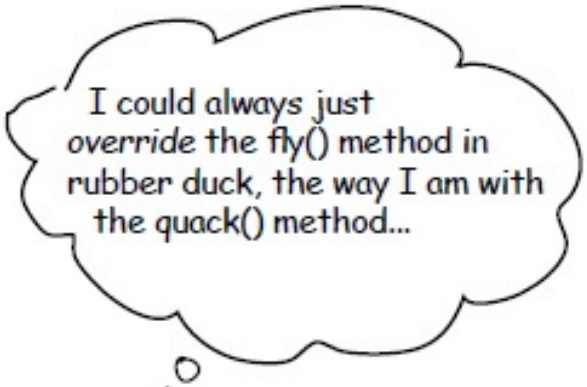



Rubber ducks don't quack, so quack() is overridden to "Squeak".

เกิดปัญหาด้าน maintainability

เพราะอะไร? RubberDuck can fly?

# Joe thinks about inheritance...

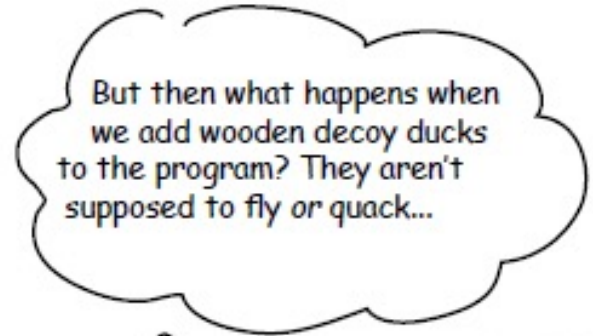



RubberDuck

```
quack()  { // squeak}
display() { .// rubber duck }
fly() {
  // override to do nothing
}
```

4

Joe แก้ปัญหาอย่างไรดี?
อ้อ ก็ override fly() ใหม่ที่ Subclass ให้ไม่ทำอะไรก็จบเรื่อง
... ดีจริงไหม?



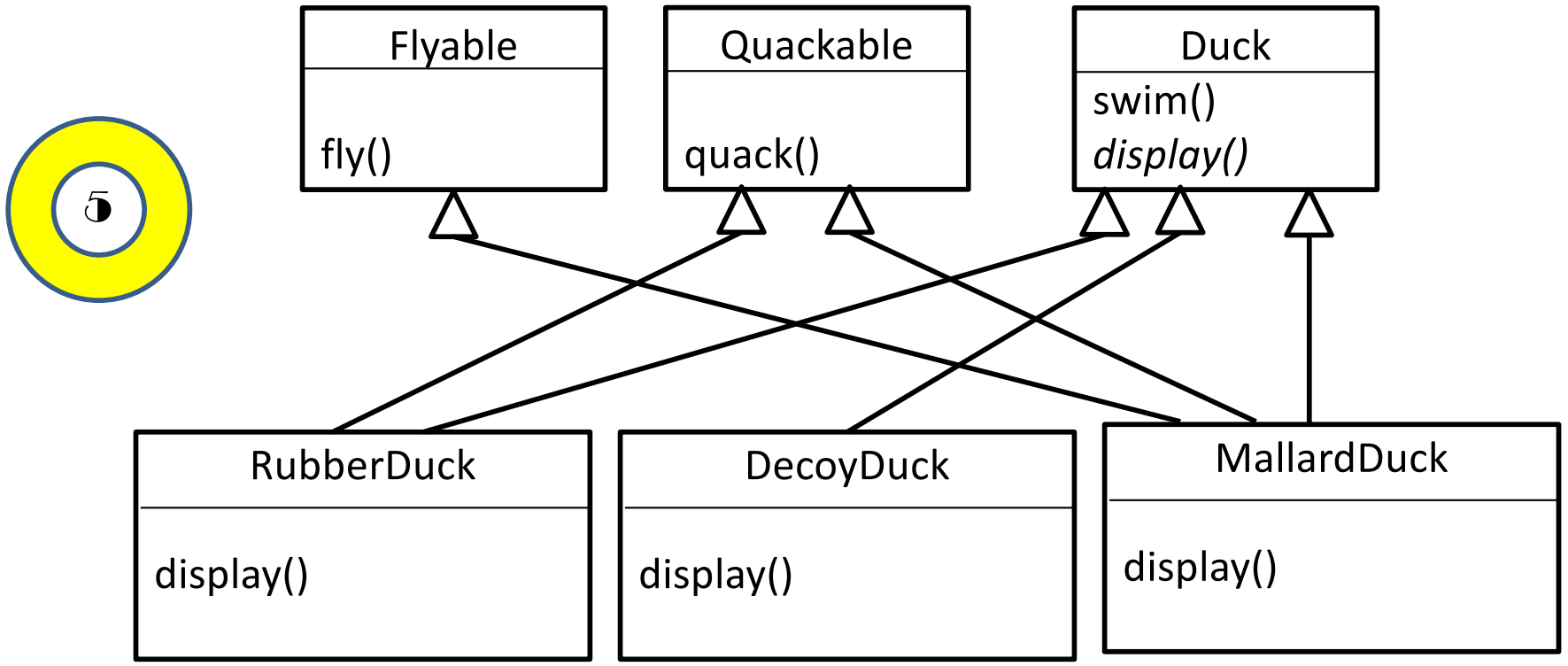

DecoyDuck

```
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

ต่อจากนี้ ใครต้องการใช้ Superclass Duck
ก็ต้องระวังว่าไม่ใช่ "COMMON Behaviors"
ต้องระวังคอย override methods ที่ไม่ใช้เองนะ
.... เผลอก็แย่เลย

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

- ถ้าเราจัดการเรื่อง COMMON behaviors เราอาจจะแยก COMMON behaviors ออกจากกัน
- และให้เลือกถ่ายทอดไปตามต้องการดีไหม




เกิดปัญหาอะไรครับ ???

Multiple Inheritance

# Interface ช่วยด้วย

- Interface ช่วยแก้ปัญหา Multiple Inheritance โดยเลือก superclass หลักแค่หนึ่งเดียว
- และใช้การทำ <<implement>> Interfaces แทนครับ






**What do YOU think about this design?**

**Design ดีขึ้น reuse, ไม่ต้องมา override กันทุกครั้ง แต่เกิดอะไรขึ้นจากนี้**

**เกิด Duplicate Code ครับ ต้อง implement fly(), quack() ซ้ำ ๆกันทุกครั้งไป**

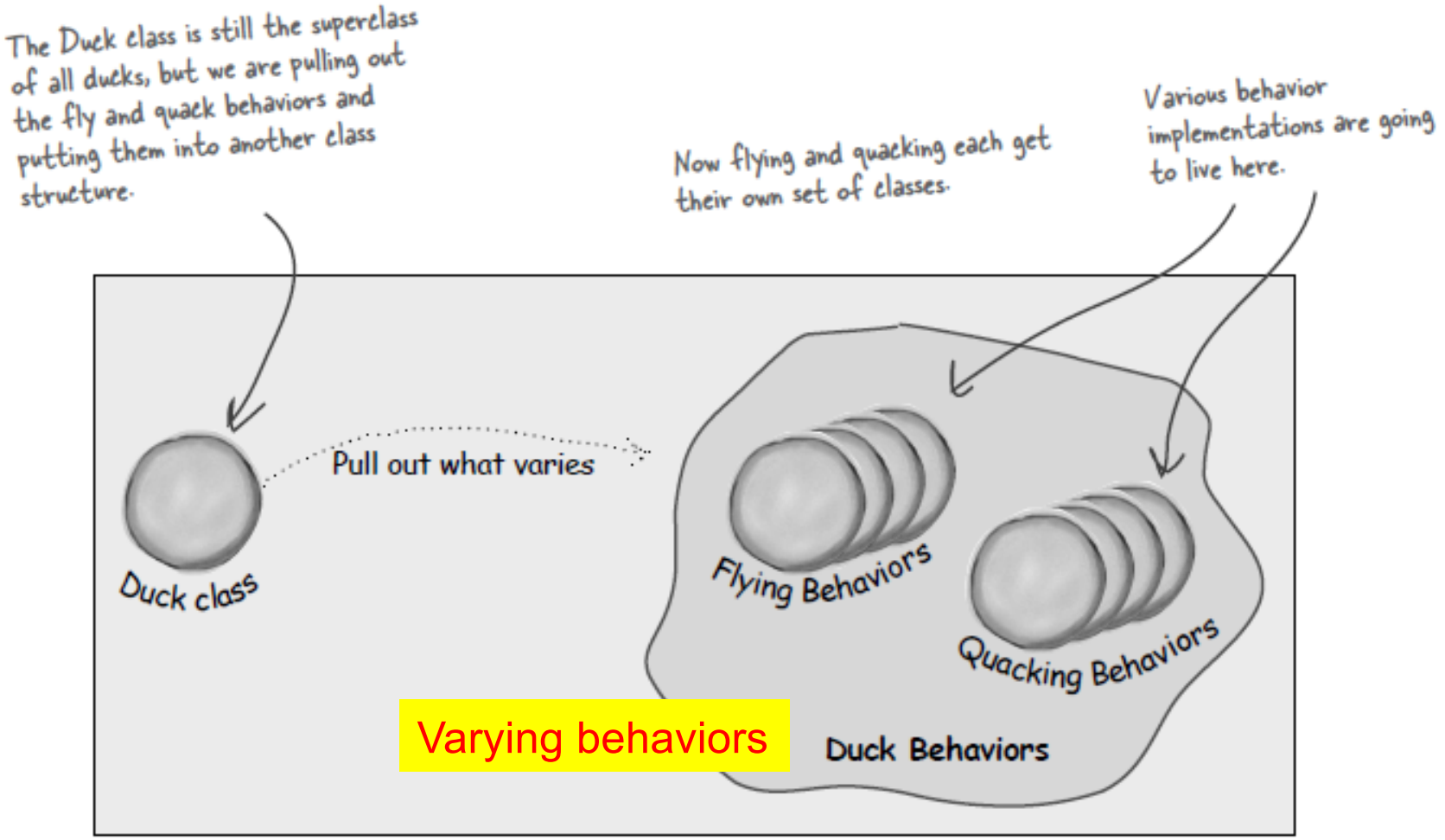
**Design Principle**

*Identify the aspects of your application that vary and separate them from what stays the same.*

Our first of many design principles. We'll spend more time on these thruoghout the book.

**We know that fly() and quack() are the parts of the Duck class that vary across ducks.**

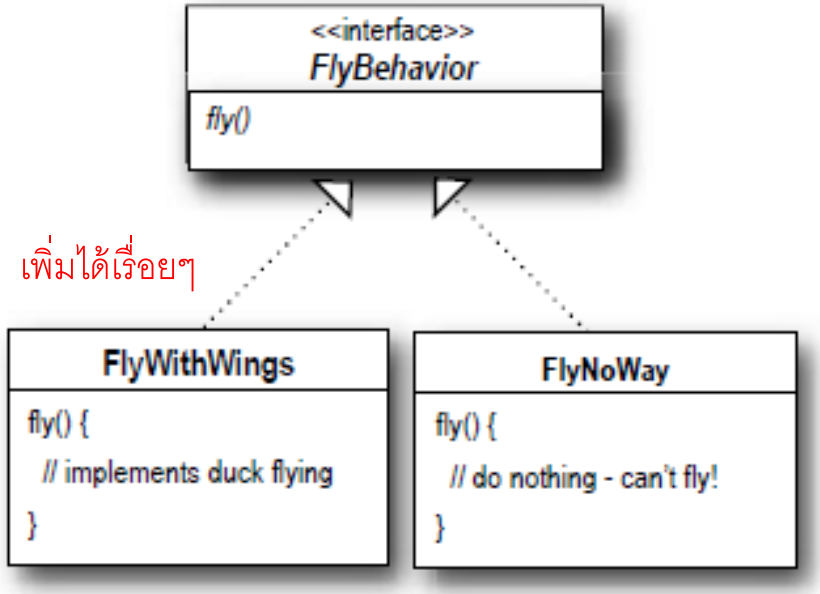
**To separate these behaviors from the Duck class, we'll pull both methods *out* of the Duck class and create a new set of classes to represent each behavior.**

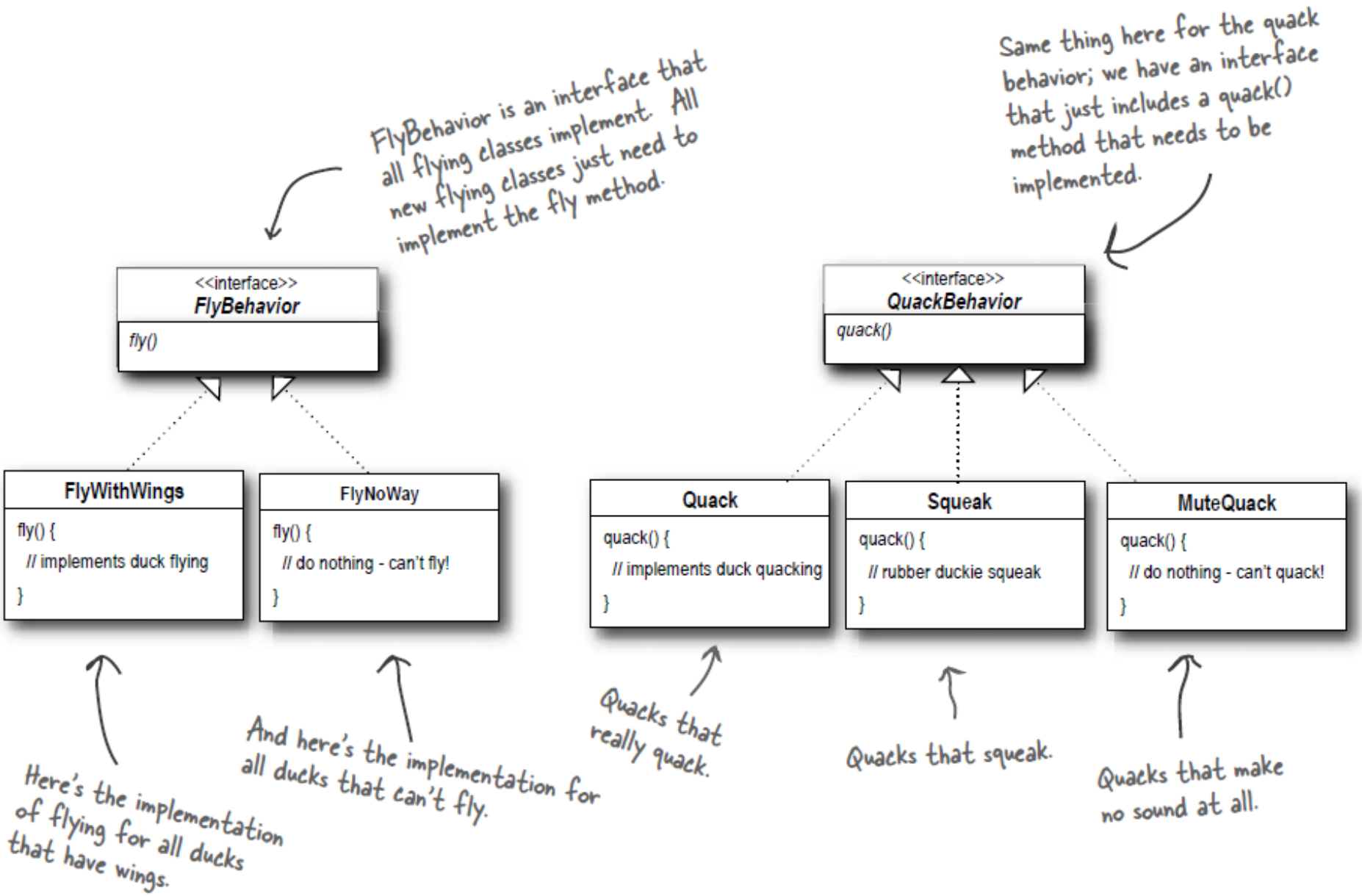


**Design Principle**

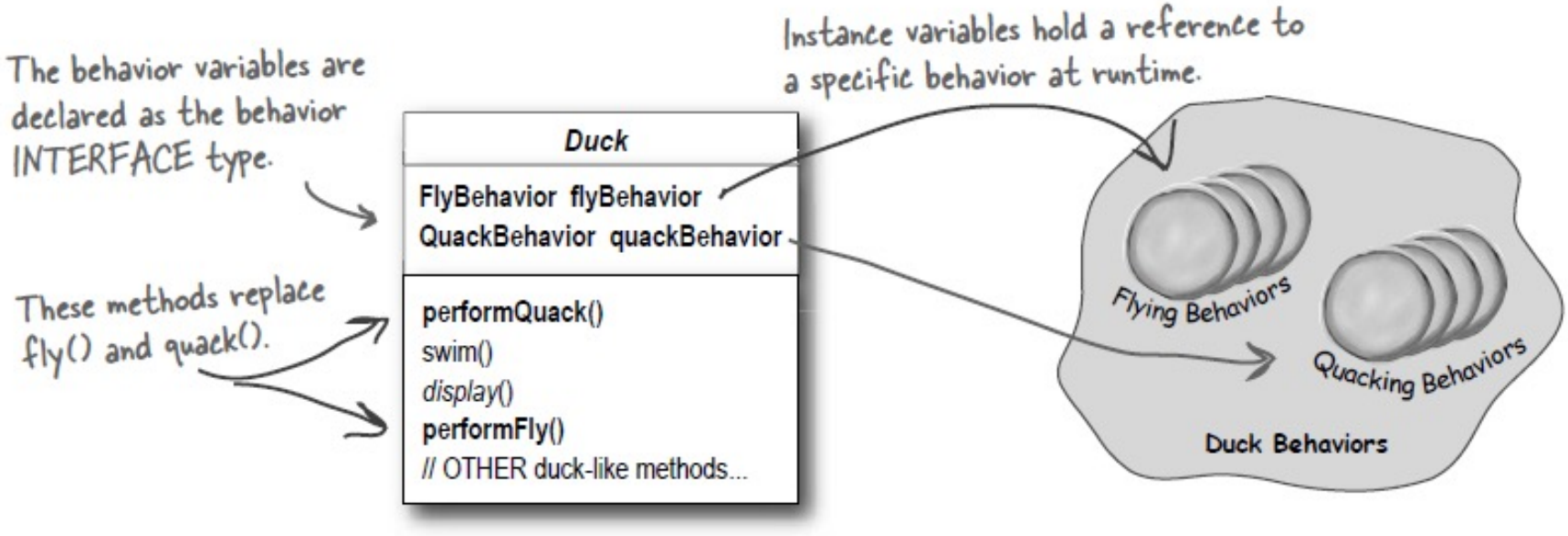
*Program to an interface, not an implementation.*

That way, the Duck classes won't need to know any of the implementation details for their own behaviors.




FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly method.
Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.
<<interface>>
FlyBehavior
fly()
<<interface>>
QuackBehavior
quack()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
Here's the implementation of flying for all ducks that have wings.
And here's the implementation for all ducks that can't fly.
Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.

1) ใช้การ separate ส่วนที่ vary ออกไป
2) Wrap ชุดของ behaviors เหล่านั้นด้วย Interface
3) Delegate ให้ object อื่นทำ behavior นั้นแทนตน



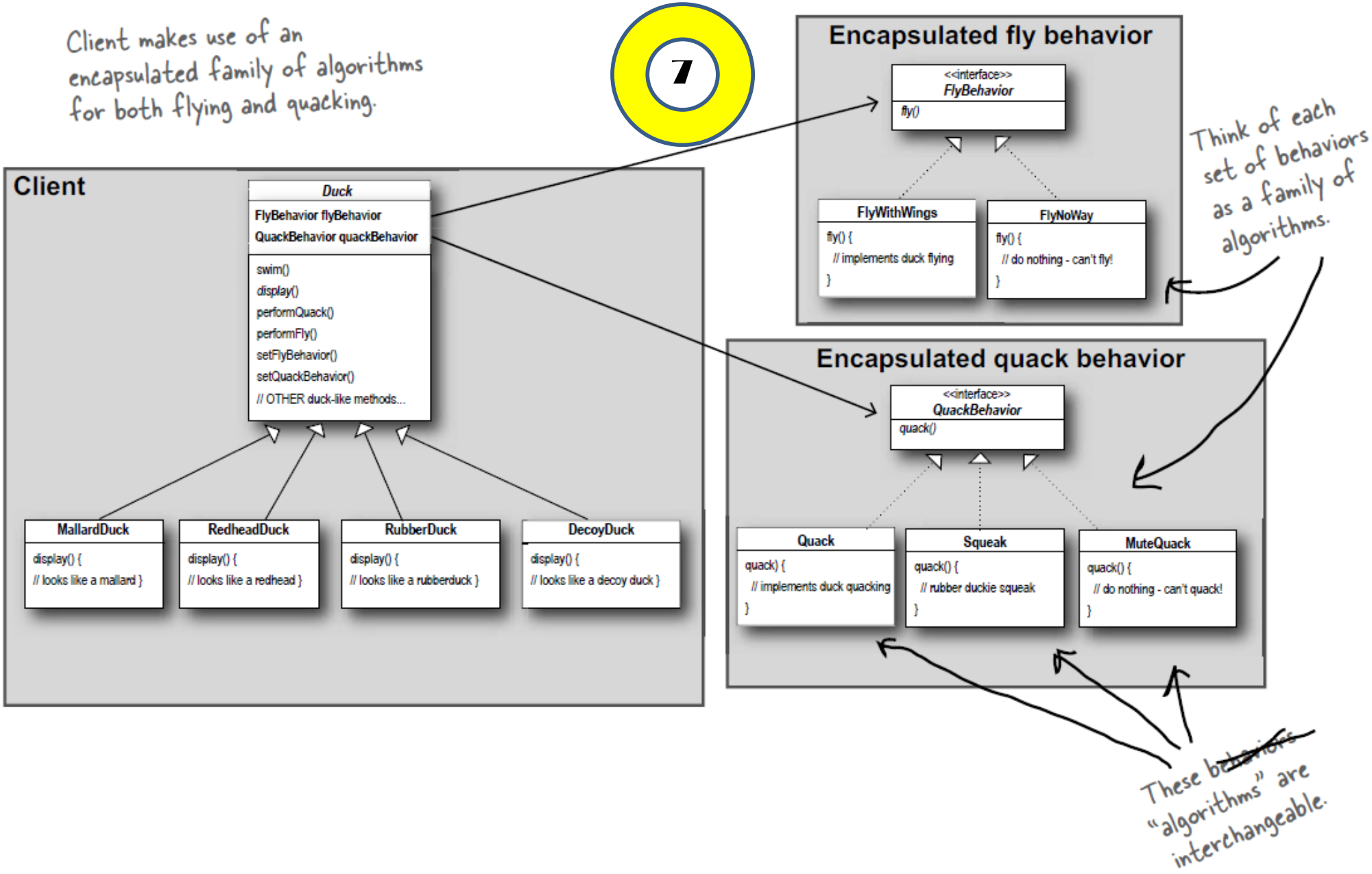

**2** **Now we implement performQuack():**

```
public class Duck {
    QuackBehavior quackBehavior;
    // more

    public void performQuack() {
      quackBehavior.quack();
    }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

# Good Design


Client makes use of an encapsulated family of algorithms for both flying and quacking.
7
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack) {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These behaviors "algorithms" are interchangeable.

# Strategy Definition

Defines a family of algorithms,

encapsulates each one,

and makes them interchangeable.

