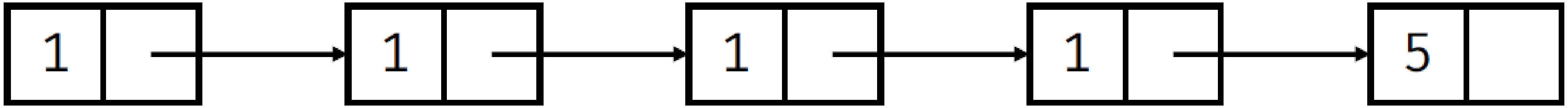
Homework07 Review

2022 SICP

Hw07p2.1 Remove Duplicates

Take in a **sorted** linked list of integers and **mutate** it so that all duplicates are removed.



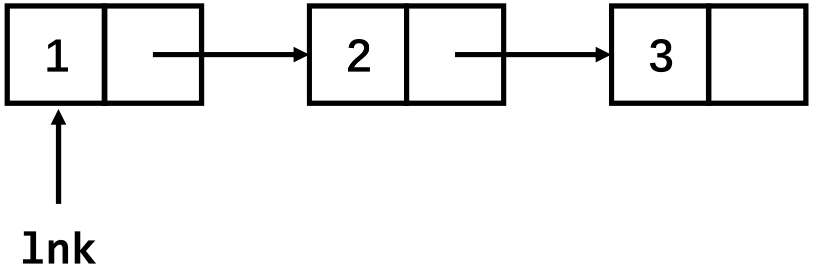
```
def remove_duplicates(lnk):
    while lnk is not Link.empty:
        while lnk.rest is not Link.empty and lnk.rest.first == lnk.first:
            lnk.rest = lnk.rest.rest
        lnk = lnk.rest
```

DO NOT create any new linked lists and DO NOT return the modified Link object.

Hw07p2.2 Reverse

Reverse the order of a linked list and **return** the reversed list

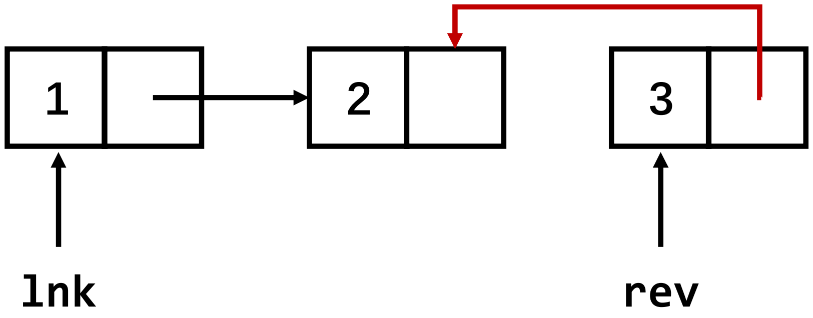
```
def reverse(lnk):
    if lnk is Link.empty or lnk.rest is Link.empty:
        return lnk
    rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = Link.empty
    return rev
```



Hw07p2.2 Reverse

Reverse the order of a linked list and **return** the reversed list

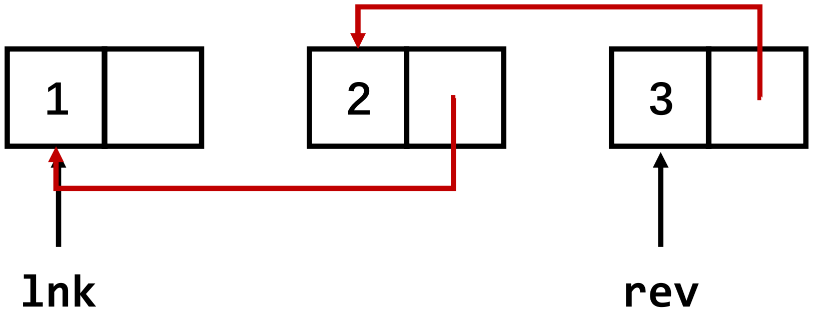
```
def reverse(lnk):
    if lnk is Link.empty or lnk.rest is Link.empty:
        return lnk
    rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = Link.empty
    return rev
```



Hw07p2.2 Reverse

Reverse the order of a linked list and **return** the reversed list

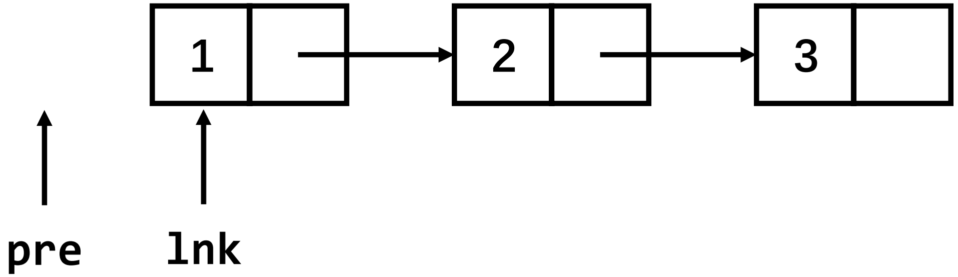
```
def reverse(lnk):
    if lnk is Link.empty or lnk.rest is Link.empty:
        return lnk
    rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = Link.empty
    return rev
```