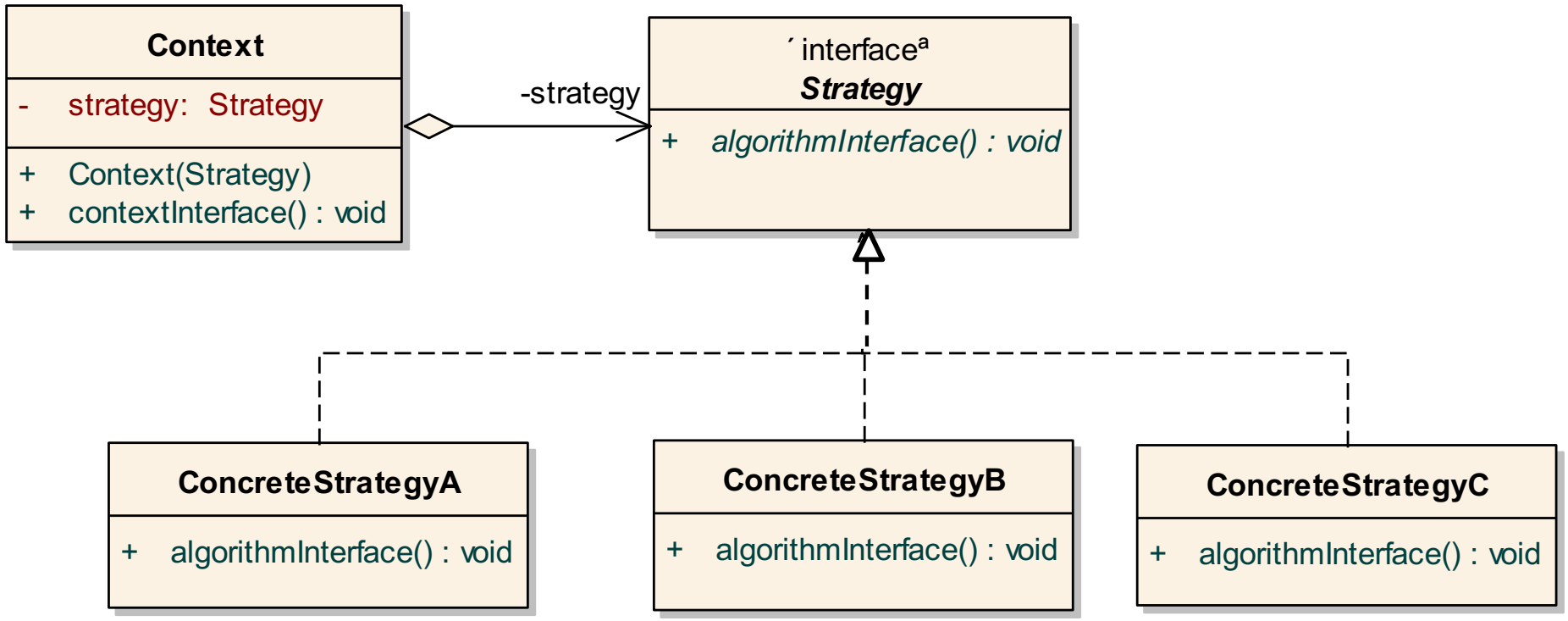
Strategy lets the algorithm vary independently from clients that use it.

# Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance

หลักการออกแบบที่ใช้เป็นเหตุผลในการจัดการ design patterns

# Strategy – Class diagram template

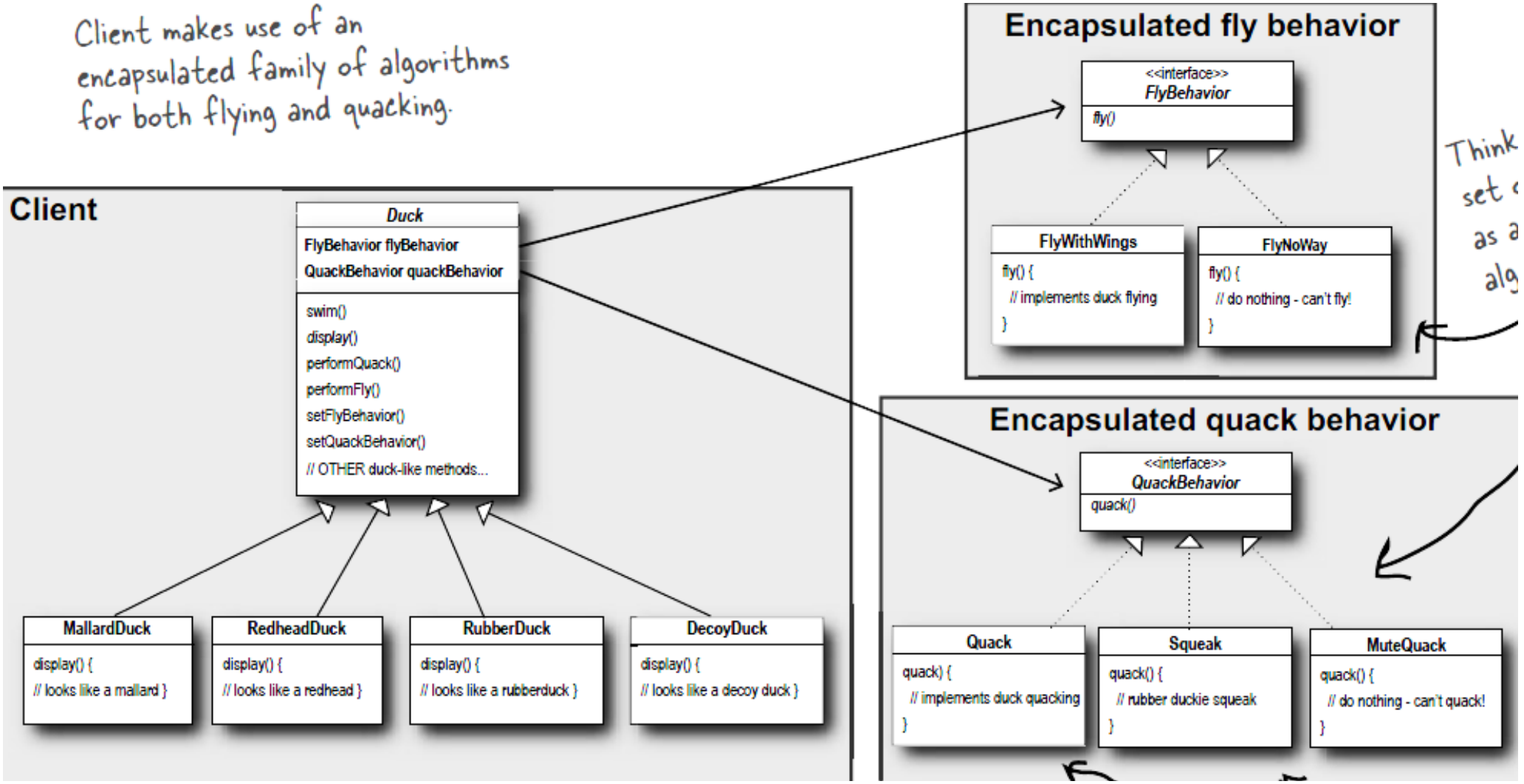



ระวัง ต้องใช้สัญลักษณ์ให้ถูกต้องเสมอ

# Strategy

- Pros
  - Provides encapsulation
  - Hides implementation
  - Allows behavior change at runtime
- Cons
  - Results in complex, hard to understand code if overused

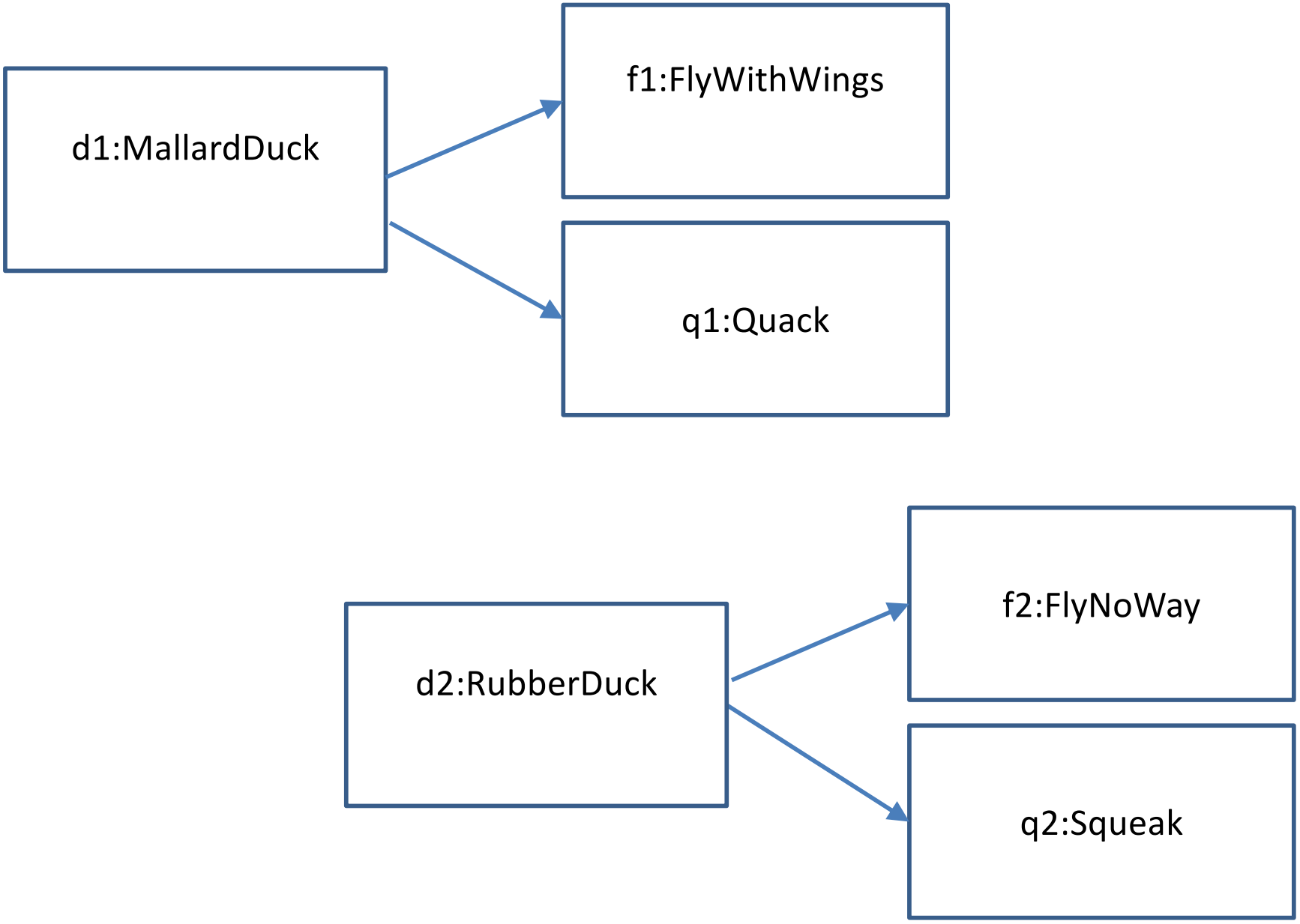
# Good Design


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack) {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}

# เรามาดู snapshot ของระบบ

- เราใช้ Object diagram
- มีกี่ Objects?
  - d1, f1, q1
  - d2, f2, q2
- ตอนกำเนิดเป็ด d1: MallardDuck ที่บินด้วย f1:FlyWithWings และร้องด้วย q1:Quack
- ตอนกำเนิดเป็ด d2:RubberDuck ที่บินด้วย f2:FlyNoWay และร้องด้วย q2:Squeak

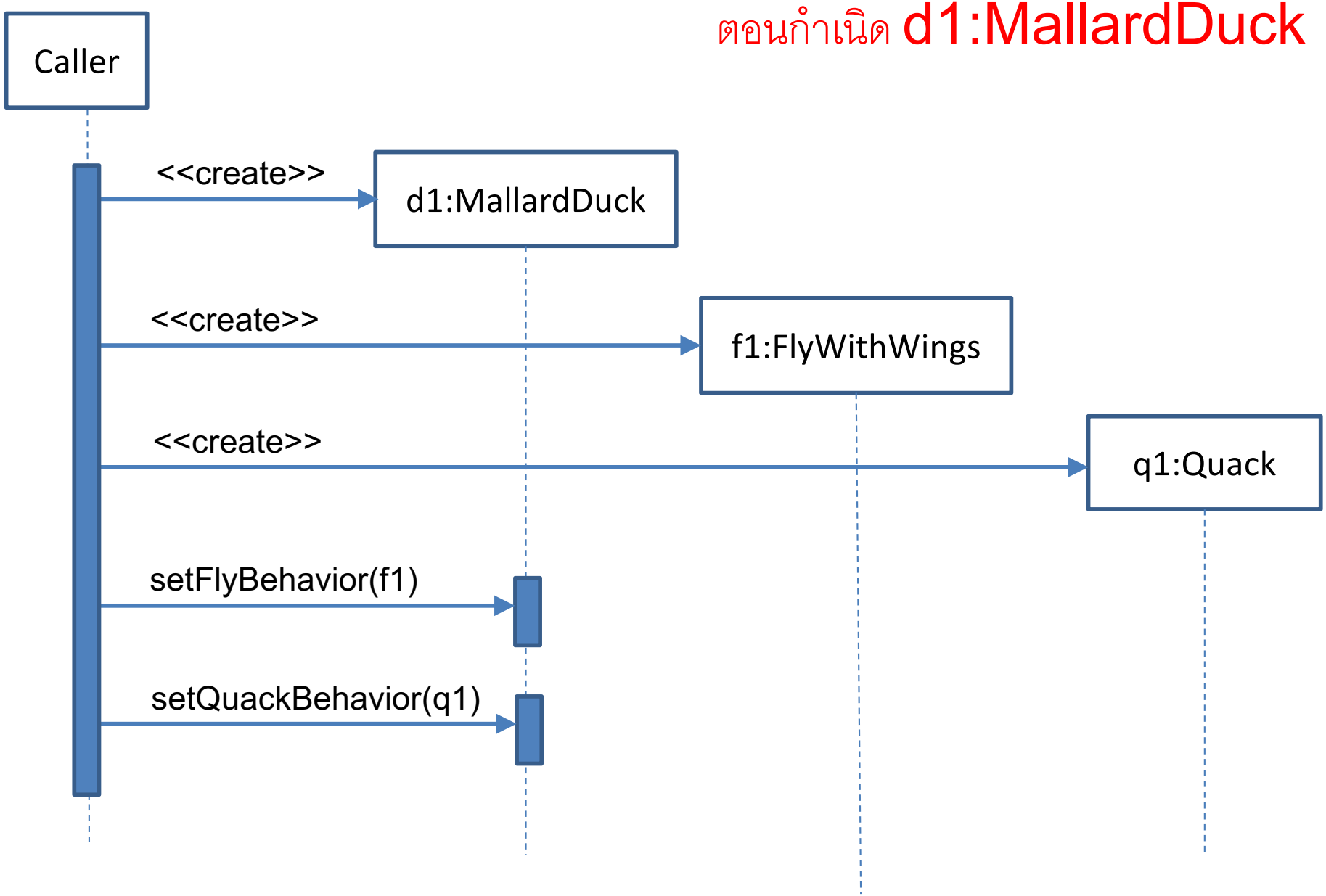
# Object Diagram ของ d1:MallardDuck และ d2:RubberDuck

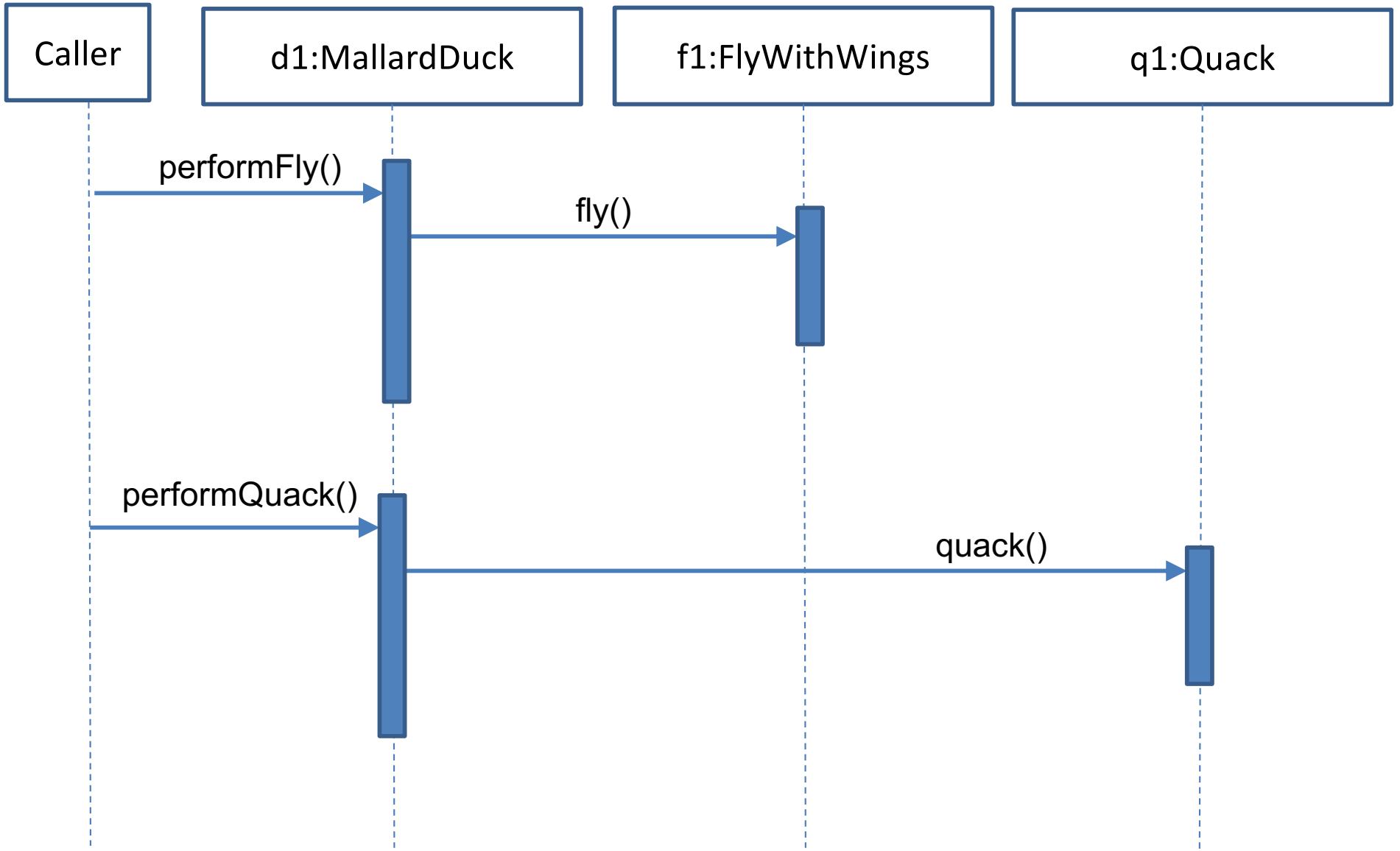


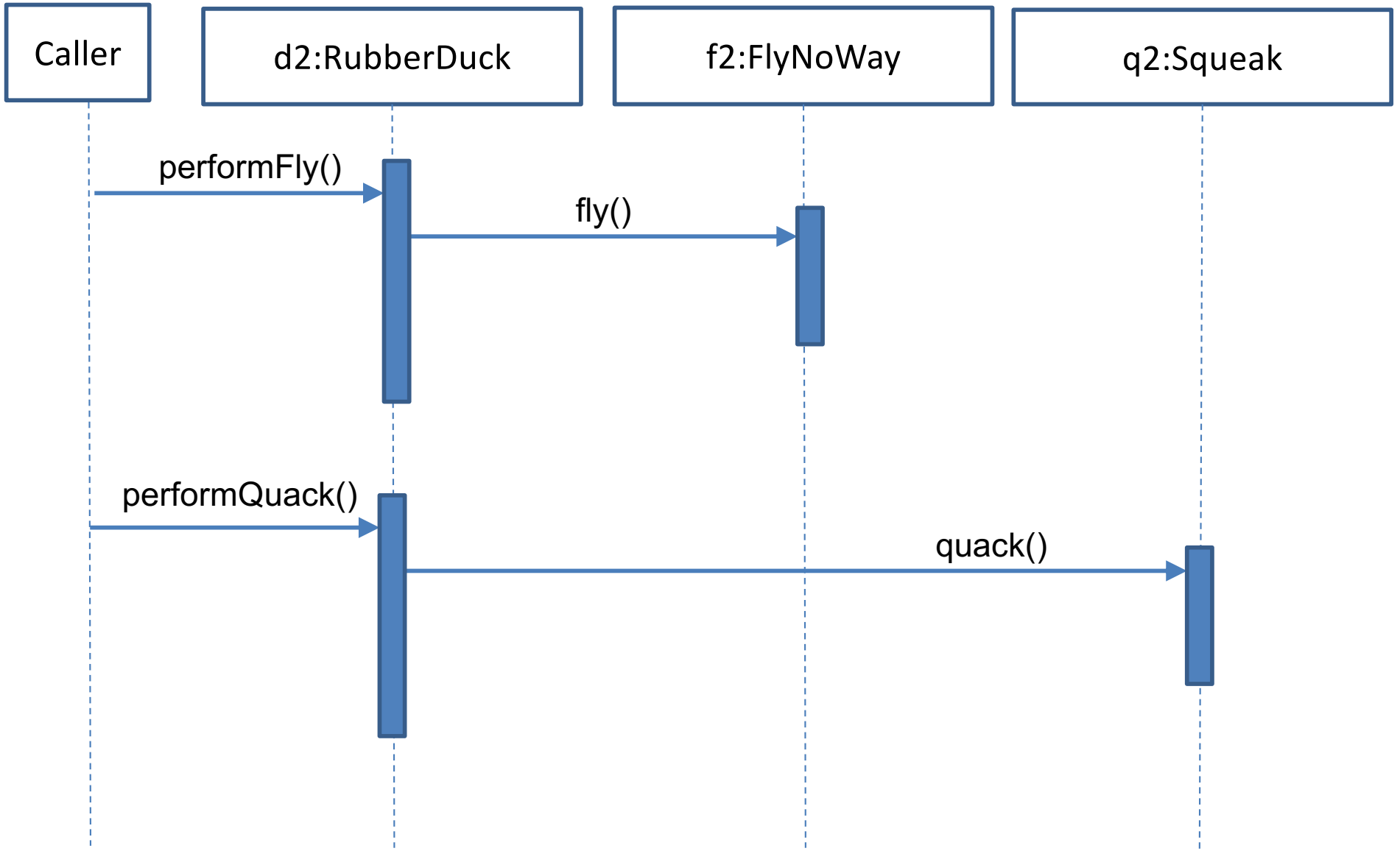
# เรามาออกแบบพฤติกรรมของเป็ด

- เราใช้ Sequence diagram
- ตอนกำเนิดเป็ด d1: MallardDuck
- ตอนสั่งให้เป็ด d1: MallardDuck ทำการ performQuack()
- ตอนสั่งให้เป็น d1: MallardDuck ทำการ performFly()
- ตอนกำเนิดเป็ด d2:RubberDuck
- ตอนสั่งให้เป็ด d2:RubberDuck ทำการ performQuack()
- ตอนสั่งให้เป็น d2:RubberDuck ทำการ performFly()

# ตอนกำเนิด d1:MallardDuck




Caller
d1:MallardDuck
f1:FlyWithWings
q1:Quack
performFly()
fly()
performQuack()
quack()

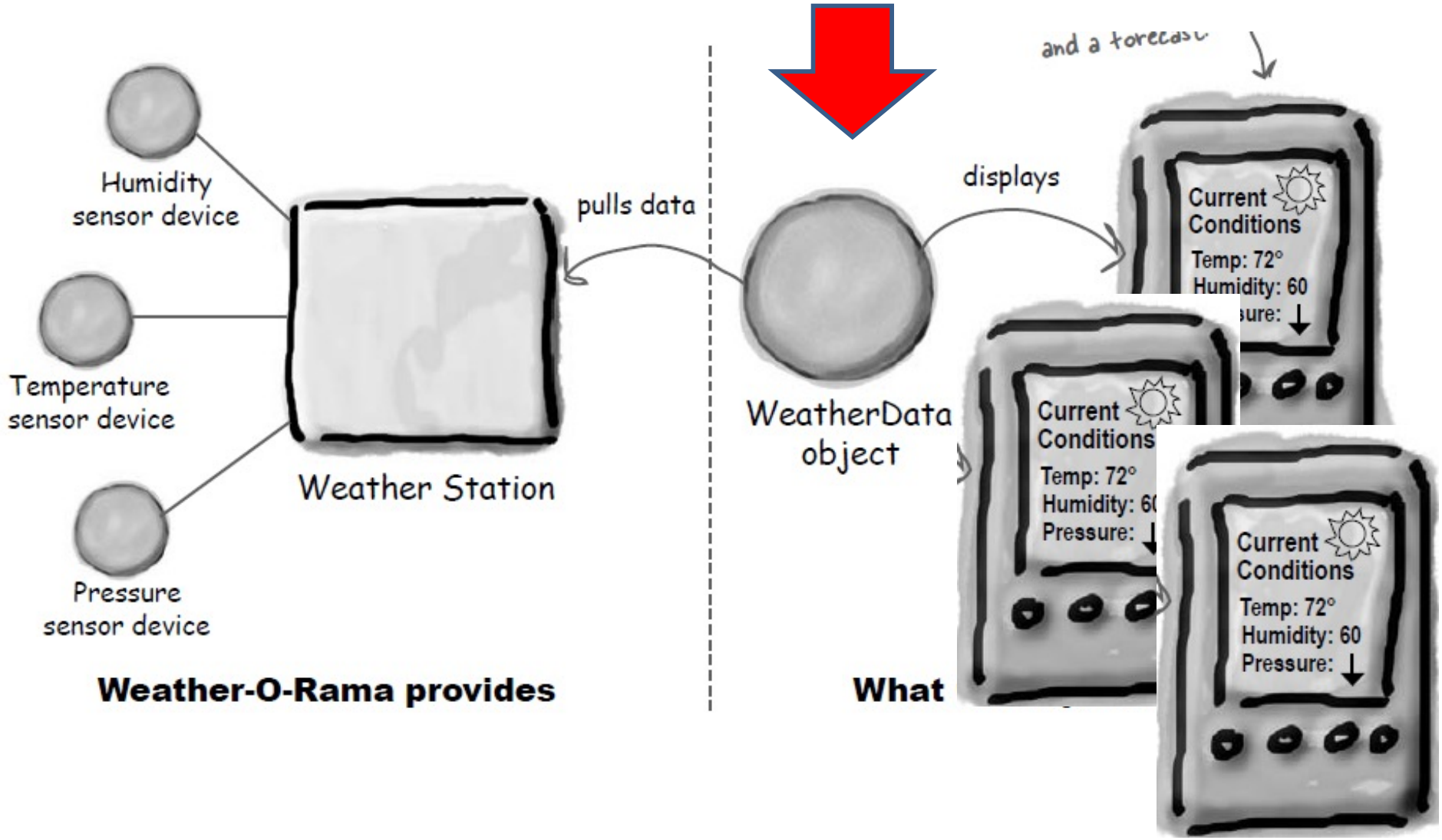

Caller
d2:RubberDuck
f2:FlyNoWay
q2:Squeak
performFly()
fly()
performQuack()
quack()

# Exercise

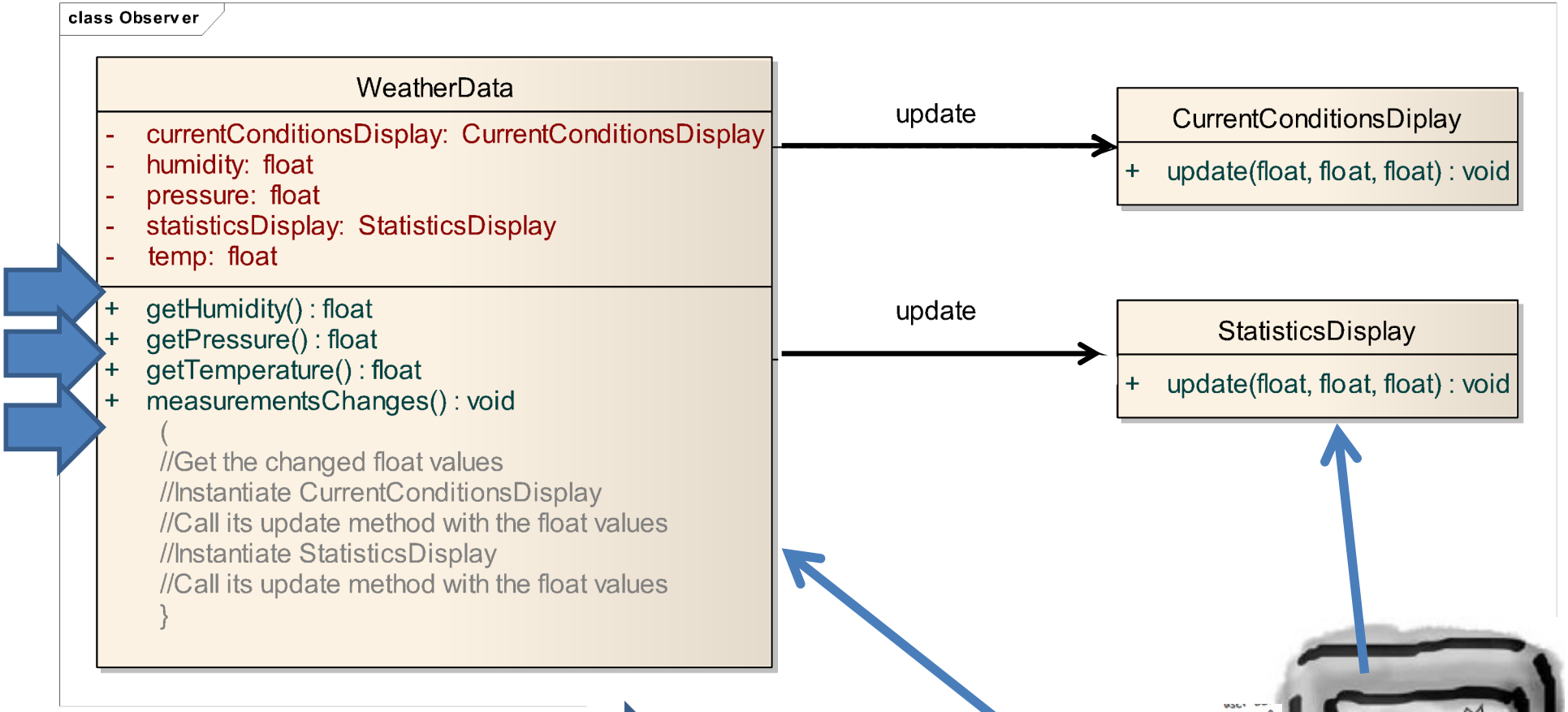
- จงออกแบบ Class diagram, Object diagram, Sequence diagrams ของระบบบัญชีเงินฝาก ที่มีหลายประเภทคือ Saving account, 6 month fixed account, 12 month fixed account, Special fixed account, etc. และมีการคิดดอกเบี้ยที่แตกต่างกันคือ 0.5%, 1%, 1.5%, 2% ตามลำดับ
- เราต้องออกแบบให้รองรับการเปลี่ยนแปลงของอัตราดอกเบี้ยที่ปรับเปลี่ยนเสมอ และอาจจะมีบัญชีเงินฝากแบบใหม่เสมอ โดยไม่กระทบของเดิม หรือกระทบให้น้อยที่สุด

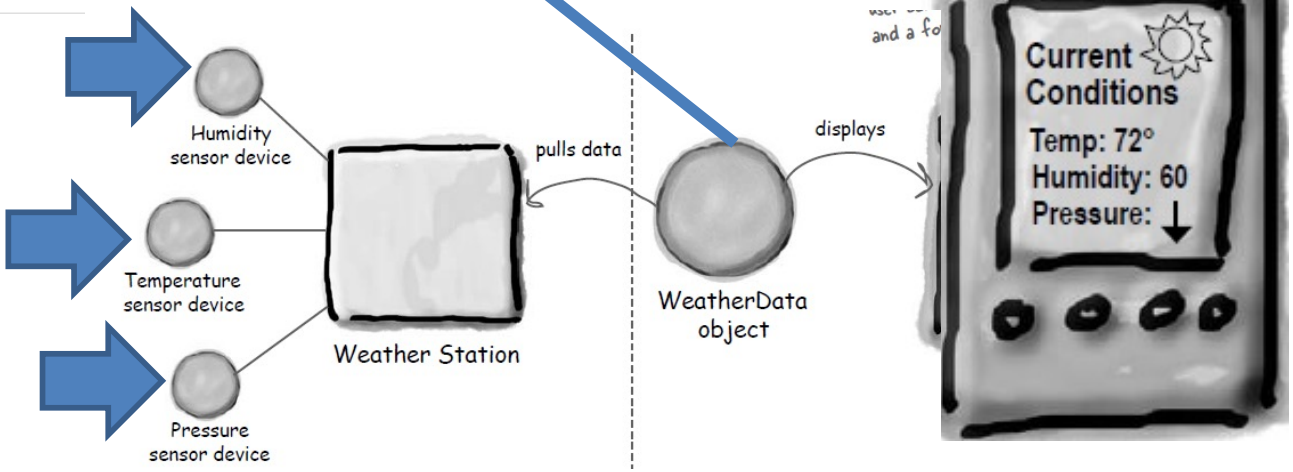
โจทย์

มีอุปกรณ์ตรวจวัดที่ติดตั้งไว้ และต้องการดึง (Pull) ข้อมูลมาแสดงผลด้วยอุปกรณ์ Display Device หลายแบบและจำนวนไม่แน่นอน



# Problem การ Push data ไปที่ device

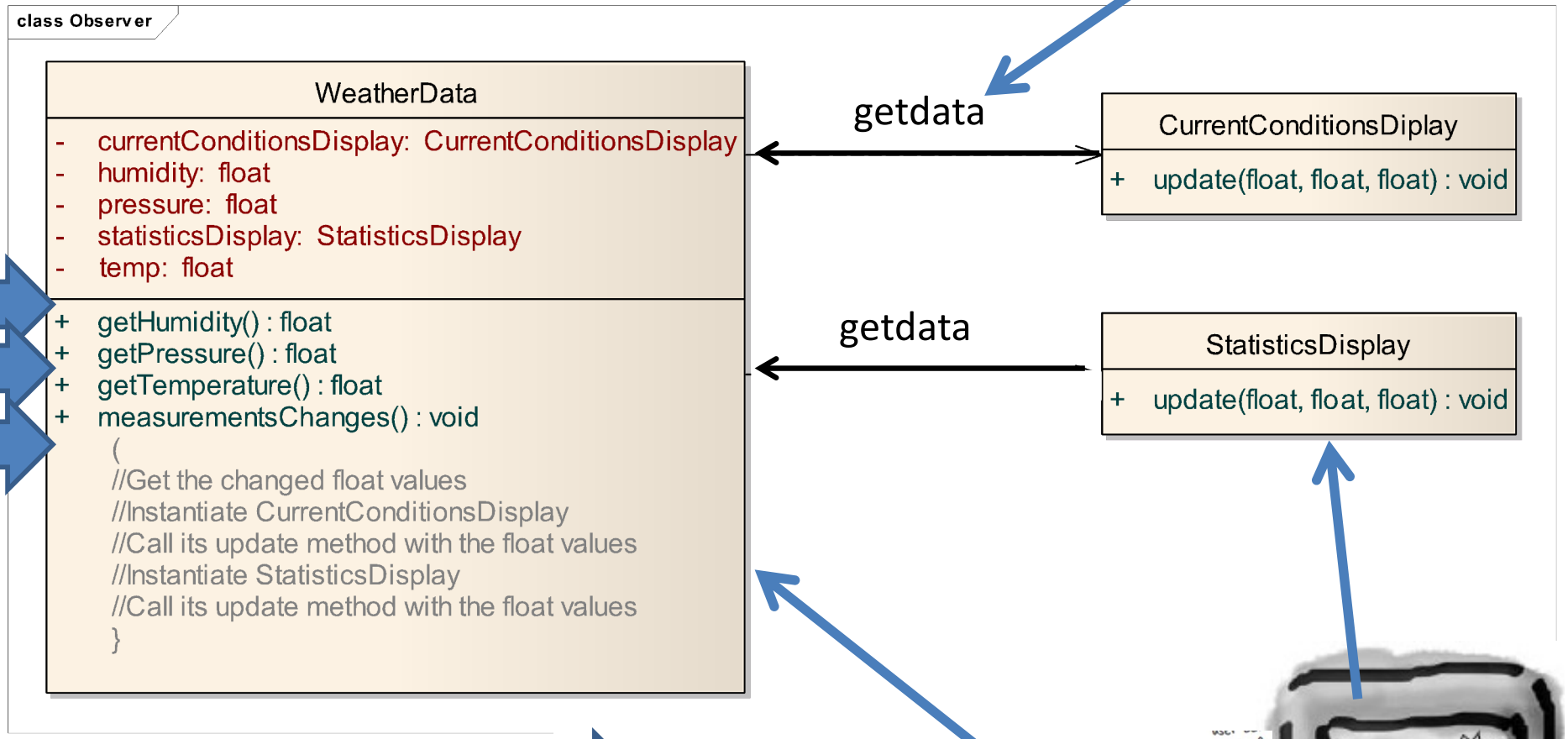


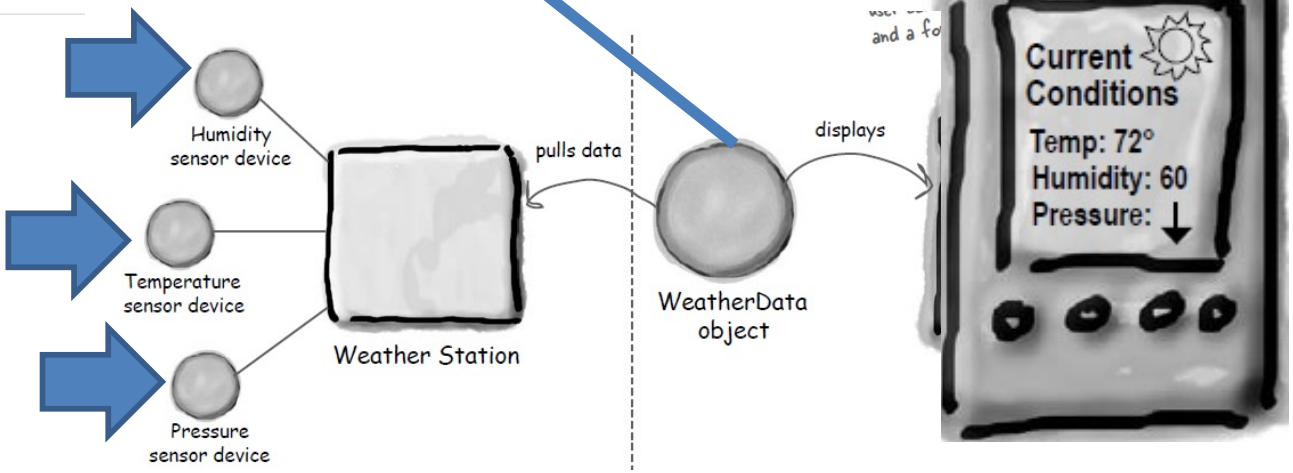




**ปัญหาคือ**

**จำนวน Display device ที่เพิ่มลดทำให้ต้องแก้ไข WeatherData เสมอ**

# หรือปัญหาเรื่องการ device ต้อง Pull data

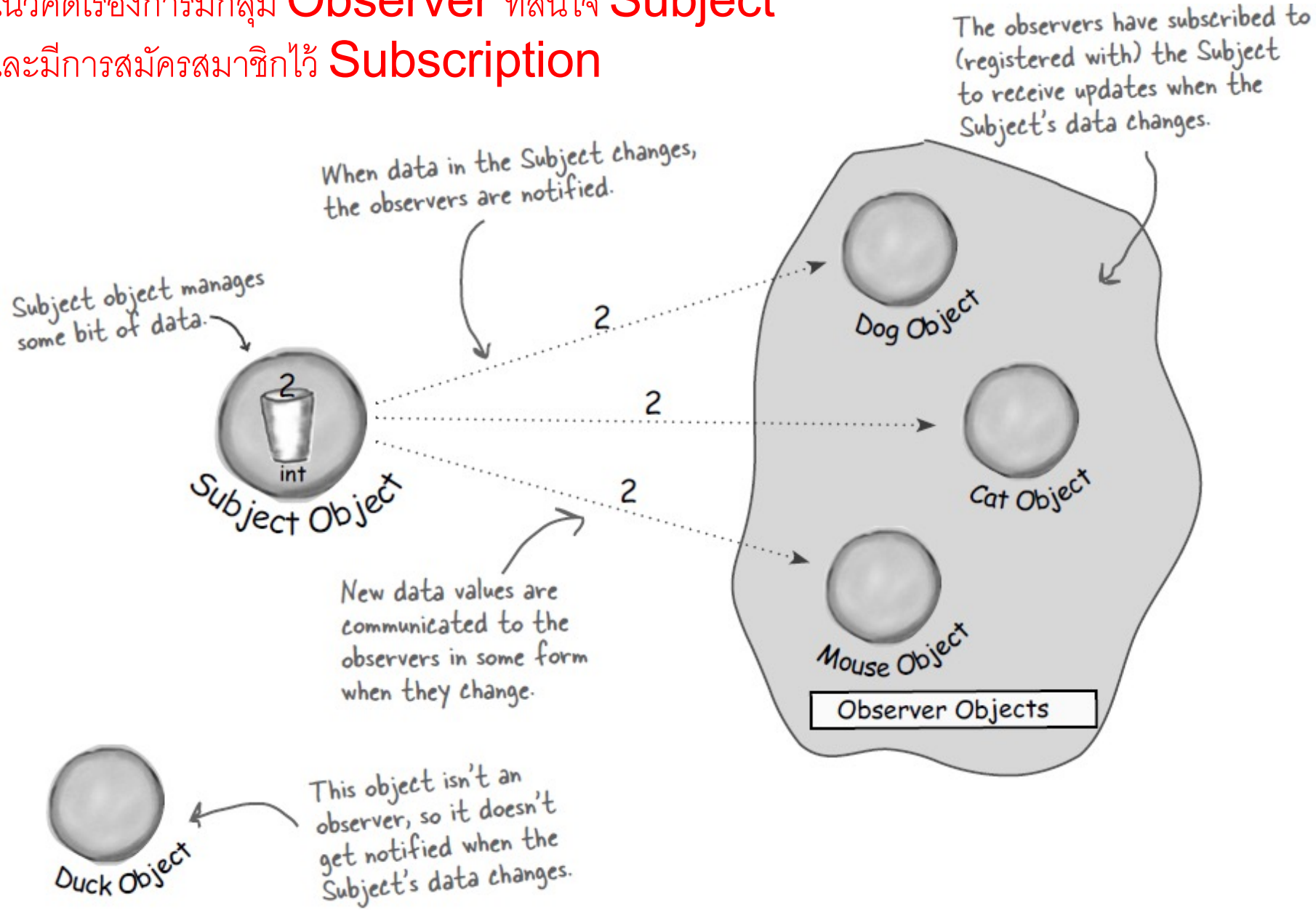


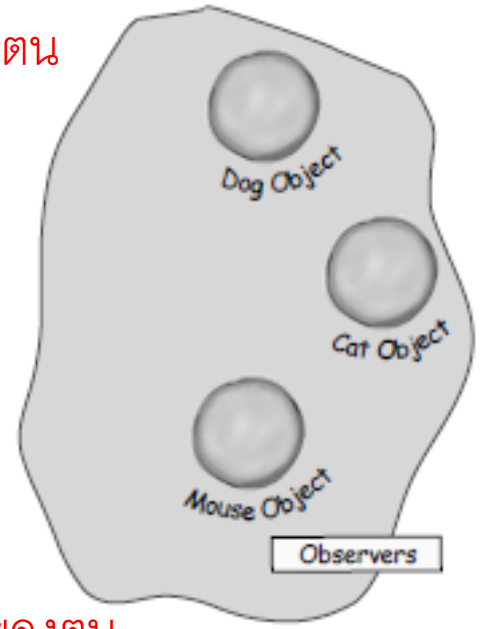

**ปัญหาคือ**

**จำนวน Display device ที่เพิ่มลดทำให้ต้องแก้ไข WeatherData เสมอ**

# แนวคิดเรื่องการมีกลุ่ม Observer ที่สนใจ Subject และมีการสมัครสมาชิกไว้ Subscription

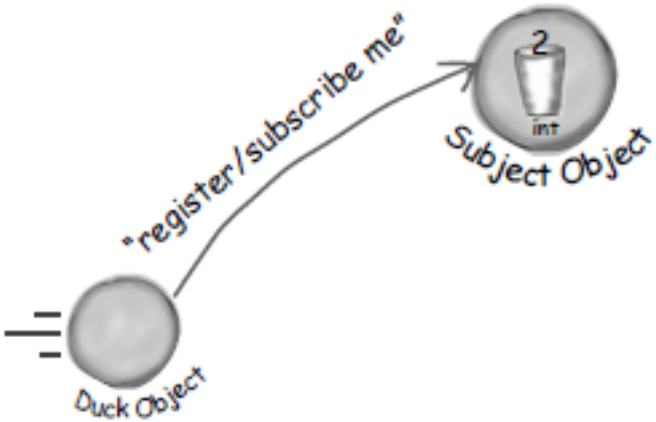


# Subject มี List ของ Observers ของตน




**A Duck object comes along and tells the Subject that it wants to become an observer.**

Duck really wants in on the action; those ints Subject is sending out whenever its state changes look pretty interesting...

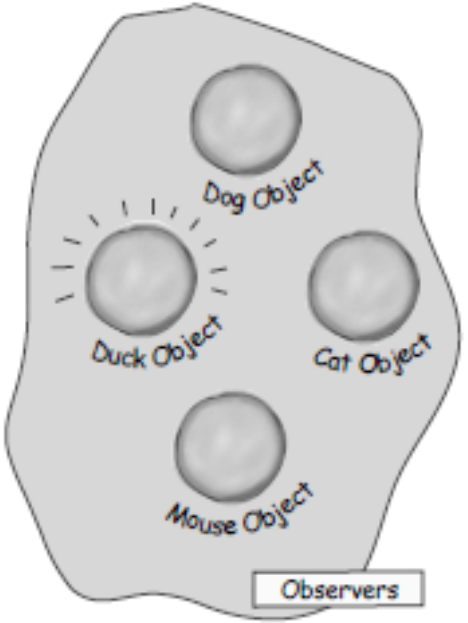



## Register new observer ของตน

---

**The Duck object is now an official observer.**

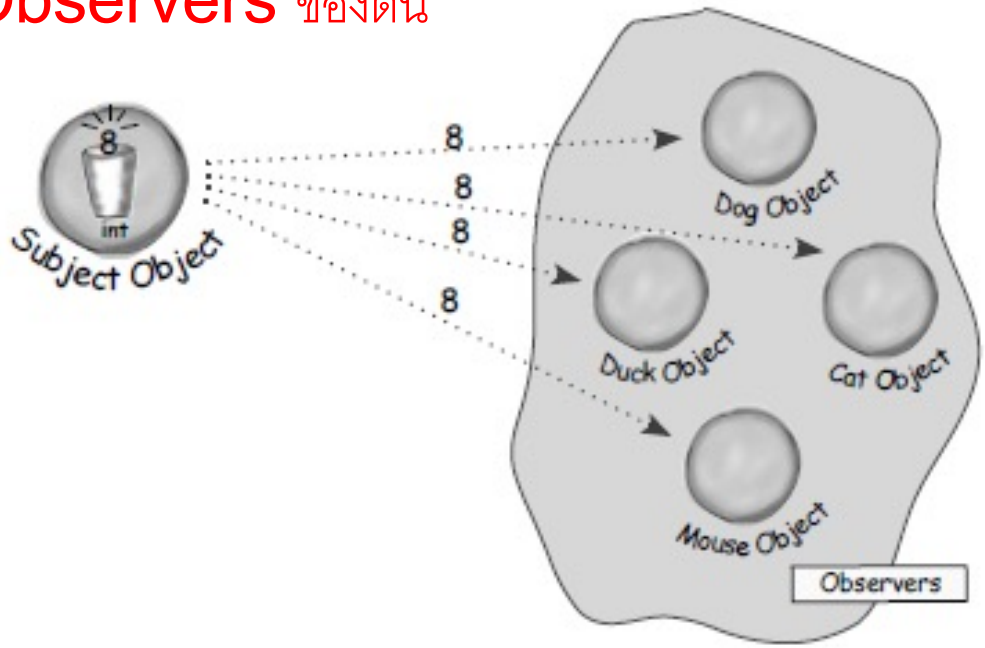
Duck is psyched... he's on the list and is waiting with great anticipation for the next notification so he can get an int.