Hw07p2.2 Reverse

Reverse the order of a linked list and **return** the reversed list

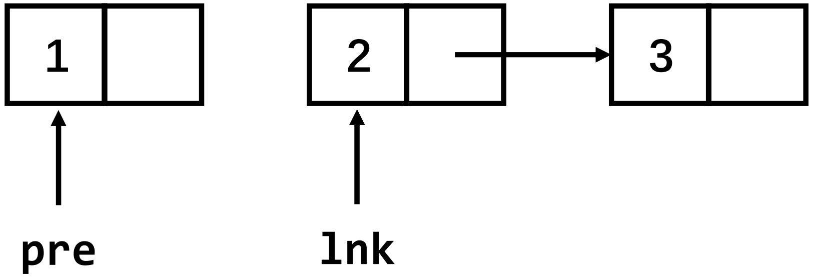
```
def reverse(lnk):
    pre = Link.empty
    while lnk is not Link.empty:
        rest = lnk.rest
        lnk.rest = pre
        pre = lnk
        lnk = rest
    return pre
```



Hw07p2.2 Reverse

Reverse the order of a linked list and **return** the reversed list

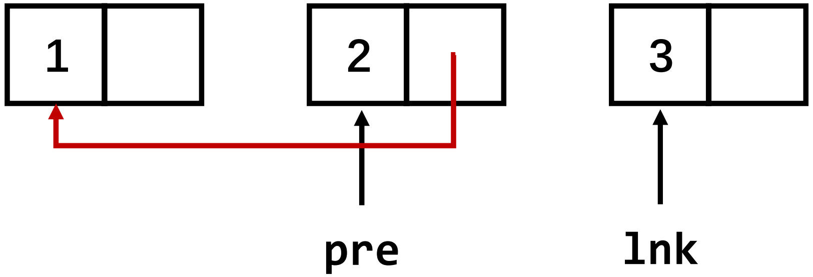
```
def reverse(lnk):
    pre = Link.empty
    while lnk is not Link.empty:
        rest = lnk.rest
        lnk.rest = pre
        pre = lnk
        lnk = rest
    return pre
```



Hw07p2.2 Reverse

Reverse the order of a linked list and **return** the reversed list

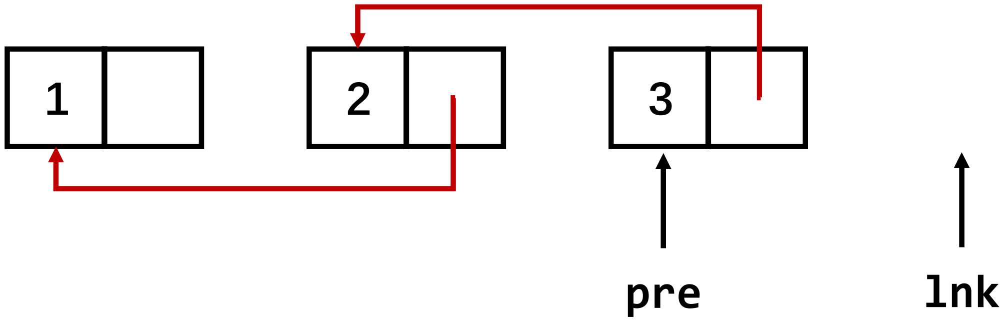
```
def reverse(lnk):
    pre = Link.empty
    while lnk is not Link.empty:
        rest = lnk.rest
        lnk.rest = pre
        pre = lnk
        lnk = rest
    return pre
```



Hw07p2.2 Reverse

Reverse the order of a linked list and **return** the reversed list

```
def reverse(lnk):
    pre = Link.empty
    while lnk is not Link.empty:
        rest = lnk.rest
        lnk.rest = pre
        pre = lnk
        lnk = rest
    return pre
```

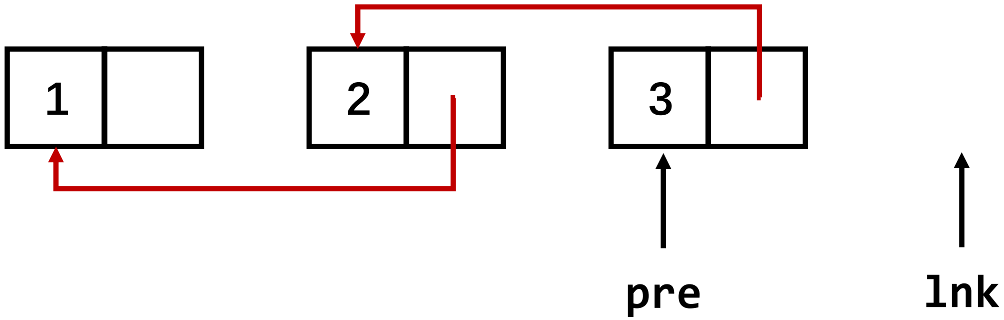


Hw07p2.2 Reverse

Reverse the order of a linked list and **return** the reversed list

```
def reverse(lnk):
    pre = Link.empty
    while lnk is not Link.empty:
        rest = lnk.rest
        lnk.rest = pre
        pre = lnk
        lnk = rest
    return pre
```

```
def reverse(lnk):
    pre = Link.empty
    while lnk is not Link.empty:
        lnk.rest, pre, lnk = pre, lnk, lnk.rest
    return pre
```



Hw07p3.1 Equal Trees

```
class Tree:
    def __eq__(self, other):
        return self.label == other.label and \
            len(self.branches) == len(other.branches) and \
            all([b1 == b2 for b1, b2 in zip(self.branches, other.branches)])
```

```
class Tree:
    def __eq__(self, other):
        return repr(self) == repr(other)
```

Hw07p3.2 Generate Paths

Yield each path from the root of `t` to a node that has label `value`.

```
def generate_paths(t, value):
    if t.label == value:
        yield [value]
    for b in t.branches:
        for p in generate_paths(b, value):
            yield [t.label] + p
```

Hw07p3.3 Count Coins Tree

```
def count_coins(total, denominations):
    if total == 0:
        return 1
    if total < 0:
        return 0
    if len(denominations) == 0:
        return 0
    without_current = count_coins(total, denominations[1:])
    with_current = count_coins(total - denominations[0], denominations)
    return without_current + with_current
```