# Subject มี Push data ที่เปลี่ยนแปลง ไปให้ตาม List ของ Observers ของตน

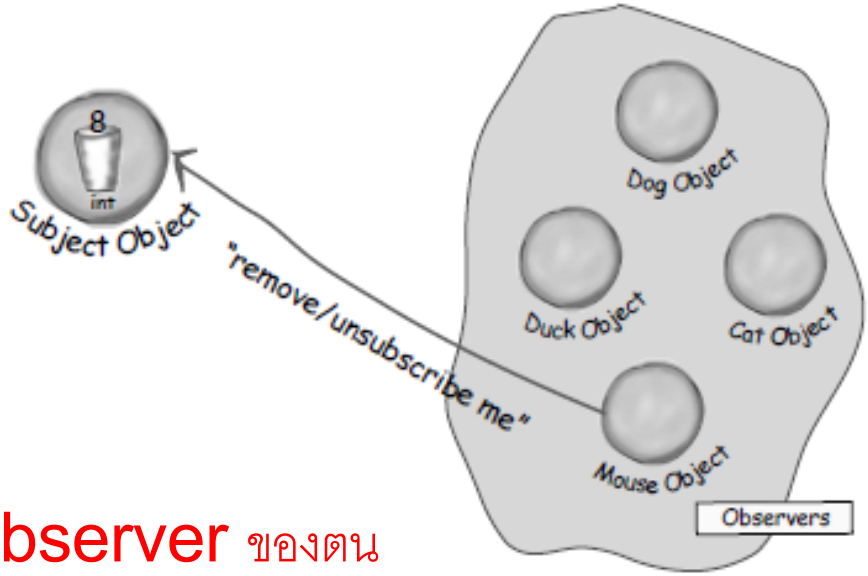
The Subject gets a new data value!

Now Duck and all the rest of the observers get a notification that the Subject has changed.



**The Mouse object asks to be removed as an observer.**

The Mouse object has been getting ints for ages and is tired of it, so it decides it's time to stop being an observer.

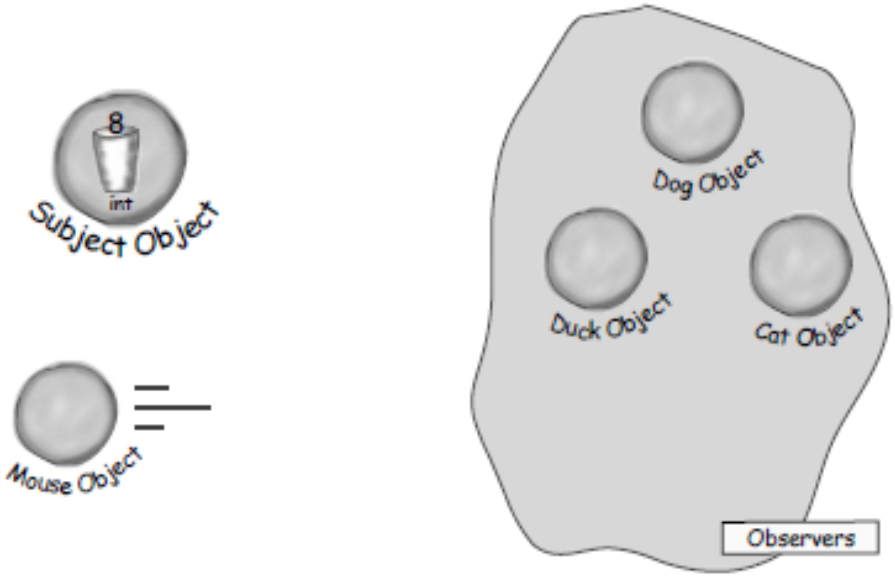



# Remove Observer ของตน

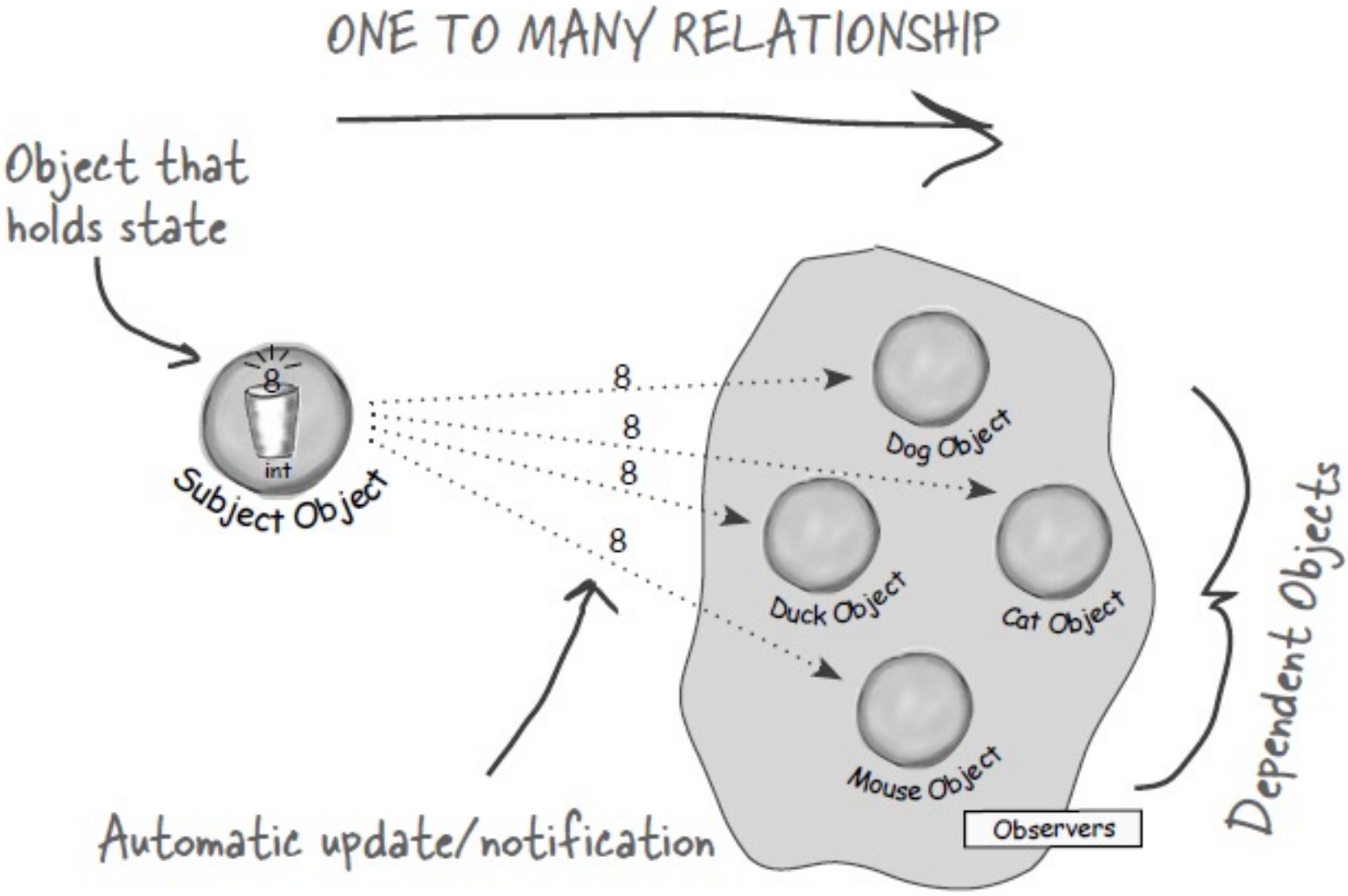
---

**Mouse is outta here!**

The Subject acknowledges the Mouse's request and removes it from the set of observers.



# 1-Many Relationship ต่อการ Observation 1 เรื่อง



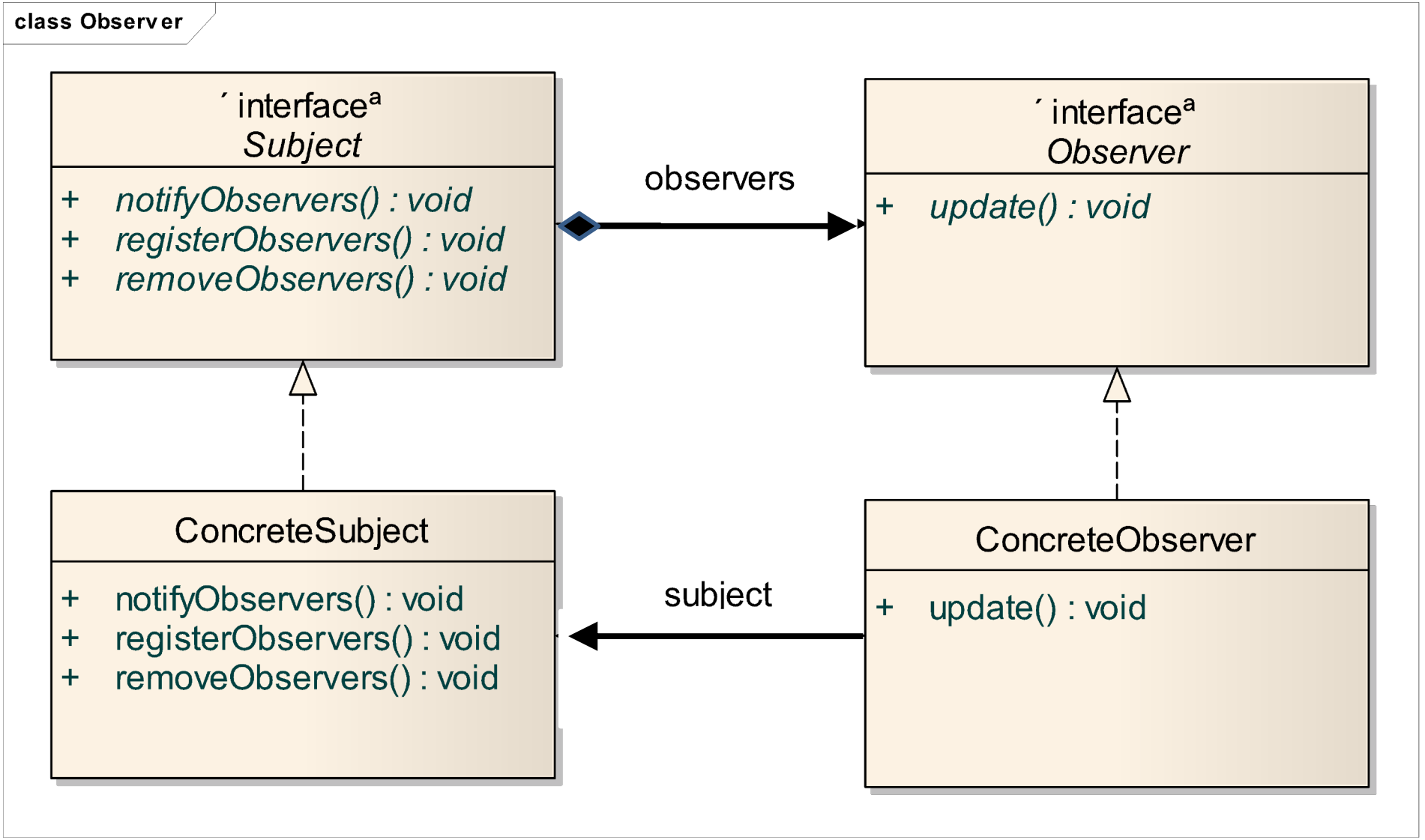
# Observer Definition

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

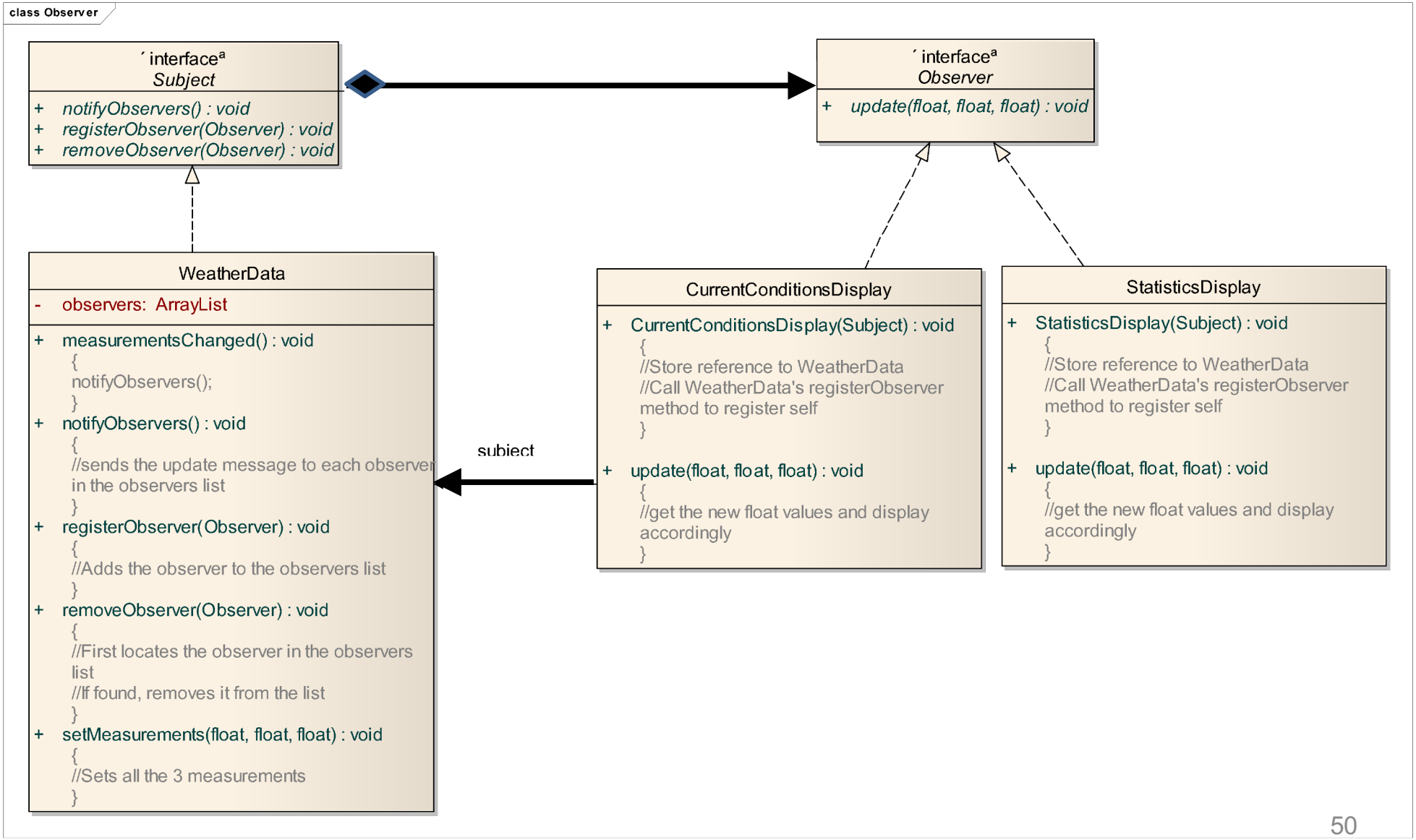
# Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact

# Observer – Class diagram



# Observer - Solution



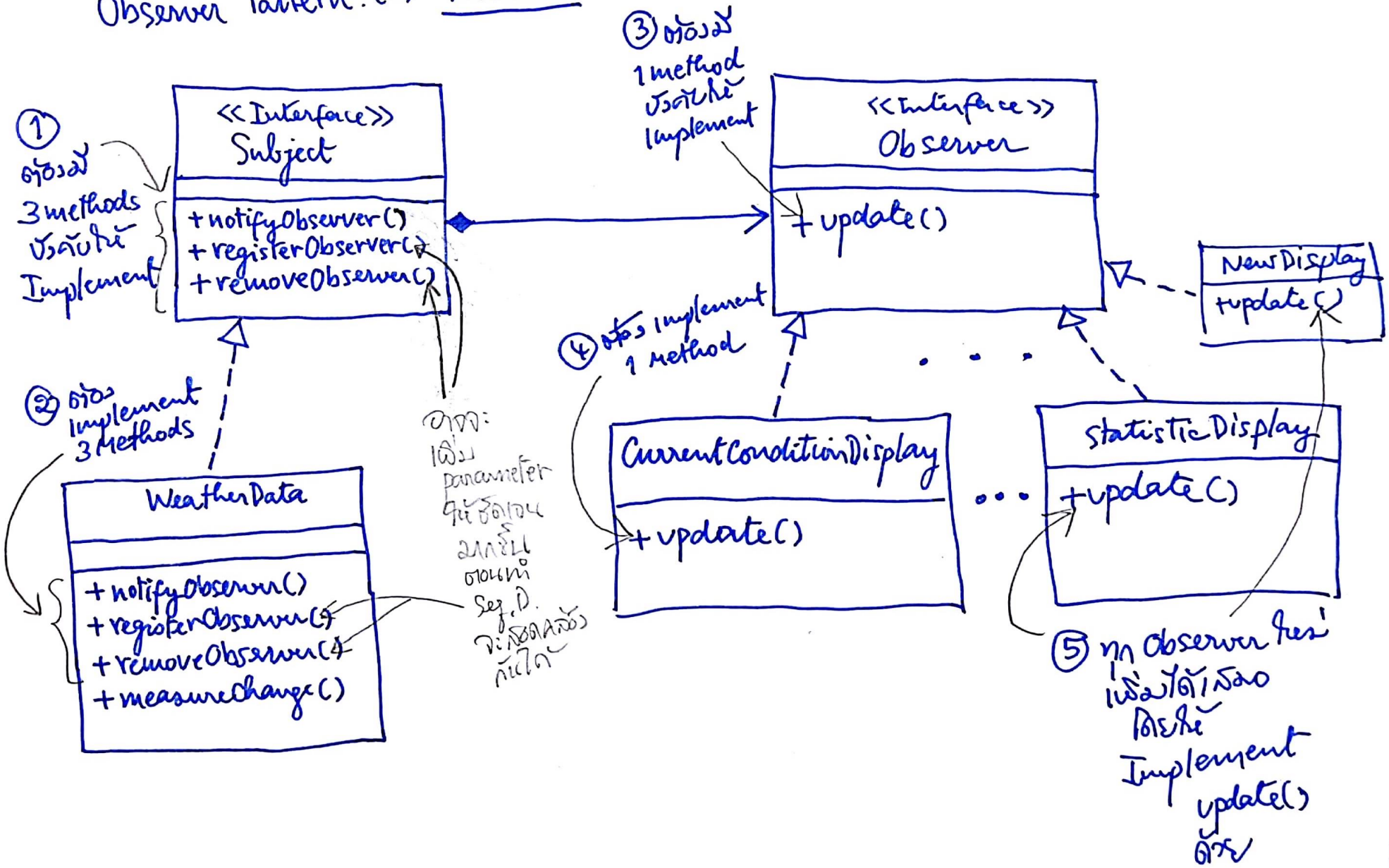
# Observer

- Pros
  - Abstracts coupling between Subject and Observer
  - Supports broadcast communication
  - Supports unexpected updates
  - Enables reusability of subjects and observers independently of each other
- Cons
  - Exposes the Observer to the Subject (with push)
  - Exposes the Subject to the Observer (with pull)

# โจทย์

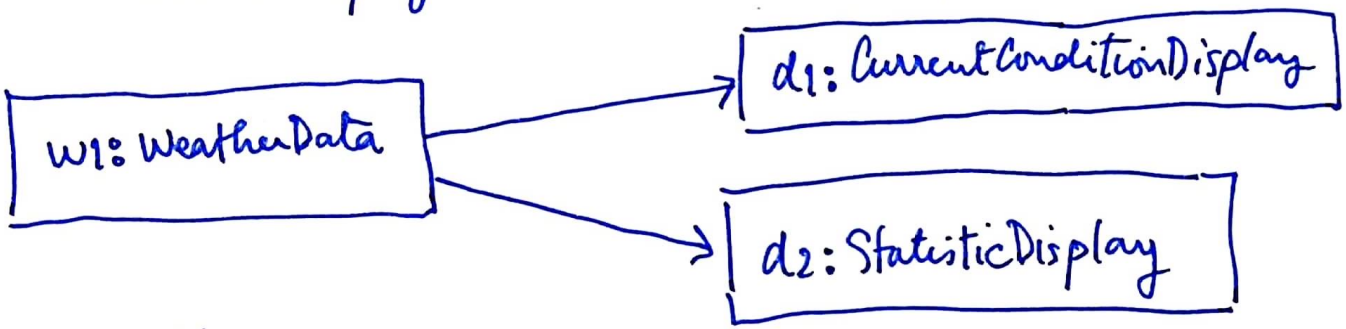
- จงวาดรูป Object diagram ของ w1:WeatherData ที่มี d1:CurrentConditionDevice, d2:StatisticDisplay
- จงวาดรูปตอน w1:WeatherData ทำการ registerObserver(d3:NewDisplay) เข้าร่วม
- จงวาดรูปตอน w1:WeatherData ทำการ notify โดยการ push data ไปยัง observers ของตนด้วยการเรียก update(float, float, float) ตามที่ตกลงกันไว้ใน Interface

# Observer Pattern. (1) Solution

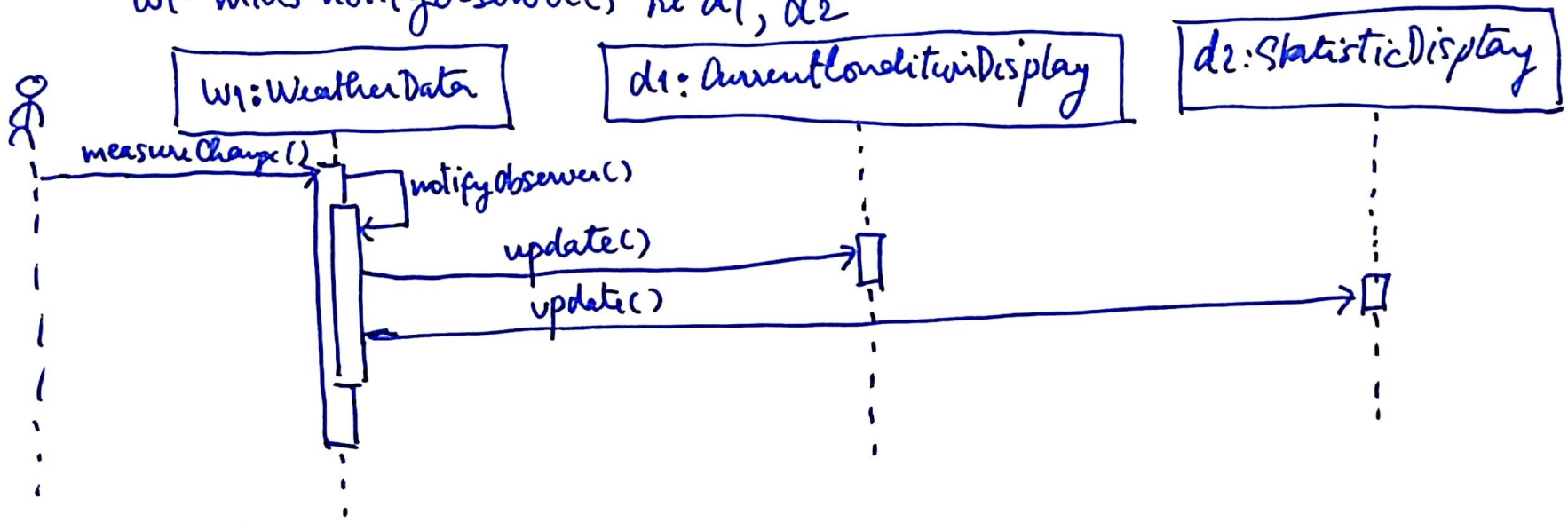


Observer Pattern (2)

# วาดรูป Object Diagram ของ w1: WeatherData ที่มีการ subject ไปยัง Observer 2 ตัว คือ
d1: CurrentConditionDisplay และ d2: StatisticDisplay

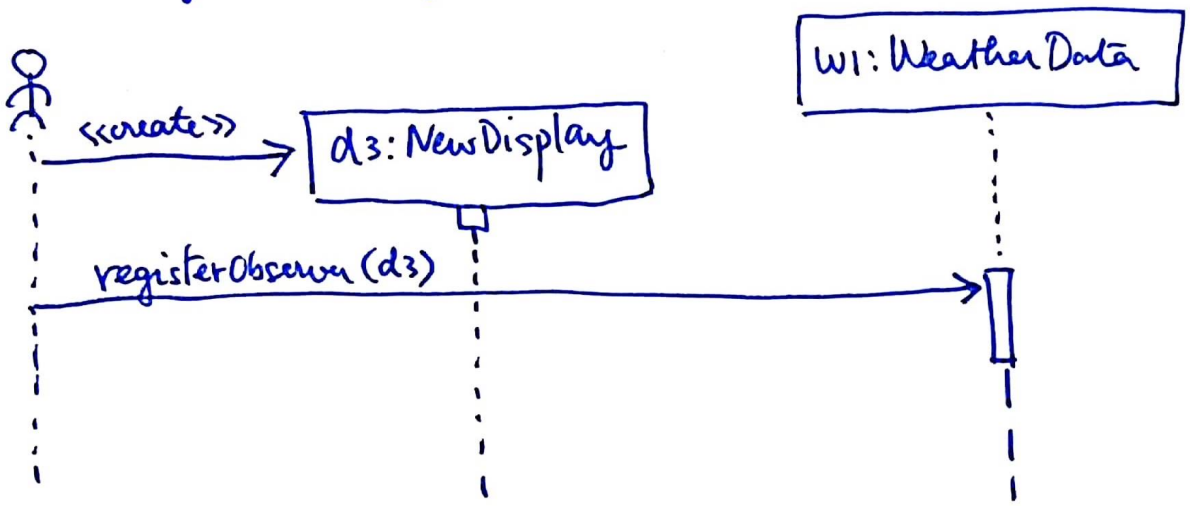



# วาดรูป Sequence Diagram ของรูป object Diagram (ข้างบน) ตอนที่มีการ
w1. เรียก notifyObserver() ให้ d1, d2

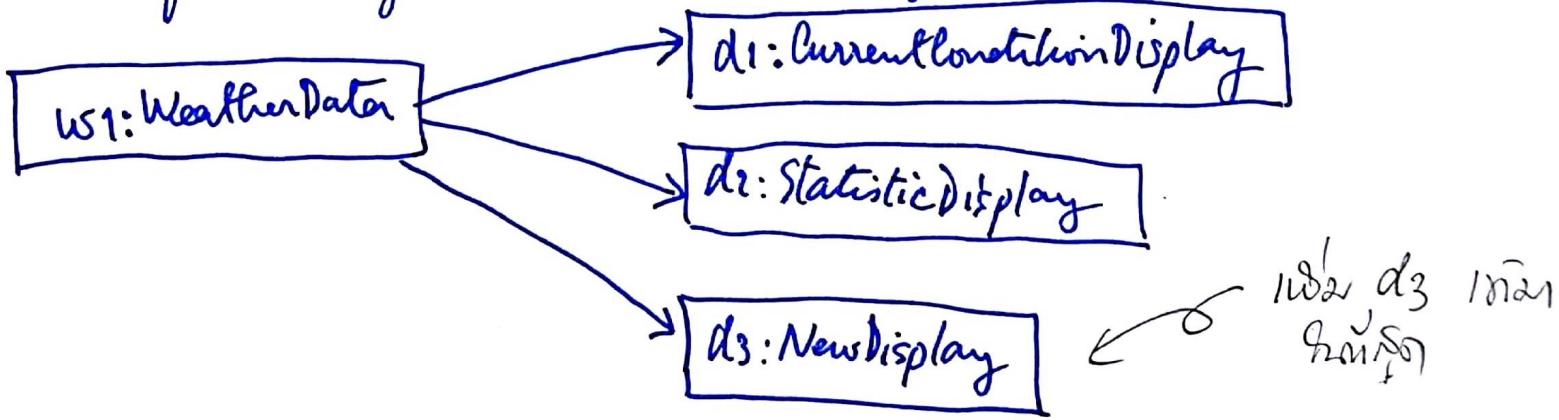


# Observer Pattern (3)

# วาดรูป Sequence Diagram ตอนเพิ่มการลง w1 รับ Observer เพิ่ม ชื่อ d3:NewDisplay




# ผลลัพธ์จาก Sequence Diagram (ด้านบน) จะได้ Object Diagram เป็นแบบนี้



# Exercise

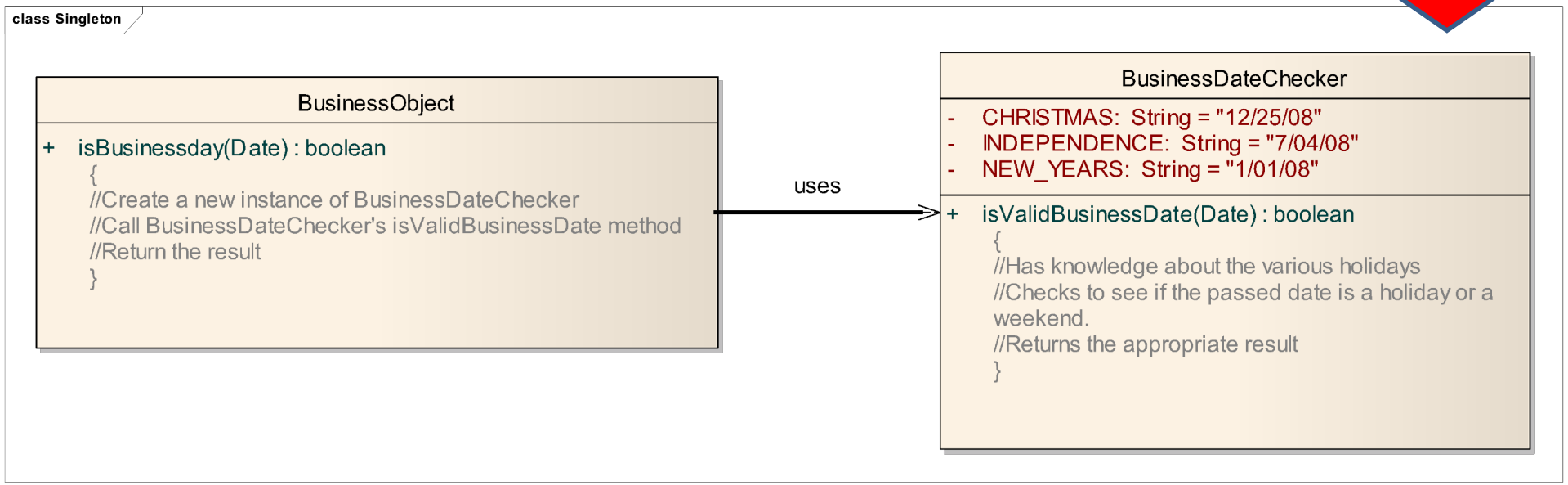
- จงออกแบบ Class diagram ที่มี Subject เป็นราคาน้ำมัน และมีหน้าจอ WebPage1 แสดงราคาน้ำมัน
  มีหน้าจอ MobilePage1 แสดงราคาน้ำมัน
  และสามารถเพิ่มหน้าจอที่จะนำราคาน้ำมันไปแสดงได้เรื่อย ๆ
  โดยเมื่อราคาน้ำมันเปลี่ยนจะ Push data ใหม่ส่งไปให้จอภาพเสมอ
- ทดลองเปลี่ยนโจทย์จากราคาน้ำมัน ไปเป็นอัตราดอกเบี้ย หรือราคาหุ้น

# Singleton Definition

Ensure a class only has one instance and provide a global point of access to it.

# Singleton - Problem

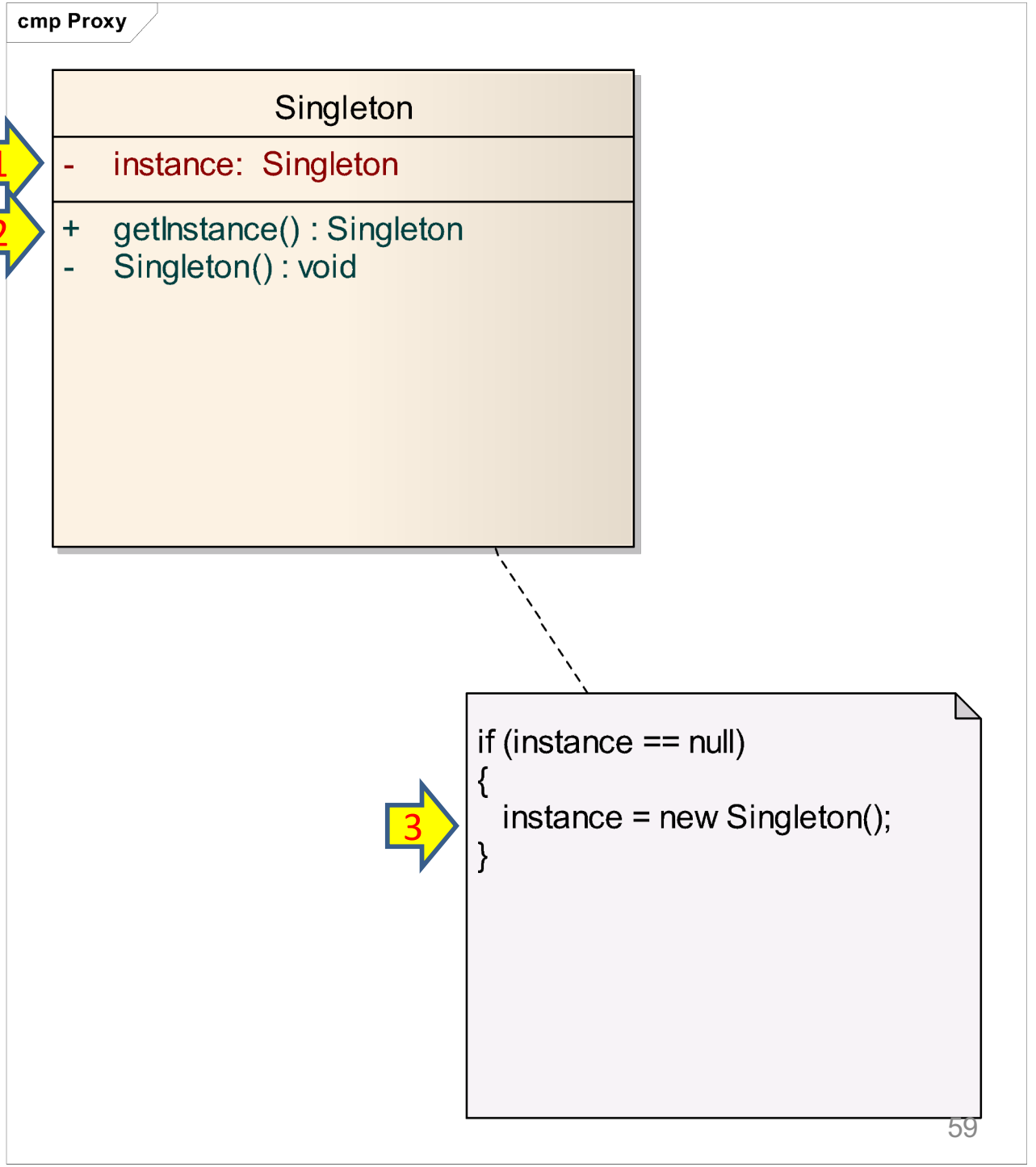
ควรเป็น Singleton




BusinessDateChecker เป็น Class ที่ตรวจว่าวันใดเป็นวันหยุด ซึ่งถ้าเราเรียกใช้เราก็จะต้อง new BusinessDateChecker() เสมอ ทำให้มี Object เกิดใหม่มากมาย ทั้งที่ใช้ร่วมกันได้

# Singleton

1) มี private var. ที่เก็บ object ที่เคยสร้างไว้หนึ่งเดียวนั้น
2) ไม่ให้ Caller Instantiate object โดยตรง ให้ทำผ่าน public static method ที่ชื่อ getInstance()
3) getInstance() จะตรวจดูว่ามี object ที่เคยสร้างไว้ไหม ถ้าไม่มีให้สร้างใหม่ ถ้ามีแล้วให้ใช้ตัวเดิมที่มี



# Singleton

cmp Proxy

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton() {
      //Exists only to defeat instantiation.
    }

    public static Singleton getInstance() {
      if(instance == null) {
        instance = new Singleton();
      }