The implementation of `count_coins_tree`

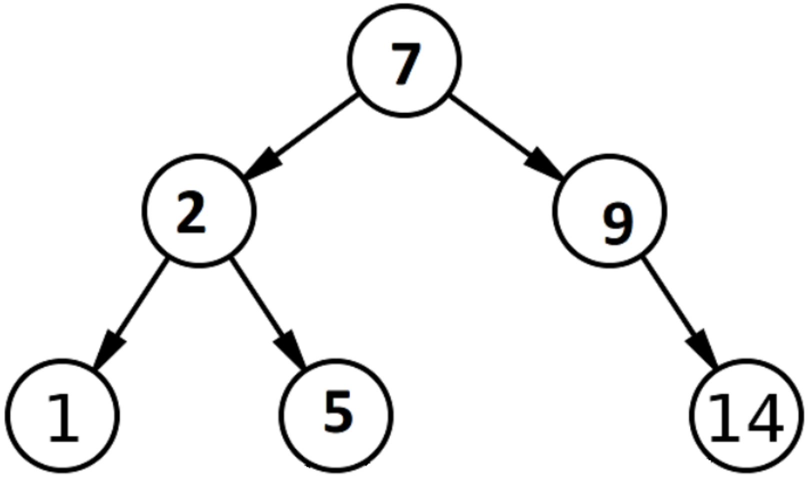
will follow **a similar logic** to `count_coins` defined above.

Hw07p3.3 Count Coins Tree

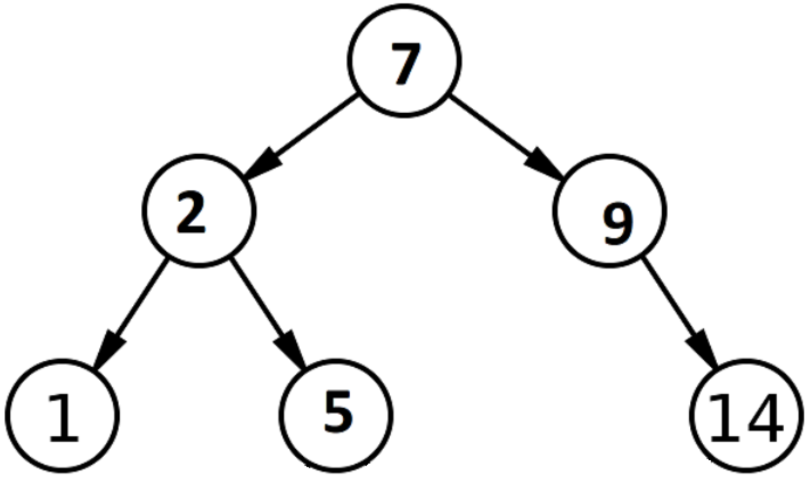
```
def count_coins_tree(total, denominations):
    if total == 0:
        return Tree('1')
    if total < 0 or len(denominations) == 0:
        return None
    without_current = count_coins_tree(total, denominations[1:])
    with_current = count_coins_tree(total - denominations[0], denominations)
    branches = list(filter(lambda x: x is not None, [without_current, with_current]))
    if len(branches) == 0:
        return None
    return Tree(f'{total}, {denominations}', branches)
```

Hw07p3.4 Is BST

- Each node has **at most** two children (a leaf is automatically a valid binary search tree)
- The children are valid binary search trees
- For every node n, the label of every node in n's **left** child are **less than or equal to** the label of n
- For every node n, the label of every node in n's **right** child are **greater** than the label of n

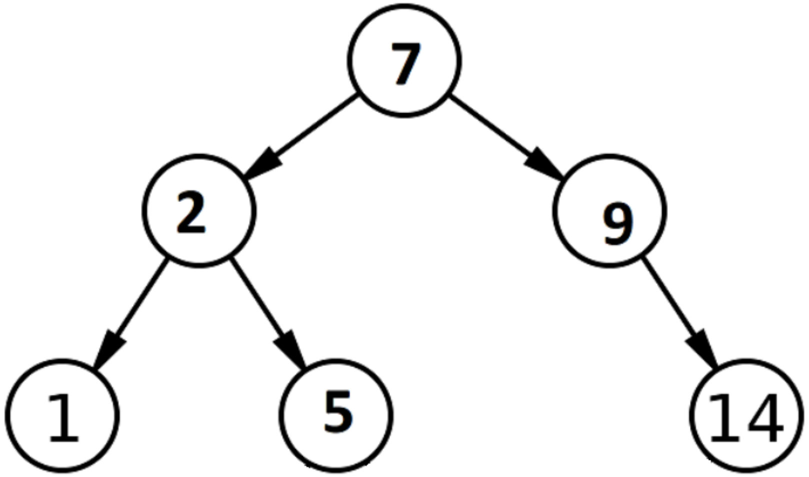


Hw07p3.4 Is BST



```
def bst_min(t):
    if t.is_leaf():
        return t.label
    return min(t.label, bst_min(t.branches[0]))

def bst_max(t):
    if t.is_leaf():
        return t.label
    return max(t.label, bst_max(t.branches[-1]))
```