      return instance;
}
```

1
2
2
3

```
public class SingletonInstantiator {
   public SingletonInstantiator() {
     Singleton instance = Singleton.getInstance();
     Singleton anotherInstance = new Singleton();
     .....
}
```

# Singleton

- Pros
  - Increases performance
  - Prevents memory wastage
  - Increases global data sharing
- Cons
  - Results in multithreading issues

# Patterns & Definitions – Group 1

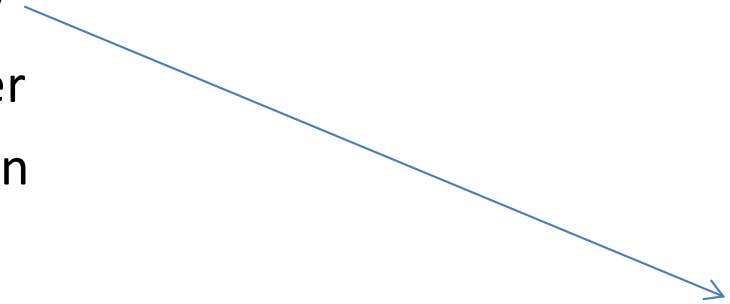
- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

ลากเส้นความสัมพันธ์ระหว่างทั้งสอง columns

# Patterns & Definitions – Group 1

- Strategy
- Observer
- Singleton

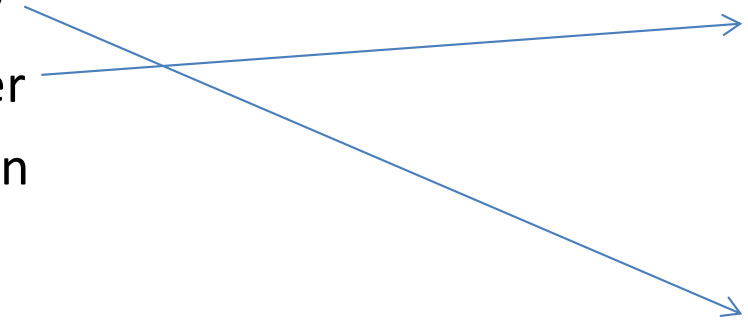


- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

ลากเส้นความสัมพันธ์ระหว่างทั้งสอง columns

# Patterns & Definitions – Group 1

- Strategy
- Observer
- Singleton

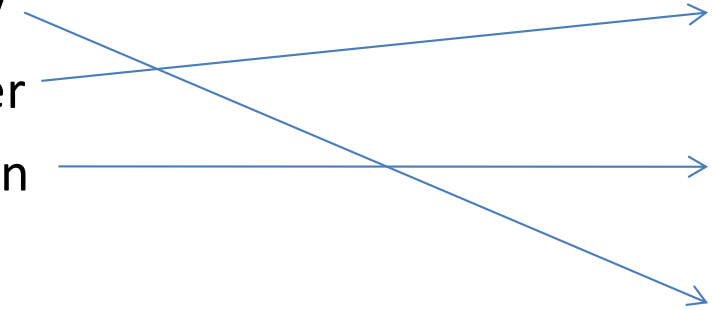


- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

ลากเส้นความสัมพันธ์ระหว่างทั้งสอง columns

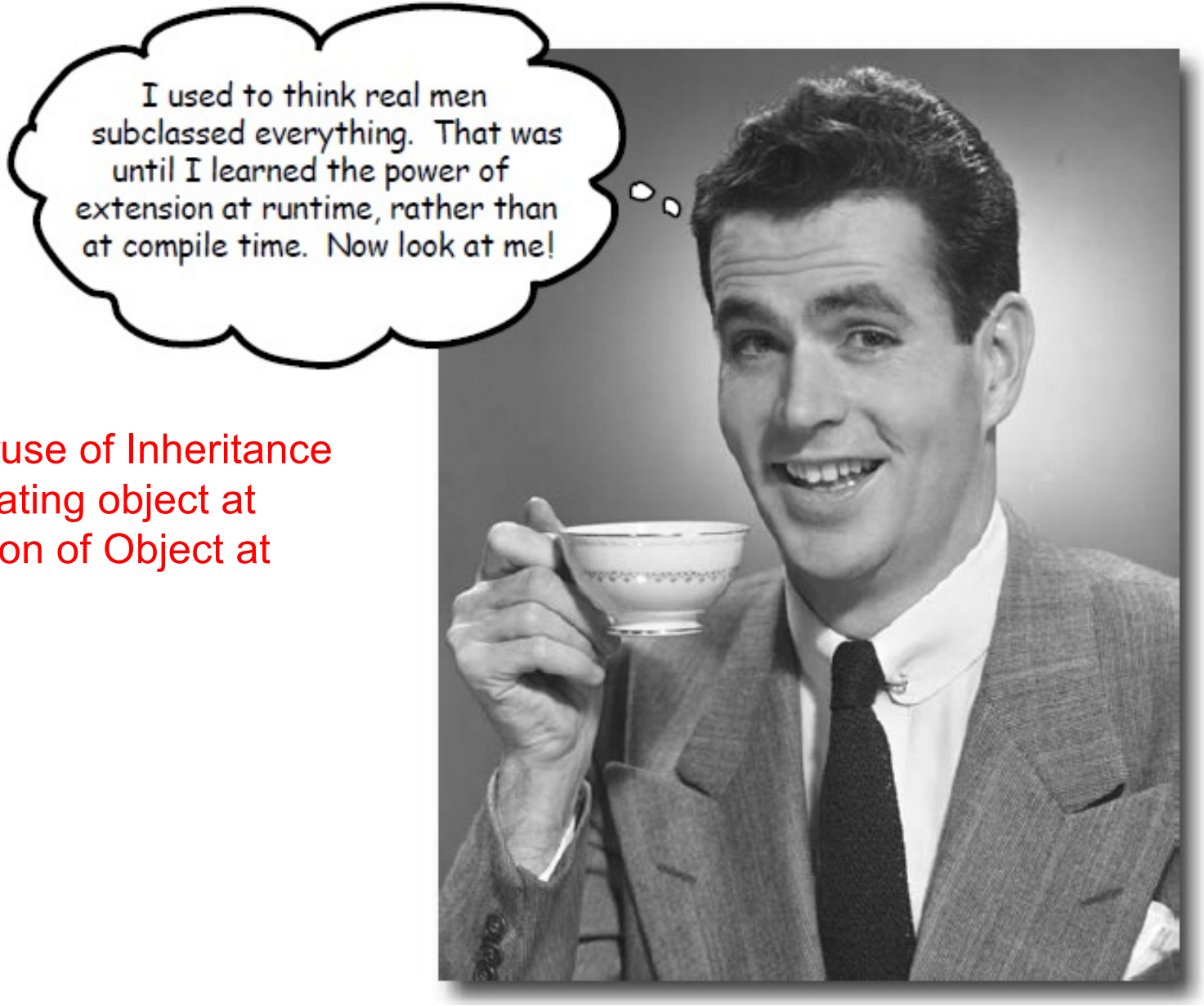
# Patterns & Definitions – Group 1

- Strategy
- Observer
- Singleton



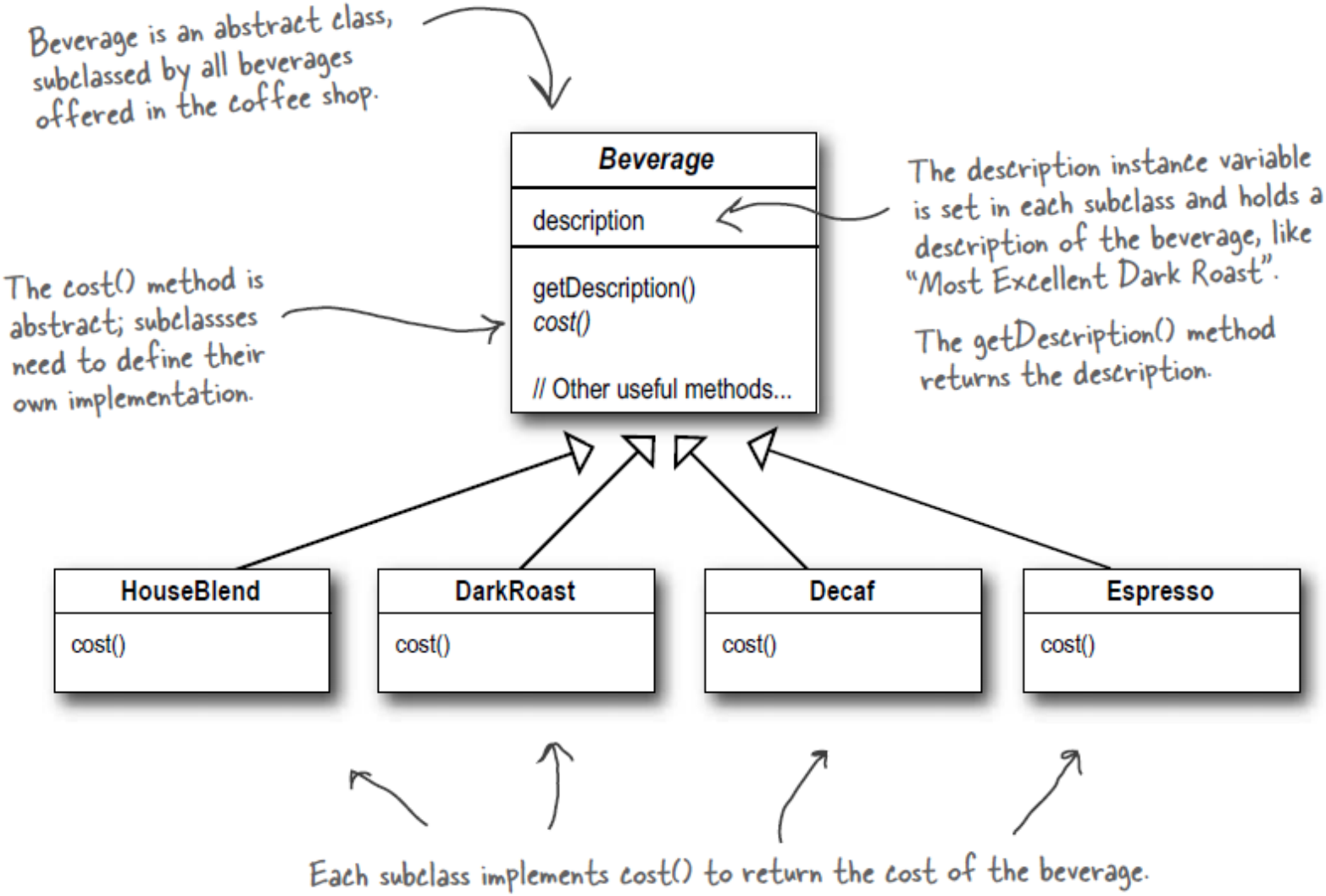
- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

ลากเส้นความสัมพันธ์ระหว่างทั้งสอง columns




Problem of Overuse of Inheritance
Solved by decorating object at runtime (Extension of Object at runtime!!)

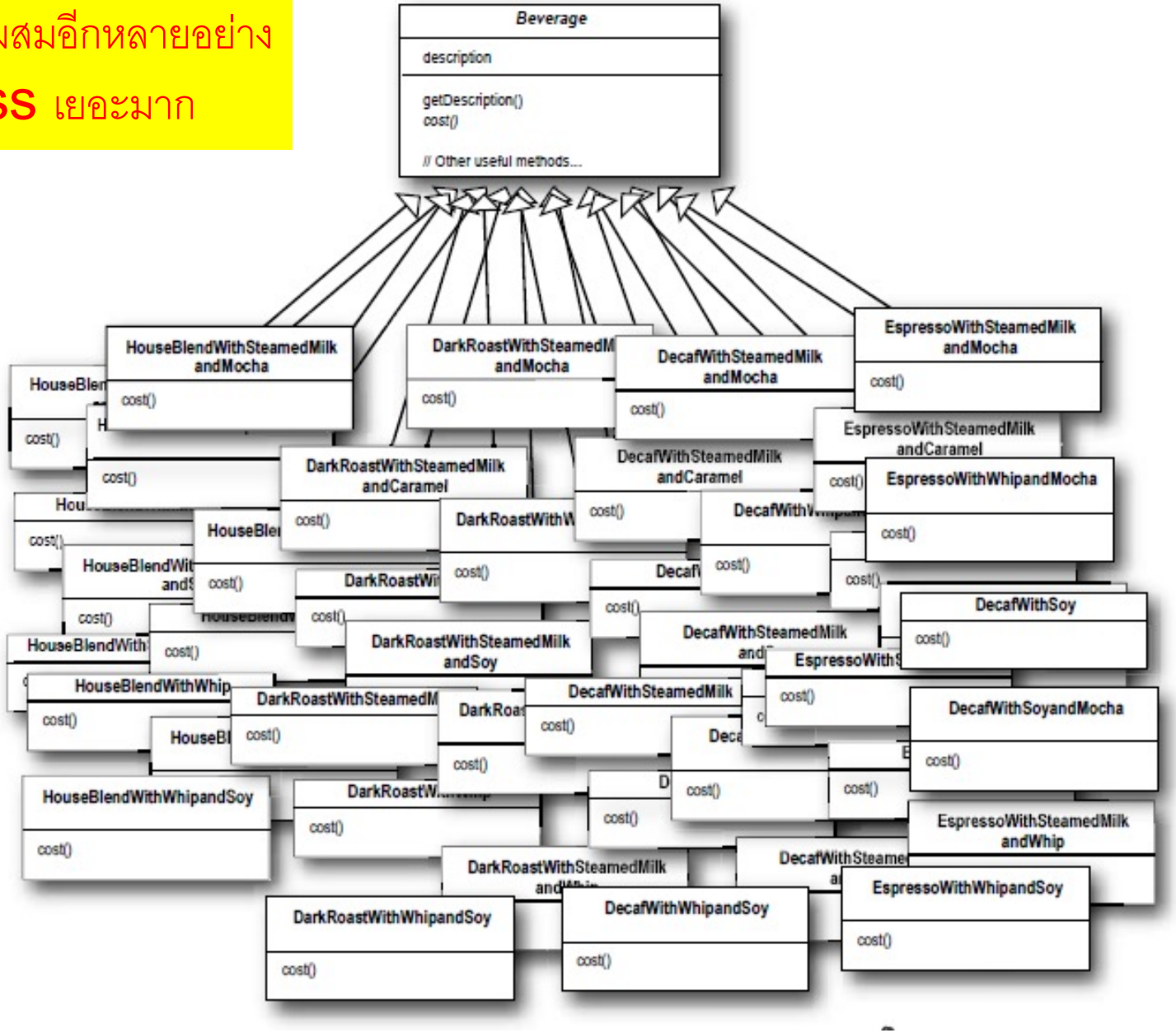
# Welcome to Starbuzz Coffee



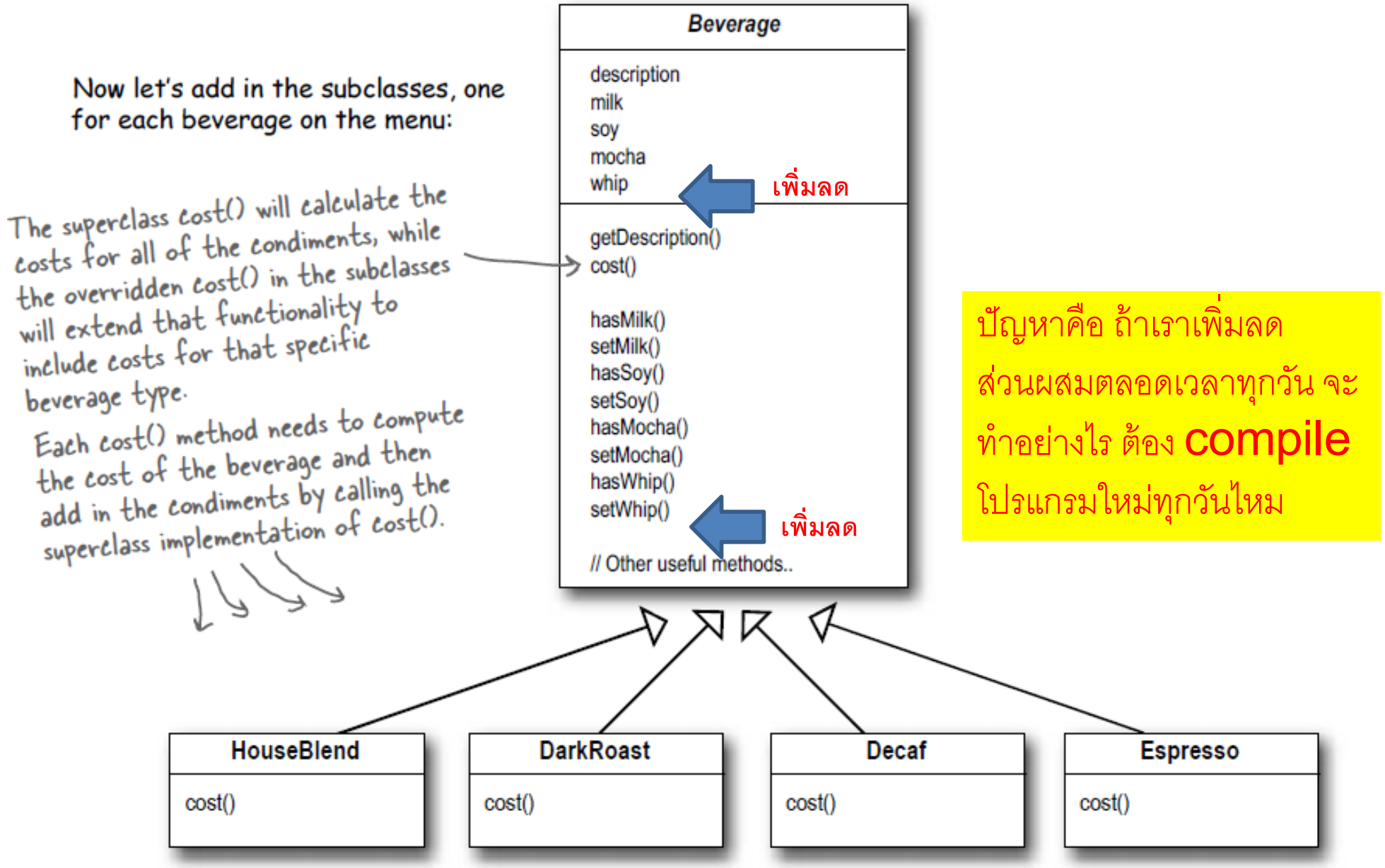

ถ้ากาแฟในร้านมีส่วนผสมให้เลือกมากขึ้น และเพิ่มได้ตามชอบใจ เราจะออกแบบอย่างไร

# Overuse of Inheritance

กาแฟหลายชนิด ใส่ส่วนผสมอีกหลายอย่าง
ทำให้ต้องใช้ **subclass** เยอะมาก



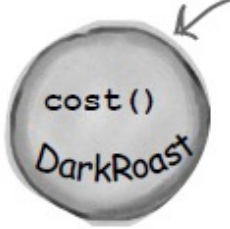
# แก้ปัญหาโดยใช้ attributes and methods สำหรับส่วนผสม 1 ชนิด




ปัญหาคือ ถ้าเราเพิ่มลดส่วนผสมตลอดเวลาทุกวัน จะทำอย่างไร ต้อง compile โปรแกรมใหม่ทุกวันไหม

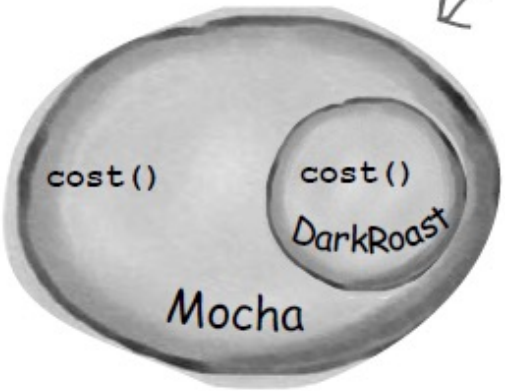
# การเกิดของกาแฟหนึ่งแก้ว

**1 We start with our DarkRoast object.**




Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

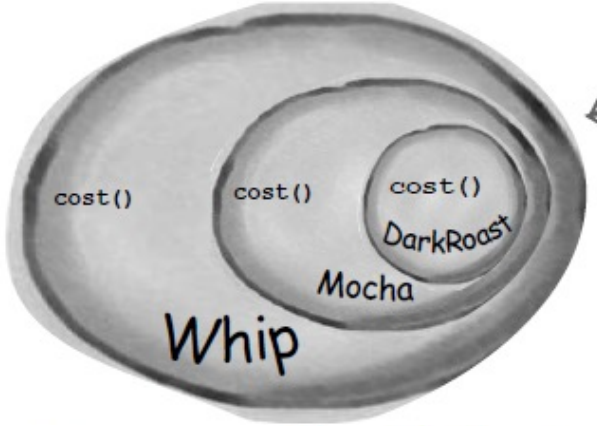
**2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**




The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

**3** **The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**




Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

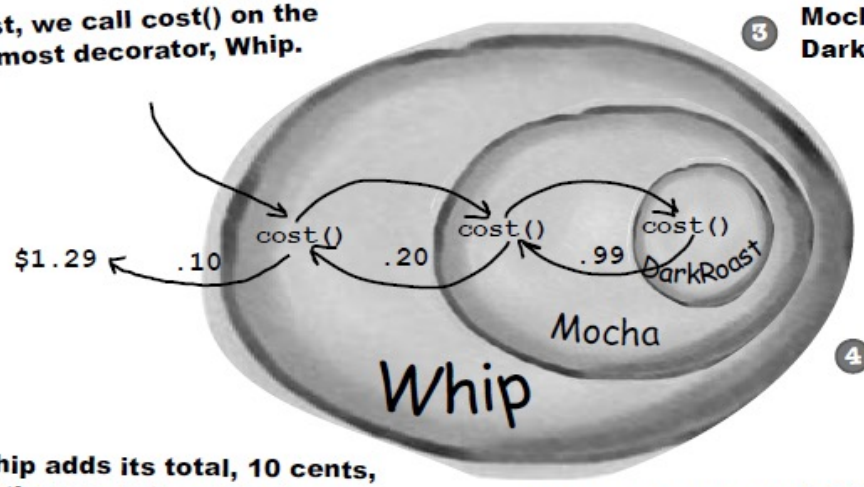
**4** **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**

(You'll see how in a few pages.)

**2** Whip calls cost() on Mocha.

**1** First, we call cost() on the outmost decorator, Whip.

**3** Mocha calls cost() on DarkRoast.




**4** DarkRoast returns its cost, 99 cents.

**5** Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

**5** Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

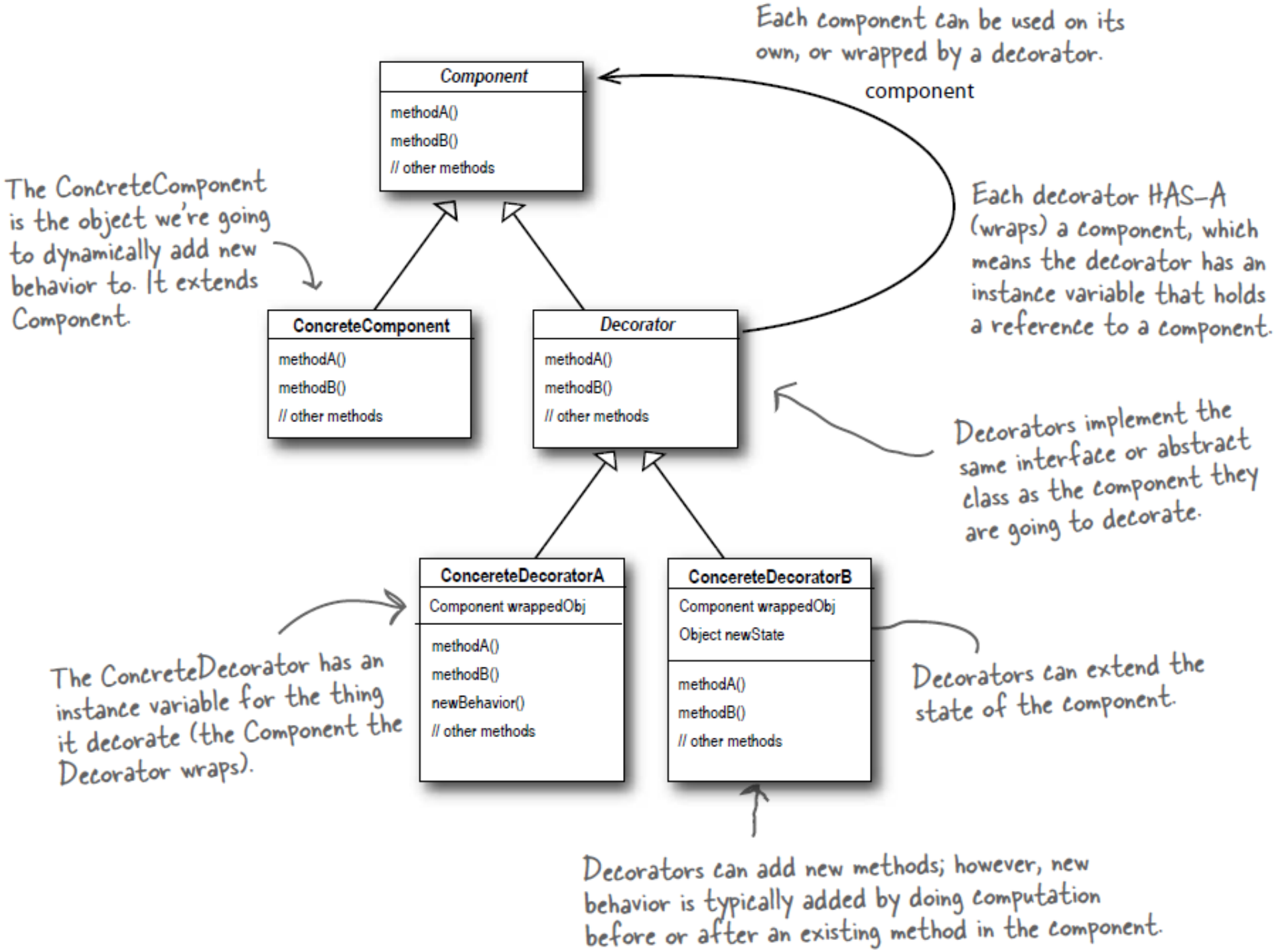
# Decorator Definition

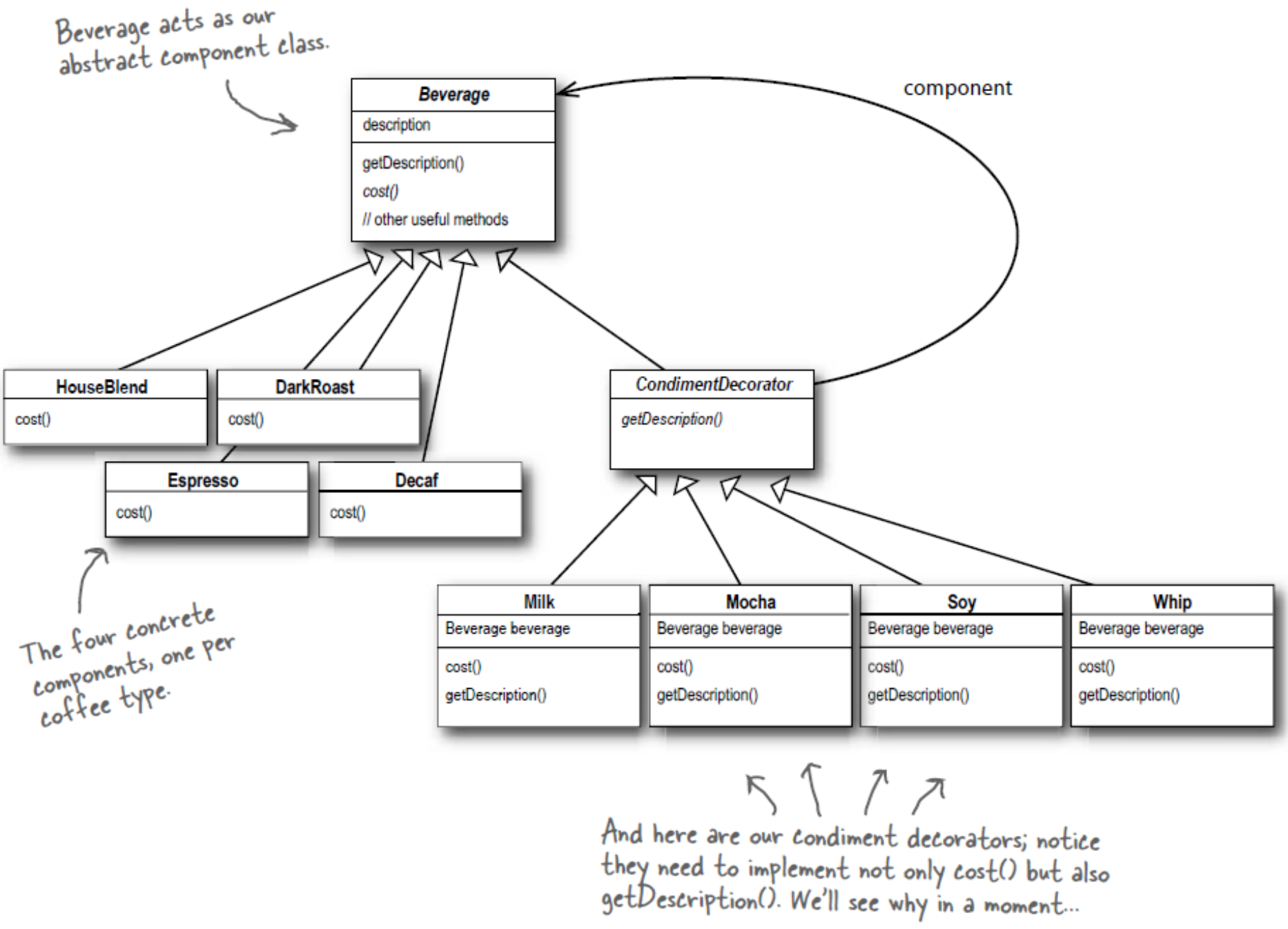
Attaches additional responsibilities to an object dynamically.

Decorators provide a flexible alternative to sub-classing for extending functionality.

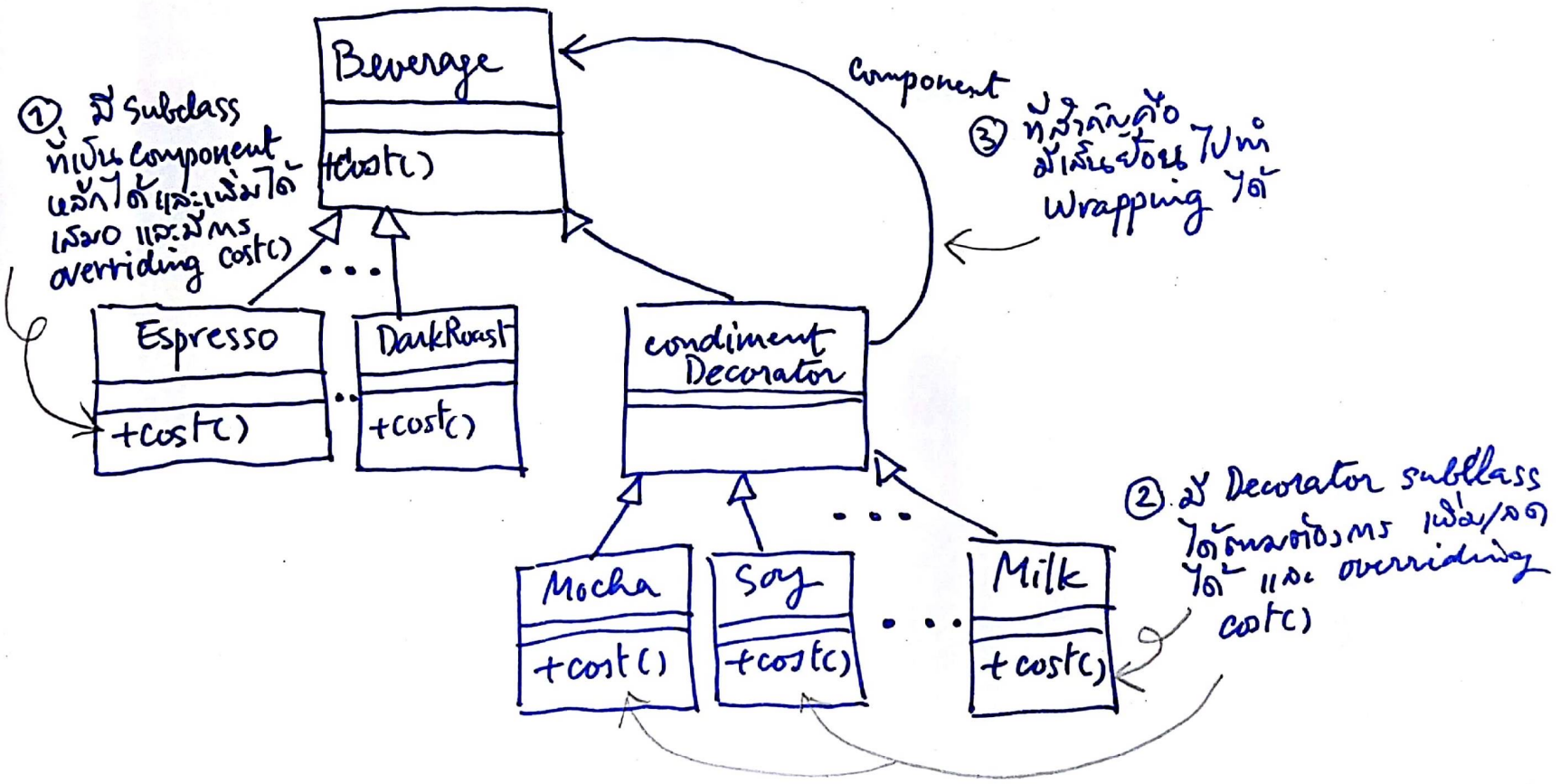
# Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification


Each component can be used on its own, or wrapped by a decorator.
Component
methodA()
methodB()
// other methods
component
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.


Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
The four concrete components, one per coffee type.
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

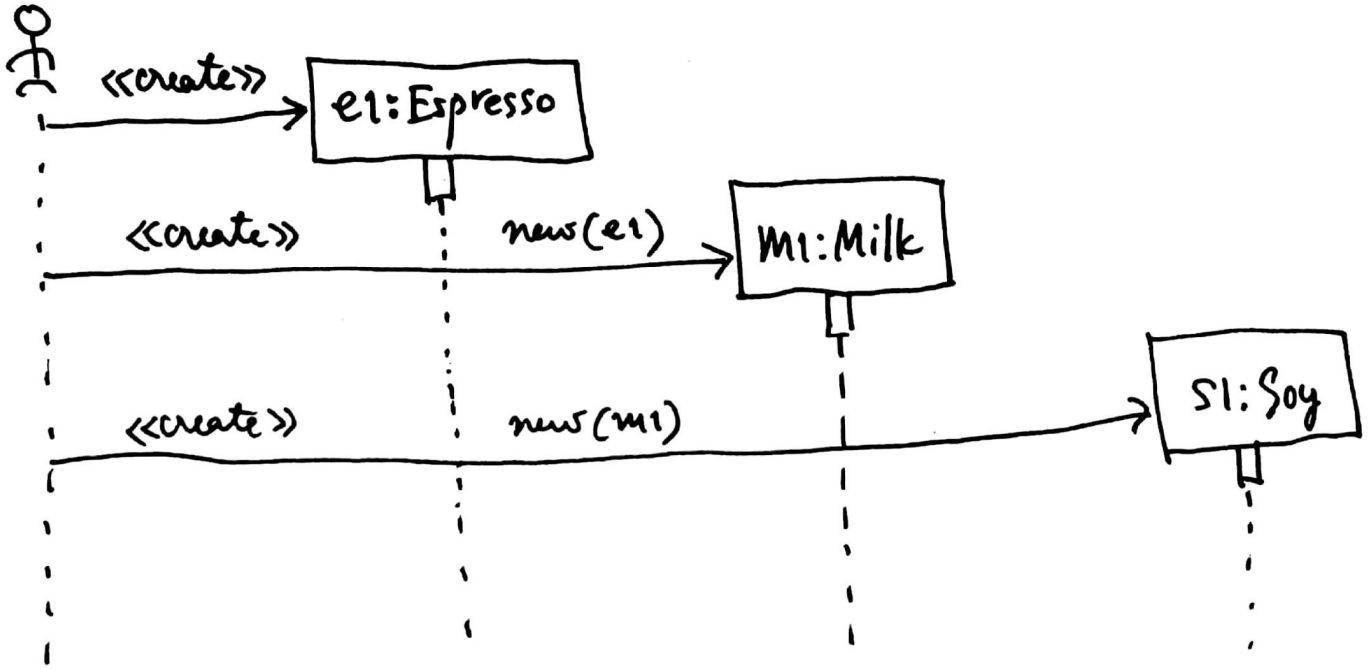
# Decorator Pattern (1) (Solution)



Decorator Pattern (2)

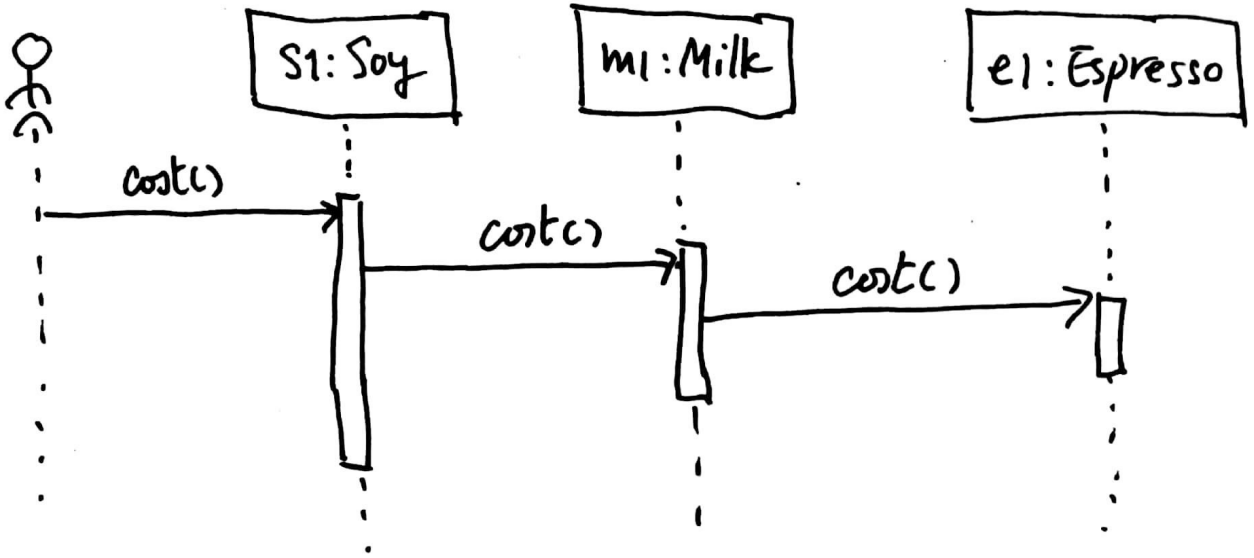
# จงวาดรูป object Diagram ของการสั่งกาแฟ e1:Espresso โดยเพิ่มส่วนผสม m1:Milk และ s1:Soy ด้วย.


# จงวาดรูป Sequence Diagram ของขั้นตอนการสั่งกาแฟแก้วนี้ (ทั้งหมด).



Decorator Pattern (3)

# ตัวอย่าง Sequence Diagram ของการสั่งเครื่องดื่มที่มีส่วนผสมเพิ่มเติม (ต่อ)



# Decorator

- Pros
  - Extends class functionality at runtime
  - Helps in building flexible systems
  - Works great if coded against the abstract component type

- Cons
  - Results in problems if there is code that relies on the concrete component's type

# Facade Definition

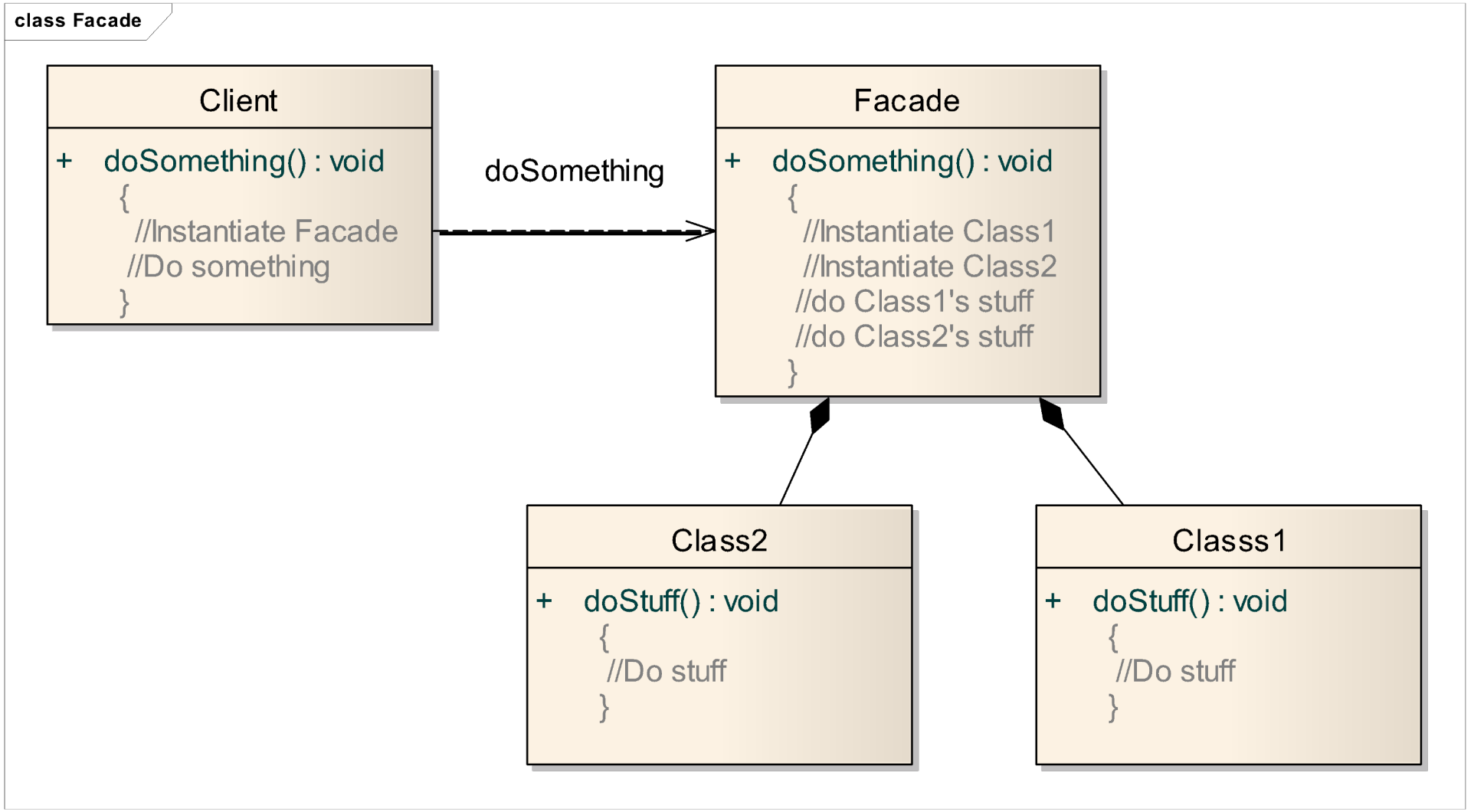
Provides a unified interface to a set of interfaces in a subsystem.

Façade defines a higher level interface that makes the subsystem easier to use.

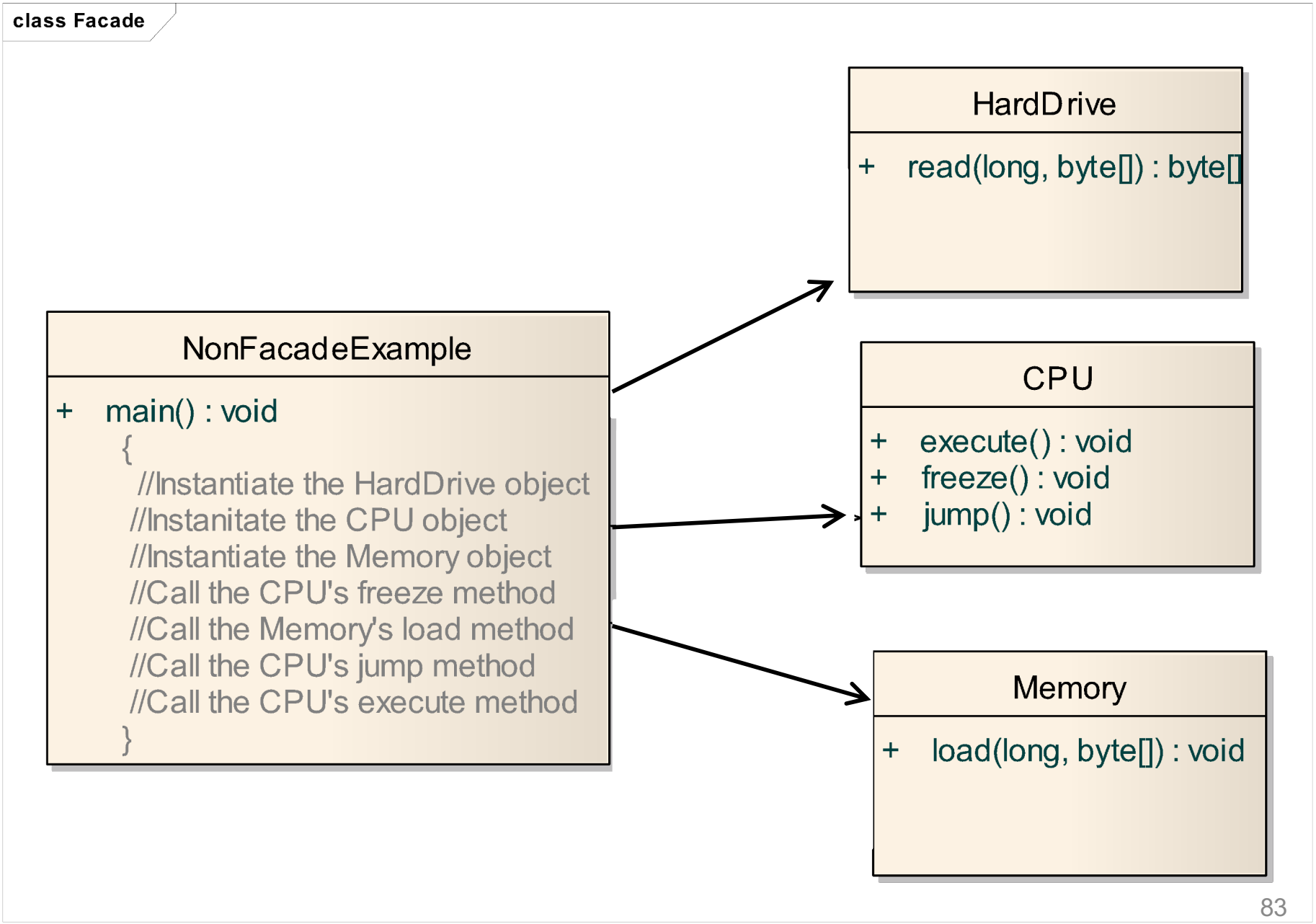
# Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Principle of least knowledge – talk only to your immediate friends

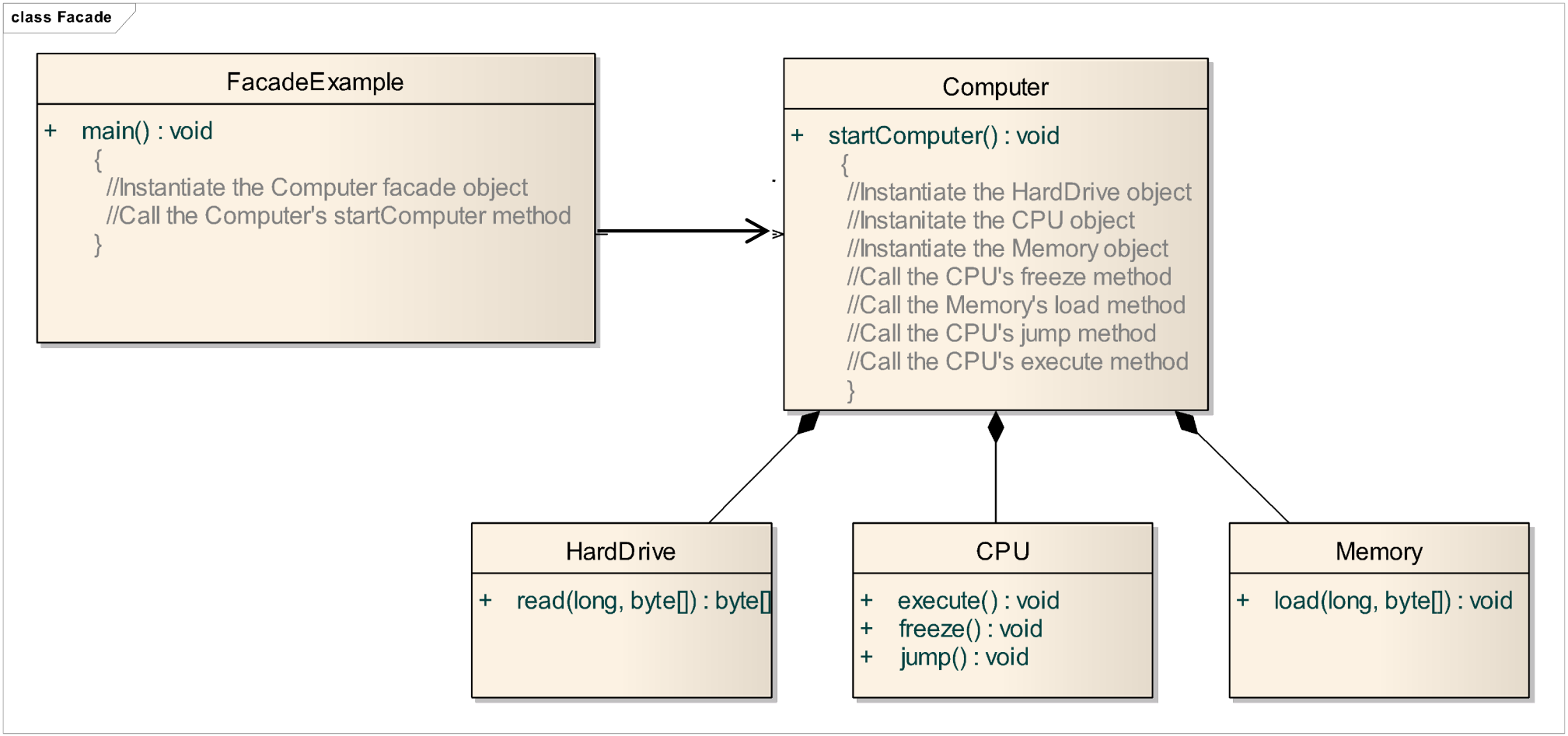