Hw07p3.4 Is BST

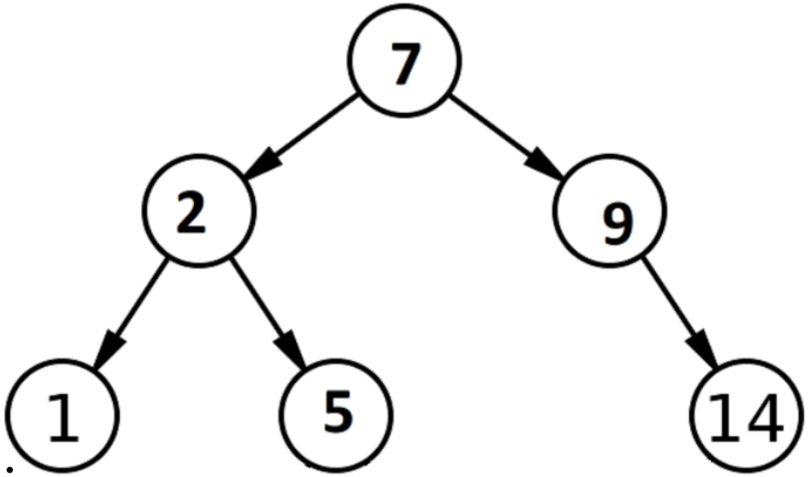


```python
def is_bst(t):
    if t.is_leaf():
        return True
    if len(t.branches) == 1:
        c = t.branches[0]
        return is_bst(c) and (bst_max(c) <= t.label or bst_min(c) > t.label)
    elif len(t.branches) == 2:
        c1, c2 = t.branches
        return is_bst(c1) and bst_max(c1) <= t.label and is_bst(c2) and bst_min(c2) > t.label
    else:
        return False
```

Hw07p3.4 Is BST

选定上下界，左开右闭区间

```
def is_bst(t):
    def helper(tr, lowerbound, upperbound):
        if tr.is_leaf():
            return lowerbound < tr.label <= upperbound
        if tr.label <= lowerbound or tr.label > upperbound:
            return False
        if len(tr.branches) == 1:
            if tr.branches[0].label <= tr.label:
                return helper(tr.branches[0], lowerbound, tr.label)
            else:
                return helper(tr.branches[0], tr.label, upperbound)
        elif len(tr.branches) == 2:
            return helper(tr.branches[0], lowerbound, tr.label) and \
                    helper(tr.branches[1], tr.label, upperbound)
        else:
            return False
    return helper(t, float("-inf"), float("inf"))
```



Hw07p4 Has Cycle

Floyd Cycle Detection Algorithm

```
def has_cycle(lnk):
    fast, slow = lnk, lnk
    while fast is not Link.empty and fast.rest is not Link.empty:
        fast = fast.rest.rest
        slow = slow.rest
        if slow is fast:
            return True
    return False
```

Hw07p5 Decorate Christmas Tree

A tree is balanced if it is a leaf or the **total weight** of its every branches **are the same** and all of its branches are balanced.

```
def balance_tree(t):
    for b in t.branches:
        balance_tree(b)
    max_total = max([b.total for b in t.branches], default=0)
    for b in t.branches:
        b.label += max_total - b.total
    t.total = t.label + max_total * len(t.branches)
```

Something about OOP...

POP vs. OOP

Procedure Oriented

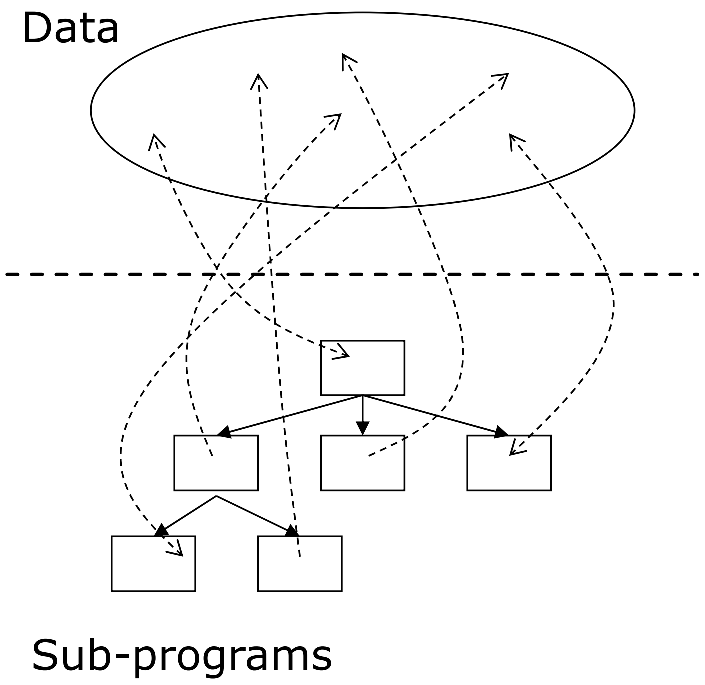
- 编程任务明确
- 效率高
- 数据与操作分离，二者联系松散
- 数据缺乏保护
- 功能易变，程序维护困难
- 难以复用

Object Oriented

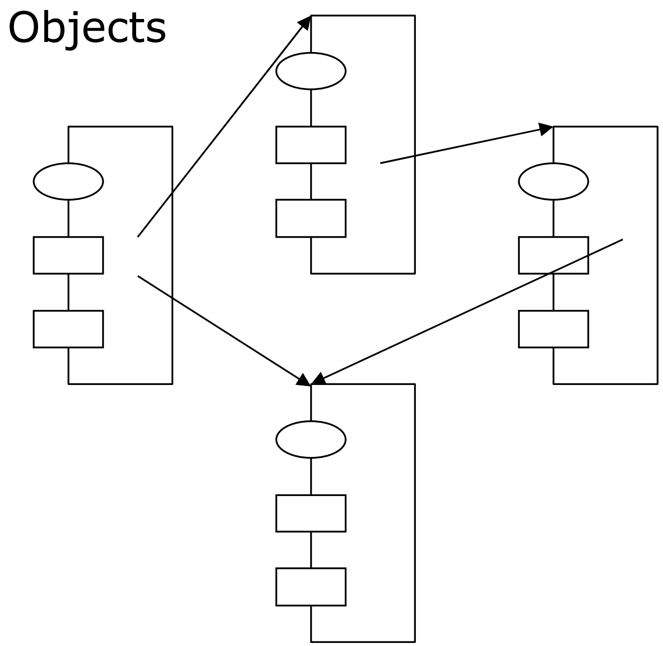
- 操作依附于数据，联系紧密
- 隐藏操作所需数据，加强数据保护
- 易于代码复用与维护(?)，低耦合(?)
- 开销大