# Façade – Class diagram



# Façade - Problem



# Façade - Solution

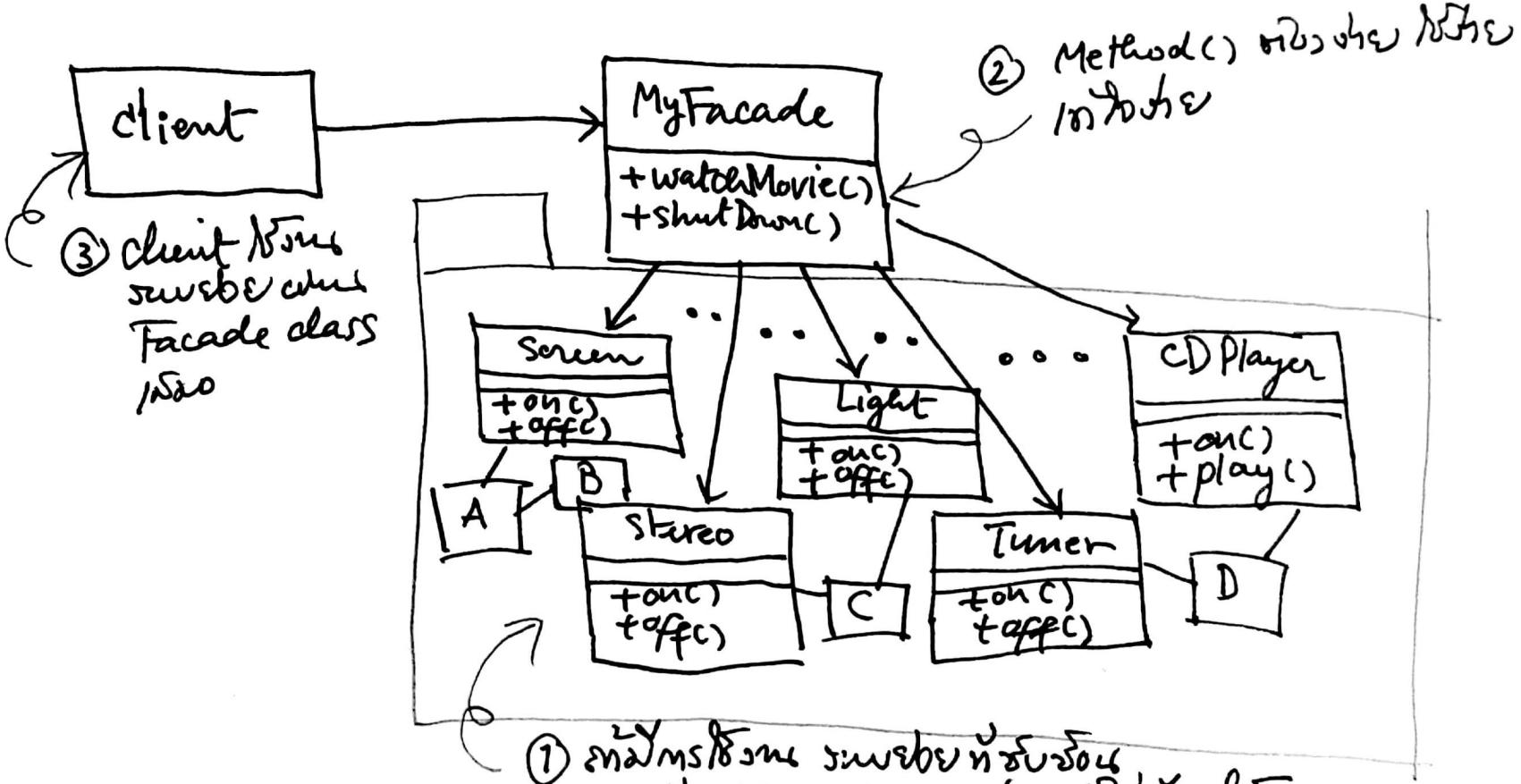


# Facade

- Pros
  - Makes code easier to use and understand
  - Reduces dependencies on classes
  - Decouples a client from a complex system

- Cons
  - Results in more rework for improperly designed Façade class
  - Increases complexity and decreases runtime performance for large number of Façade classes

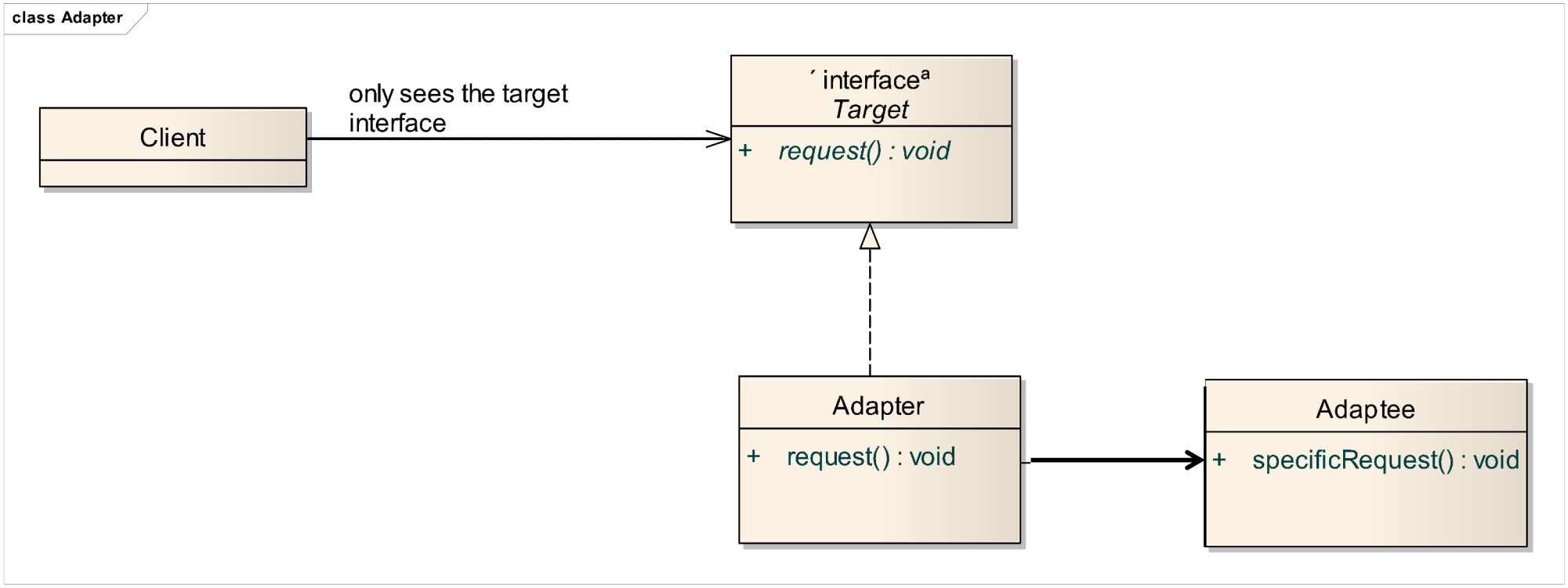
# Facade Pattern (1) (Solution)



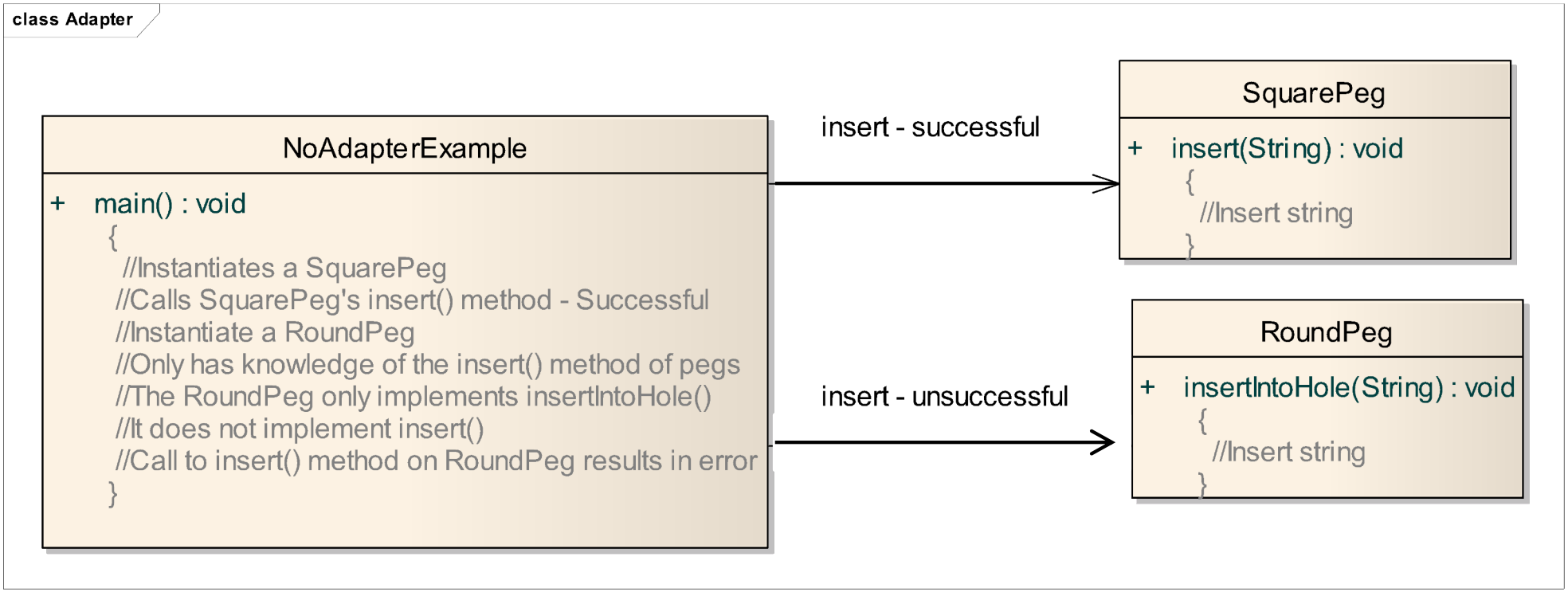
# Adapter Definition

Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

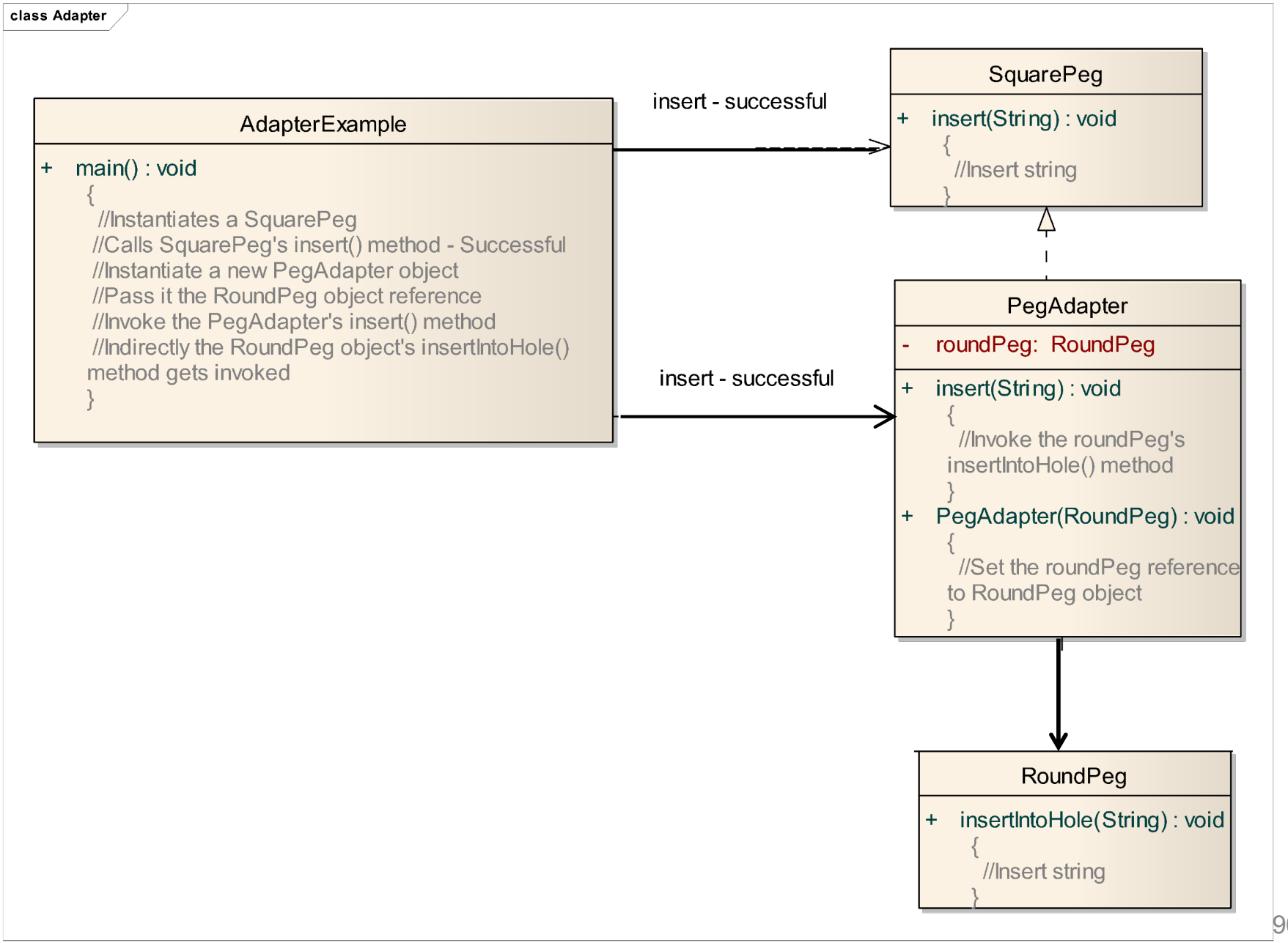
# Adapter – Class diagram



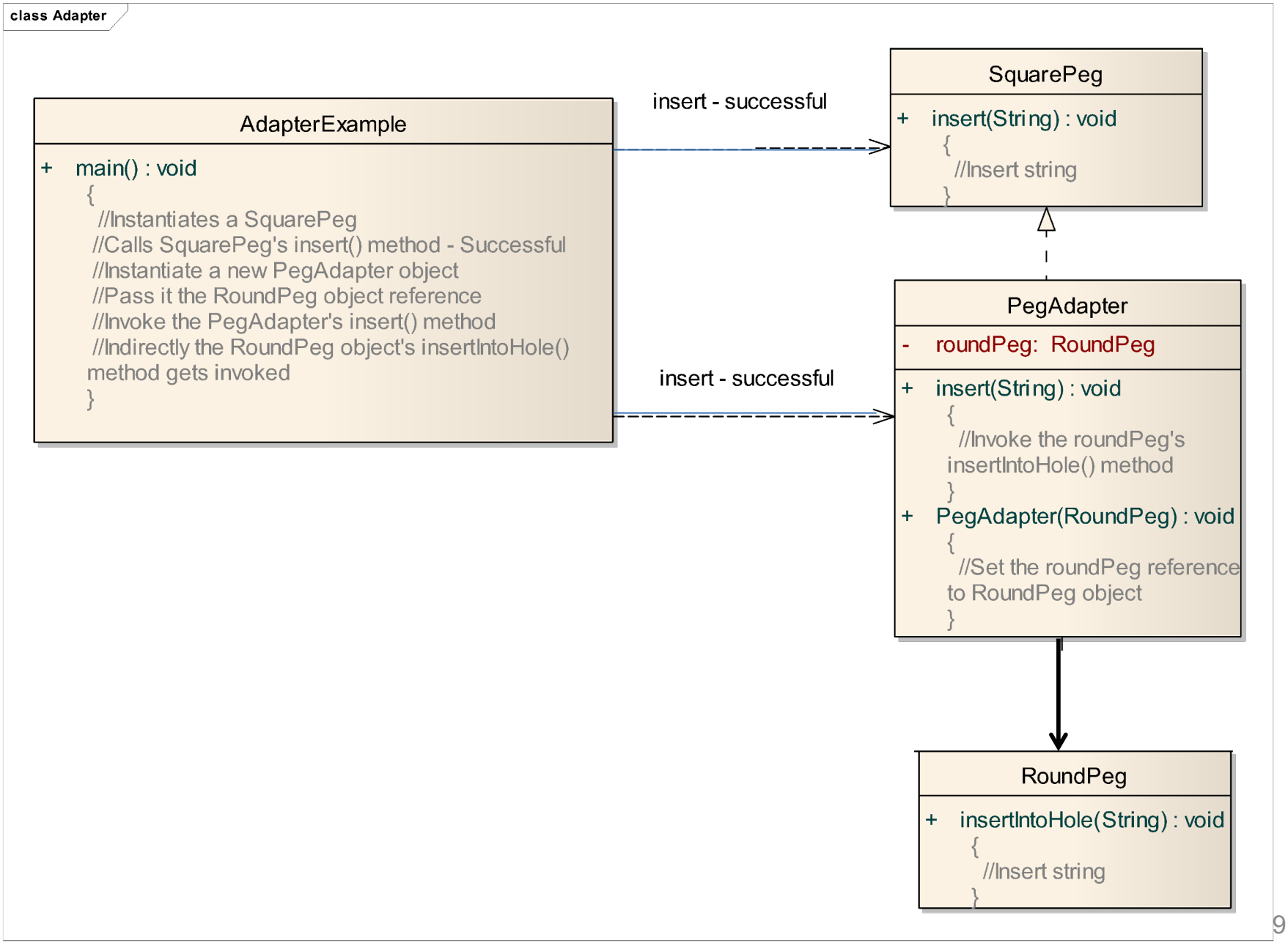
# Adapter - Problem



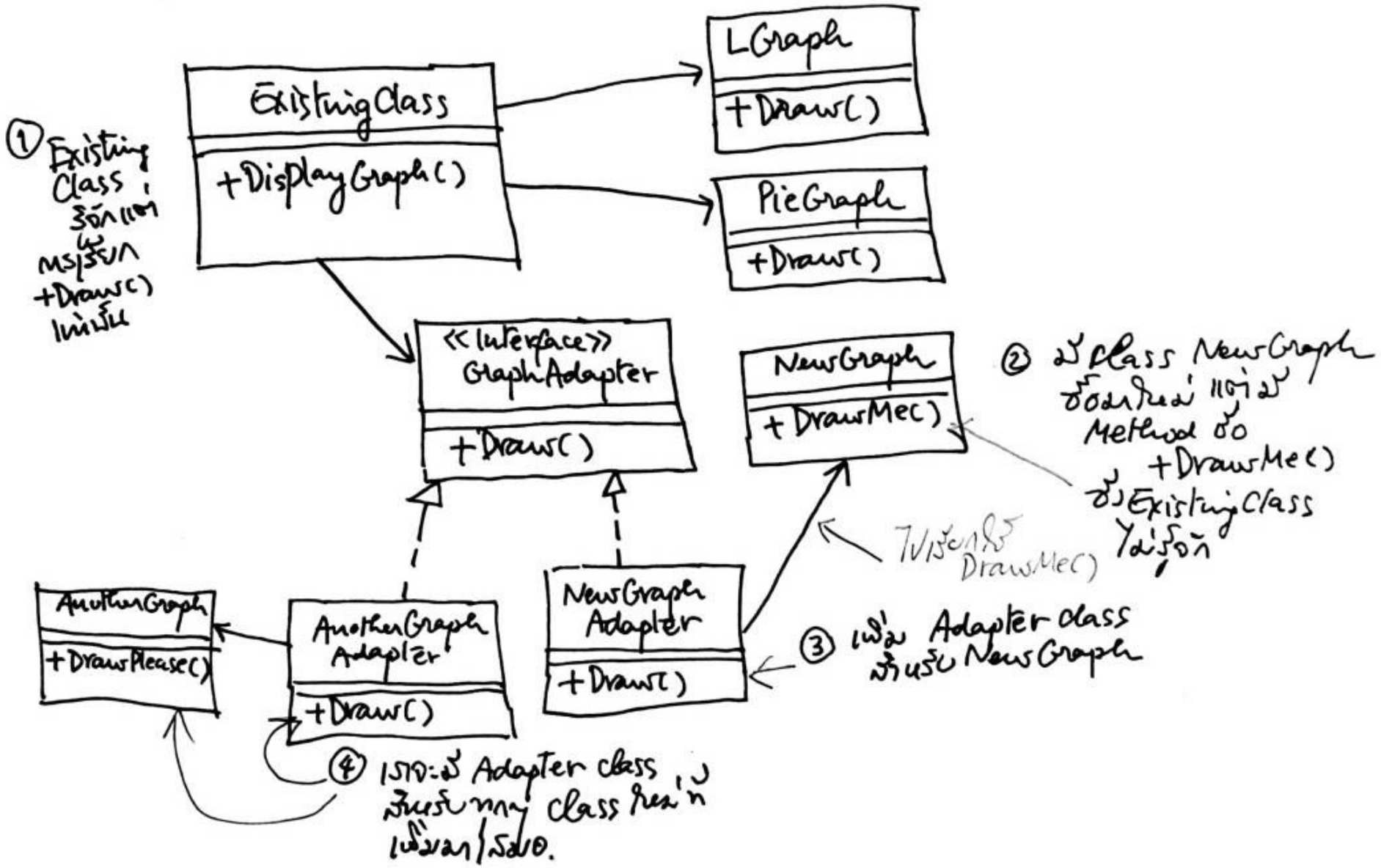
# Adapter - Solution



# Adapter - Solution

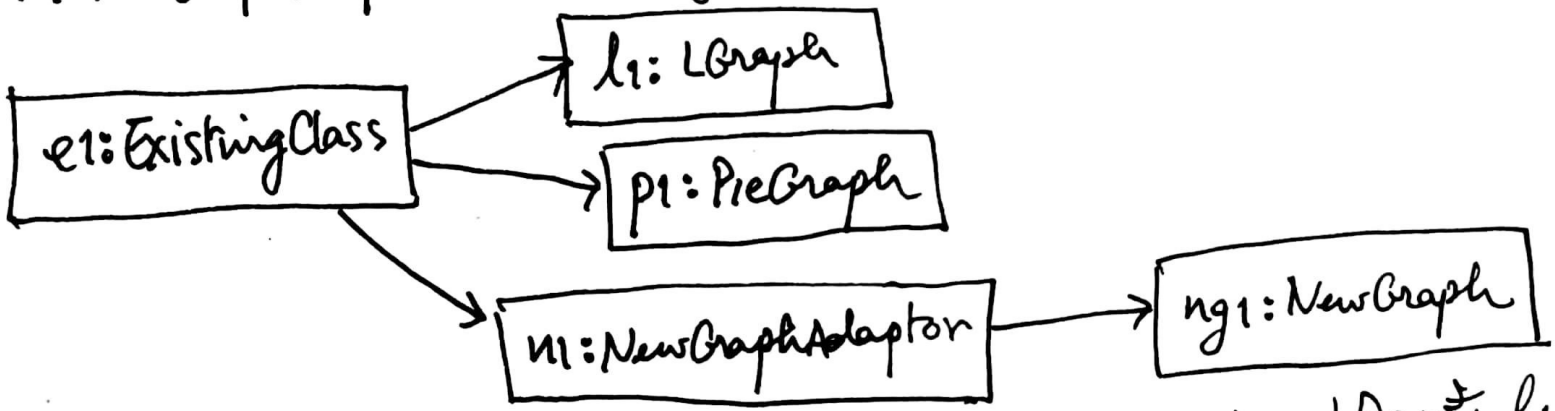


# Adapter Pattern (1) (Solution)

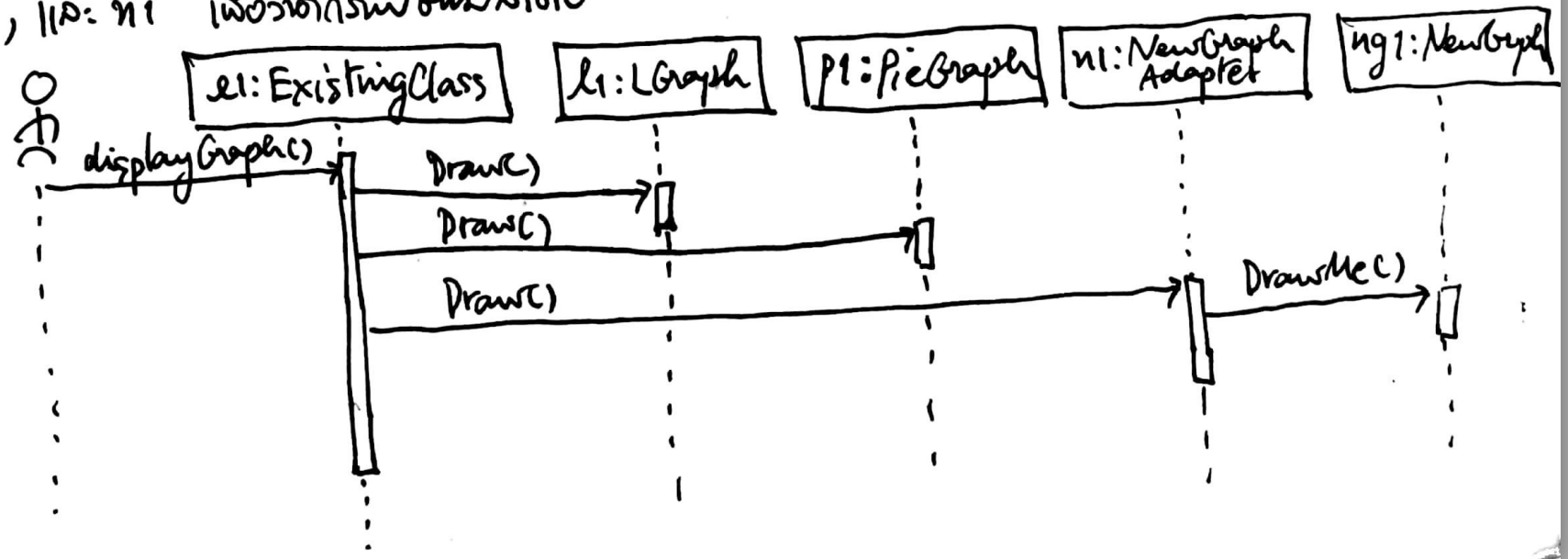


# Adapter Pattern (2)

# จงวาด Object Diagram ของ e1:ExistingClass ที่ทำงาน l1:LGraph, p1:PieGraph และ n1:NewGraphAdapter พร้อม ng1:NewGraph




# จงใช้ Sequence Diagram อธิบายการทำงานของ e1:ExistingClass วาดรูปกราฟวงกลม l1, p1, และ n1 เพื่อวาดกราฟเส้นตามลำดับ



# Adapter

- Pros
  - Increases code reuse
  - Encapsulates the interface change
  - Handles legacy code

- Cons
  - Increases complexity for large number of changes

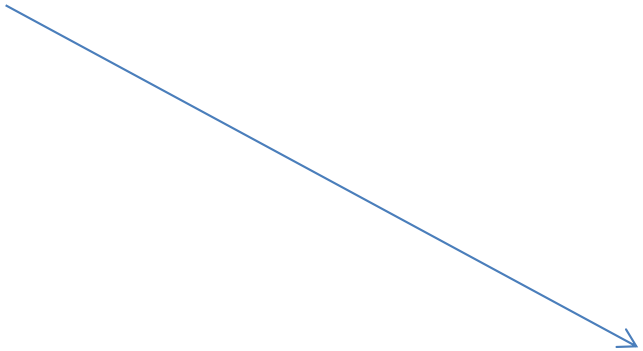
# Patterns & Definitions – Group 2

- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

# Patterns & Definitions – Group 2

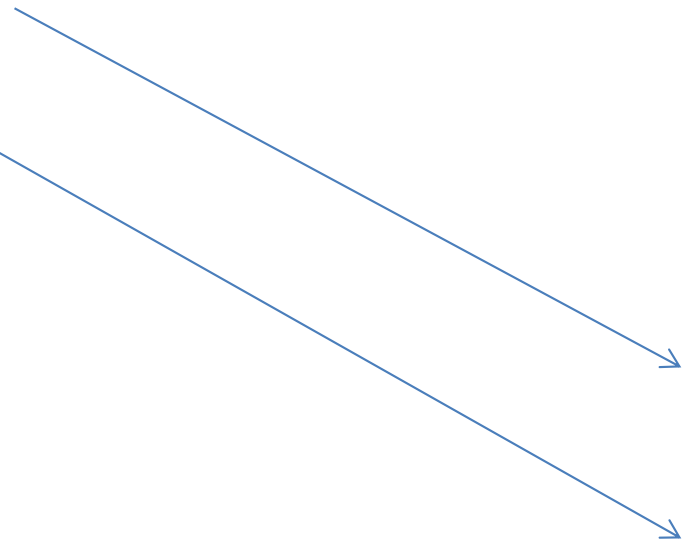
- Decorator
- Proxy
- Façade
- Adapter



- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

# Patterns & Definitions – Group 2

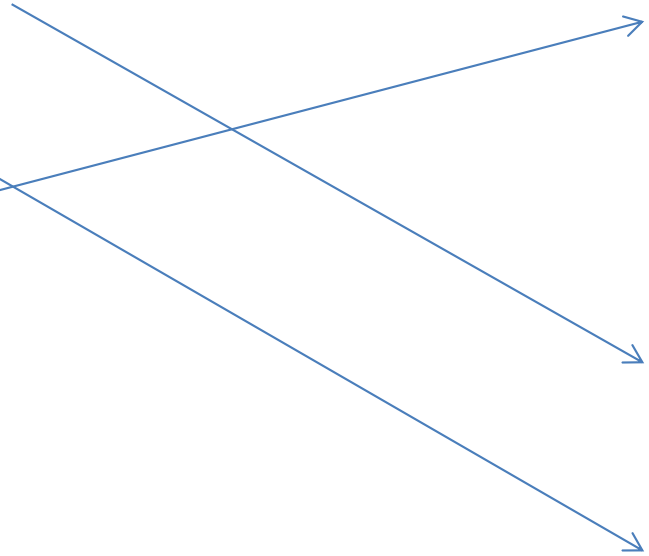
- Decorator
- Proxy
- Façade
- Adapter



- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

# Patterns & Definitions – Group 2

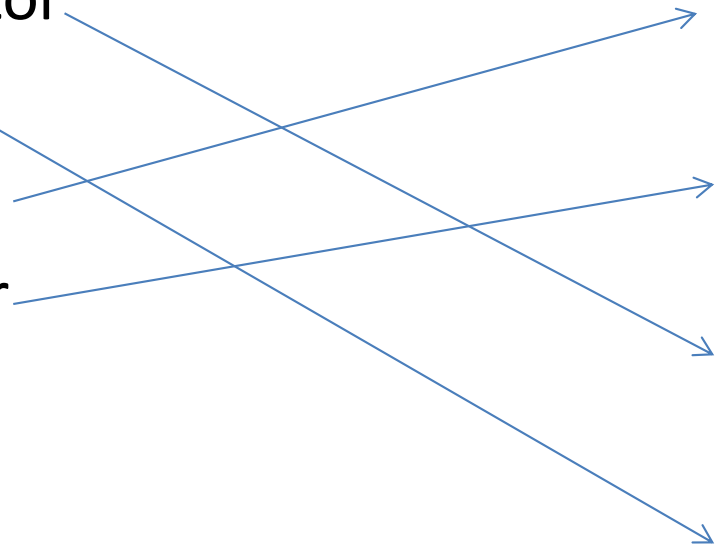
- Decorator
- Proxy
- Façade
- Adapter



- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

# Patterns & Definitions – Group 2

- Decorator
- Proxy
- Façade
- Adapter



- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

# Conclusion - Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Principle of least knowledge – talk only to your immediate friends