POP vs. OOP

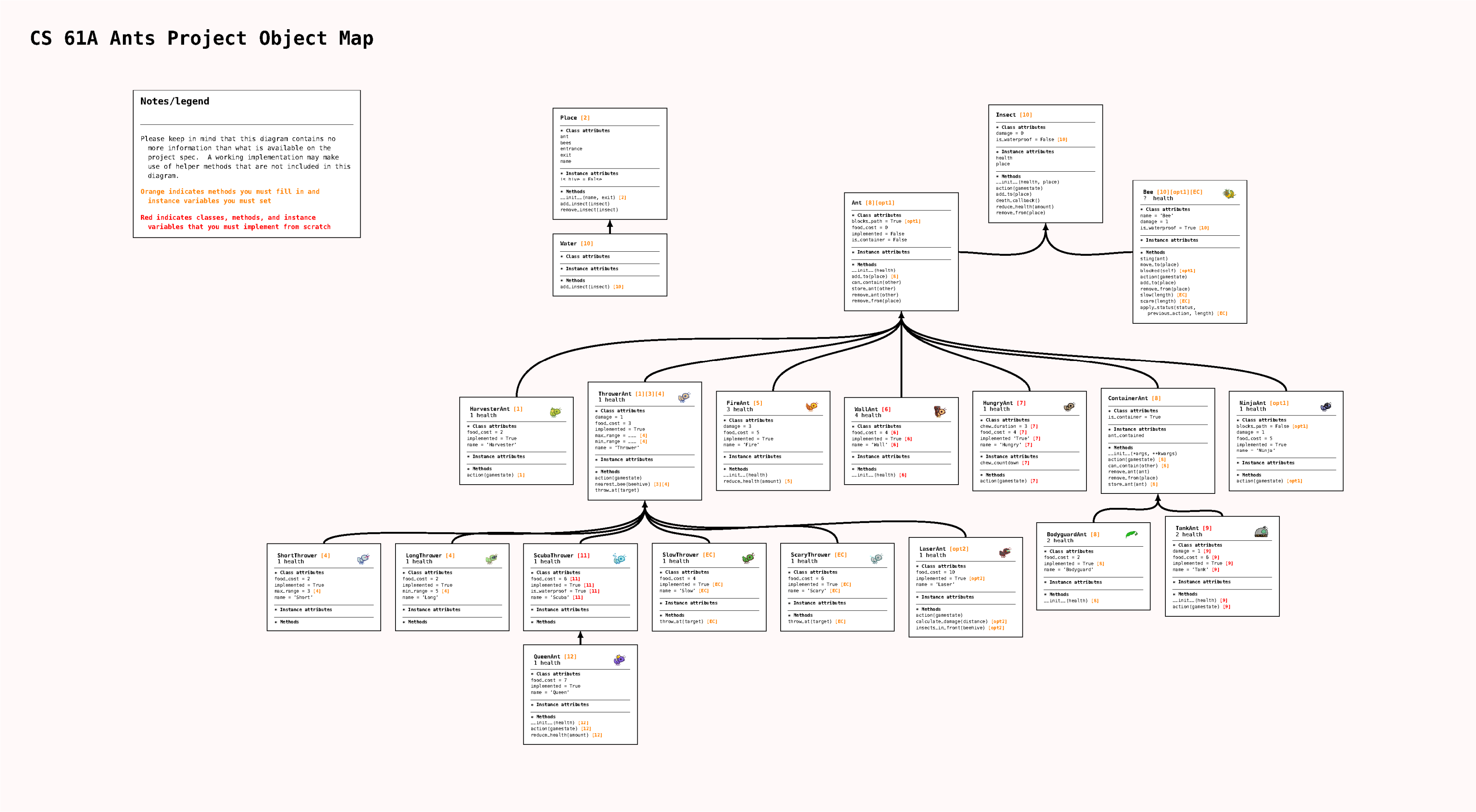
Procedure Oriented



Object Oriented



OOP



CS 61A Ants Project Object Map

Notes/legend

Please keep in mind that this diagram contains no more information than what is available on the project spec. A working implementation may make use of helper methods that are not included in this diagram.

Orange indicates methods you must fill in and instance variables you must set

Red indicates classes, methods, and instance variables that you must implement from scratch

```
Place [2]
* Class attributes
ant
bees
entrance
exit
name
* Instance attributes
is_hive = False
* Methods
__init__(name, exit) [2]
add_insect(insect)
remove_insect(insect)
```

```
Water [10]
* Class attributes
* Instance attributes
* Methods
add_insect(insect) [10]
```

```
Insect [10]
* Class attributes
damage = 0
is_waterproof = False [10]
* Instance attributes
health
place
* Methods
__init__(health, place)
action(gamestate)
add_to(place)
death_callback()
reduce_health(amount)
remove_from(place)
```

```
Ant [8][opt1]
* Class attributes
blocks_path = True [opt1]
food_cost = 0
implemented = False
is_container = False
* Instance attributes
* Methods
__init__(health)
add_to(place) [8]
can_contain(other)
store_ant(other)
remove_ant(other)
remove_from(place)
```

```
Bee [10][opt1][EC]
?  health
* Class attributes
name = 'Bee'
damage = 1
is_waterproof = True [10]
* Instance attributes
* Methods
sting(ant)
move_to(place)
blocked(self) [opt1]
action(gamestate)
add_to(place)
remove_from(place)
slow(length) [EC]
scare(length) [EC]
apply_status(status,
  previous_action, length) [EC]
```

```
HarvesterAnt [1]
1 health
* Class attributes
food_cost = 2
implemented = True
name = 'Harvester'
* Instance attributes
* Methods
action(gamestate) [1]
```

```
ThrowerAnt [1][3][4]
1 health
* Class attributes
damage = 1
food_cost = 3
implemented = True
max_range = ___ [4]
min_range = ___ [4]
name = 'Thrower'
* Instance attributes
* Methods
action(gamestate)
nearest_bee(beehive) [3][4]
throw_at(target)
```

```
FireAnt [5]
3 health
* Class attributes
damage = 3
food_cost = 5
implemented = True
name = 'Fire'
* Instance attributes
* Methods
__init__(health)
reduce_health(amount) [5]
```

```
WallAnt [6]
4 health
* Class attributes
food_cost = 4 [6]
implemented = True [6]
name = 'Wall' [6]
* Instance attributes
* Methods
__init__(health) [6]
```

```
HungryAnt [7]
1 health
* Class attributes
chew_duration = 3 [7]
food_cost = 4 [7]
implemented 'True' [7]
name = 'Hungry' [7]
* Instance attributes
chew_countdown [7]
* Methods
action(gamestate) [7]
```

```
ContainerAnt [8]
* Class attributes
is_container = True
* Instance attributes
ant_contained
* Methods
__init__(*args, **kwargs)
action(gamestate) [8]
can_contain(other) [8]
remove_ant(ant)
remove_from(place)
store_ant(ant) [8]
```

```
NinjaAnt [opt1]
1 health
* Class attributes
blocks_path = False [opt1]
damage = 1
food_cost = 5
implemented = True
name - 'Ninja'
* Instance attributes
* Methods
action(gamestate) [opt1]
```

```
ShortThrower [4]
1 health
* Class attributes
food_cost = 2
implemented = True
max_range = 3 [4]
name = 'Short'
* Instance attributes
* Methods
```

```
LongThrower [4]
1 health
* Class attributes
food_cost = 2
implemented = True
min_range = 5 [4]
name = 'Long'
* Instance attributes
* Methods
```

```
ScubaThrower [11]
1 health
* Class attributes
food_cost = 6 [11]
implemented = True [11]
is_waterproof = True [11]
name = 'Scuba' [11]
* Instance attributes
* Methods
```

```
SlowThrower [EC]
1 health
* Class attributes
food_cost = 4
implemented = True [EC]
name = 'Slow' [EC]
* Instance attributes
* Methods
throw_at(target) [EC]
```

```
ScaryThrower [EC]
1 health
* Class attributes
food_cost = 6
implemented = True [EC]
name = 'Scary' [EC]
* Instance attributes
* Methods
throw_at(target) [EC]
```

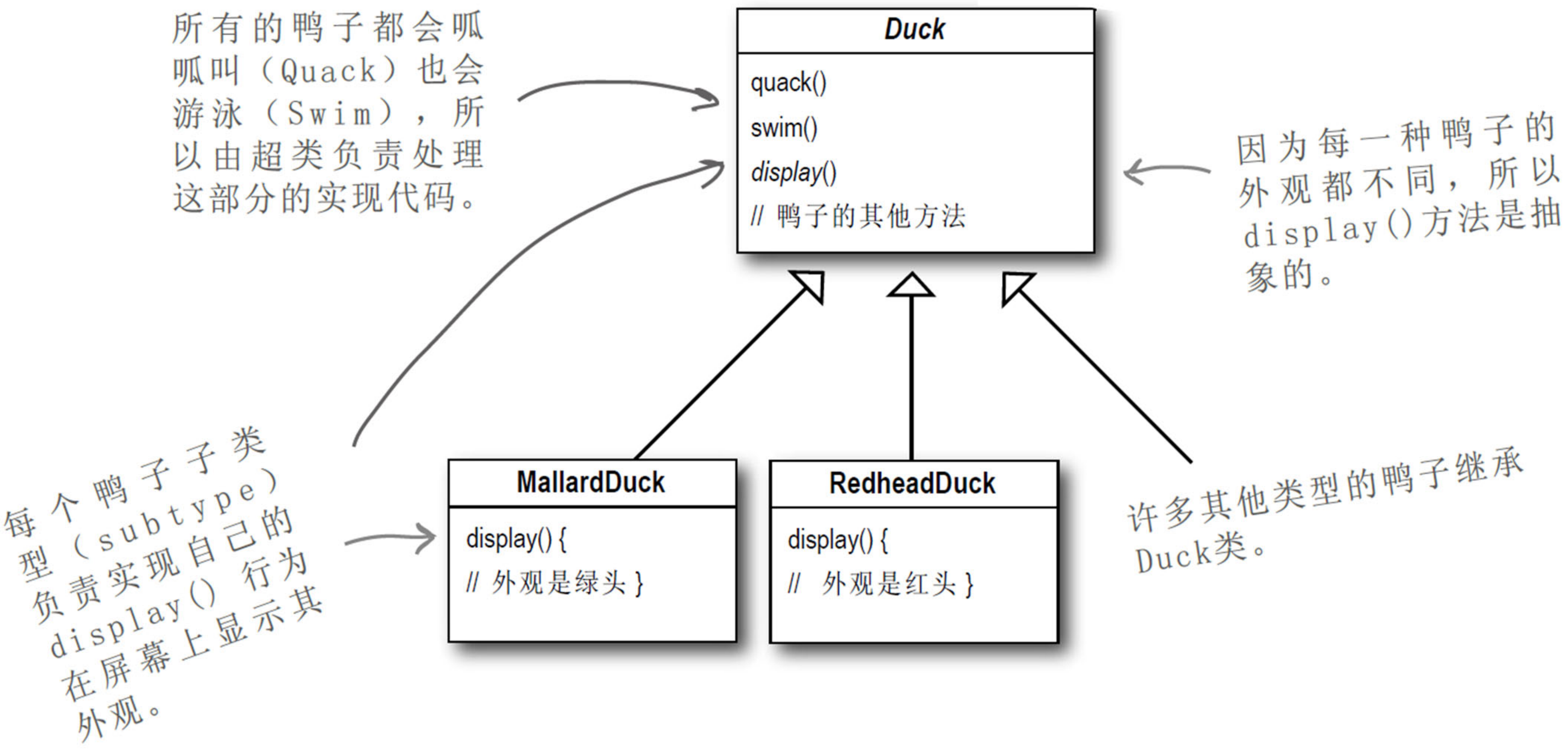
```
LaserAnt [opt2]
1 health
* Class attributes
food_cost = 10
implemented = True [opt2]
name = 'Laser'
* Instance attributes
* Methods
action(gamestate)
calculate_damage(distance) [opt2]
insects_in_front(beehive) [opt2]
```

```
BodyguardAnt [8]
2 health
* Class attributes
food_cost = 2
implemented = True [8]
name = 'Bodyguard'
* Instance attributes
* Methods
__init__(health) [8]
```

```
TankAnt [9]
2 health
* Class attributes
damage = 1 [9]
food_cost = 6 [9]
implemented = True [9]
name = 'Tank' [9]
* Instance attributes
* Methods
__init__(health) [9]
action(gamestate) [9]
```

```
QueenAnt [12]
1 health
* Class attributes
food_cost = 7
implemented = True
name = 'Queen'
* Instance attributes
* Methods
__init__(health) [12]
action(gamestate) [12]
reduce_health(amount) [12]
```

设计一个Duck类



所有的鸭子都会呱呱叫（Quack）也会游泳（Swim），所以由超类负责处理这部分的实现代码。

Duck

quack()

swim()

display()

// 鸭子的其他方法

因为每一种鸭子的外观都不同，所以display()方法是抽象的。

MallardDuck

display() {

// 外观是绿头 }

RedheadDuck

display() {

// 外观是红头 }

每个鸭子子类型（subtype）负责实现自己的display()行为在屏幕上显示其外观。

许多其他类型的鸭子继承Duck类。

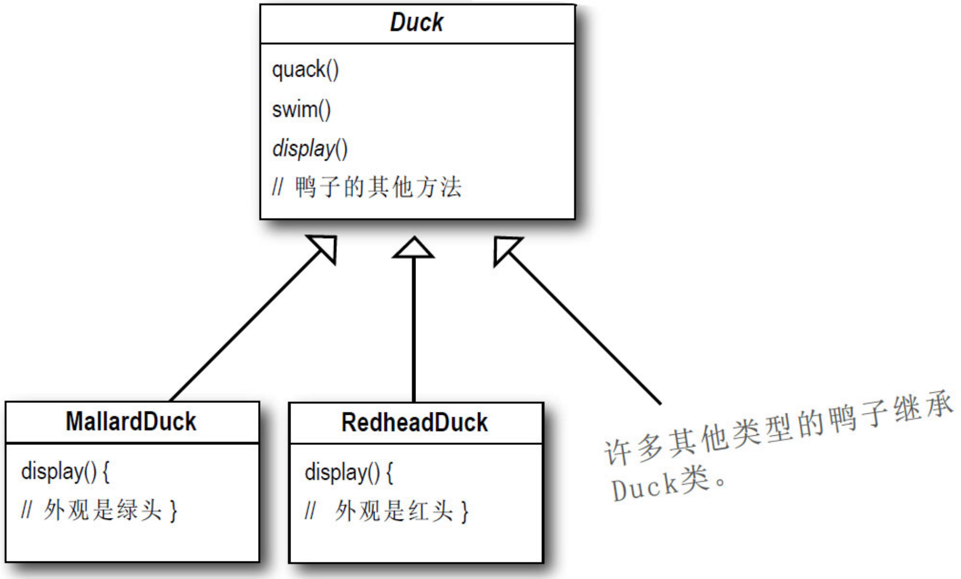
设计一个Duck类

```python
class Duck:
    def __init__(self, name):
        self.name = name

    def swim(self):
        print(self.name + " is swimming.")

    def quack(self):
        print(self.name + " is quacking.")

    def display(self):
        print("I am a Duck.")
```

```python
class MallardDuck(Duck):
    def __init__(self, name):
        super().__init__(name)

    def display(self):
        print("I am a MallardDuck.")


class RedheadDuck(Duck):
    def __init__(self, name):
        super().__init__(name)

    def display(self):
        print("I am a RedheadDuck.")
```

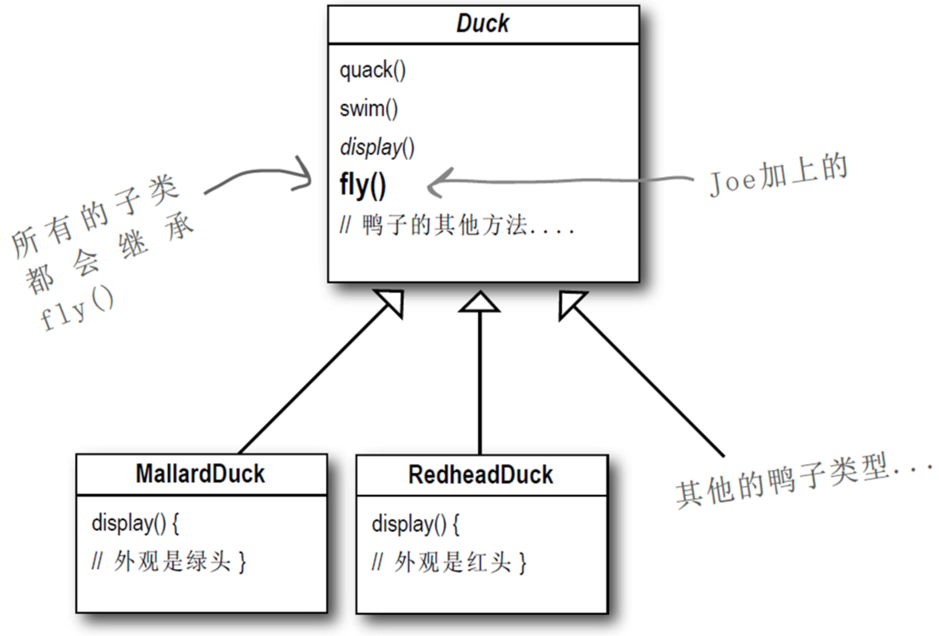
设计一个Duck类

- 给鸭子加上fly行为



设计一个Duck类

- 给鸭子加上fly行为

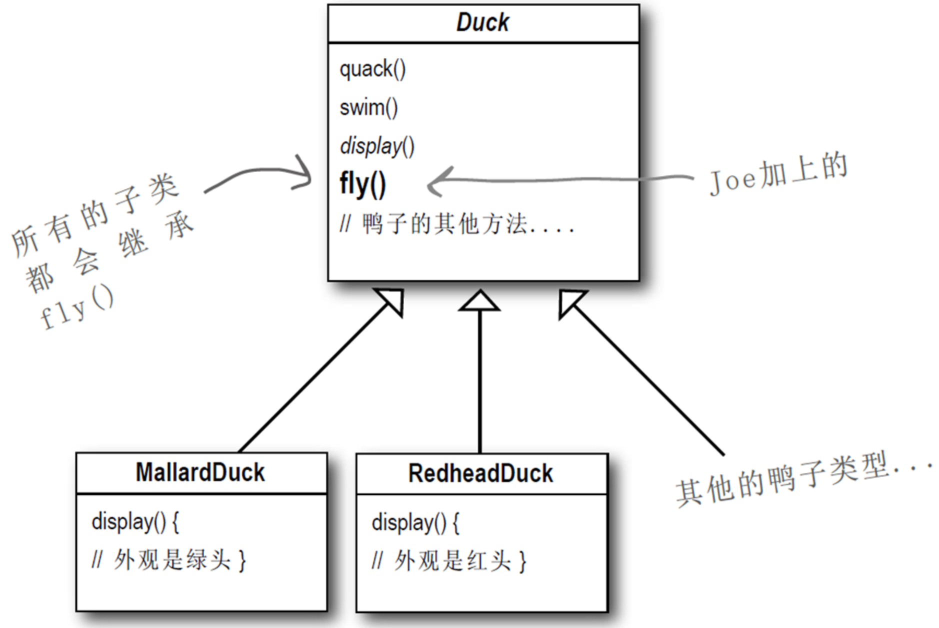


给鸭子加上fly行为

```python
class Duck:
    def __init__(self, name):
        self.name = name

    def swim(self):
        print(self.name + " is swimming.")

    def quack(self):
        print(self.name + " is quacking.")

    def display(self):
        print("I am a Duck.")

    def fly(self):
        print(self.name + " is flying.")
```

给鸭子加上fly行为

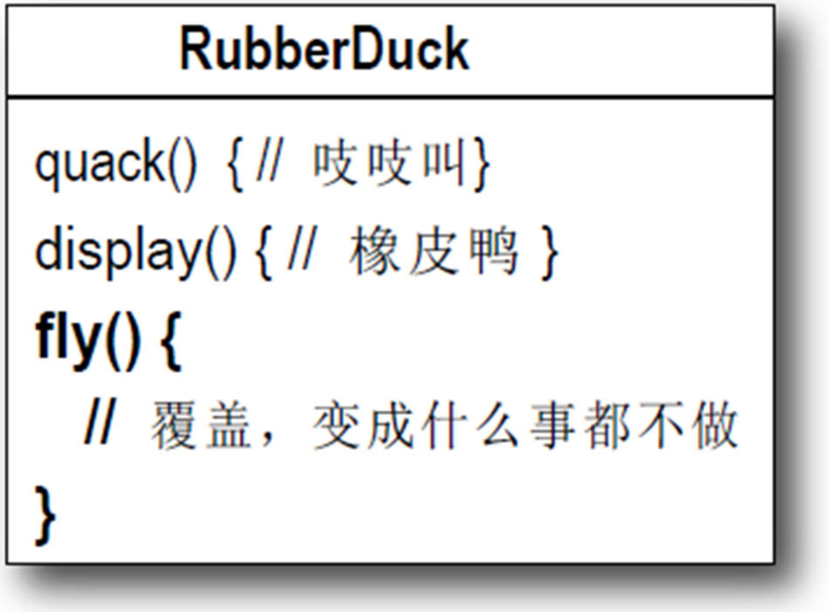
- 给鸭子加上fly行为
- 但是有些鸭子不会飞
- 在超类中加上fly()，会导致所有的子类都具有该行为，连那些不应该具有fly行为的子类也无法避免



覆盖

- RubberDuck不会飞
- 且它们只会吱吱叫(squeak)

```
class RubberDuck(Duck):
    def __init__(self, name):
        super().__init__(name)

    def quack(self):
        print(self.name + " is squeaking.")

    def fly(self):
        pass

    def display(self):
        print("I am a RubberDuck.")
```



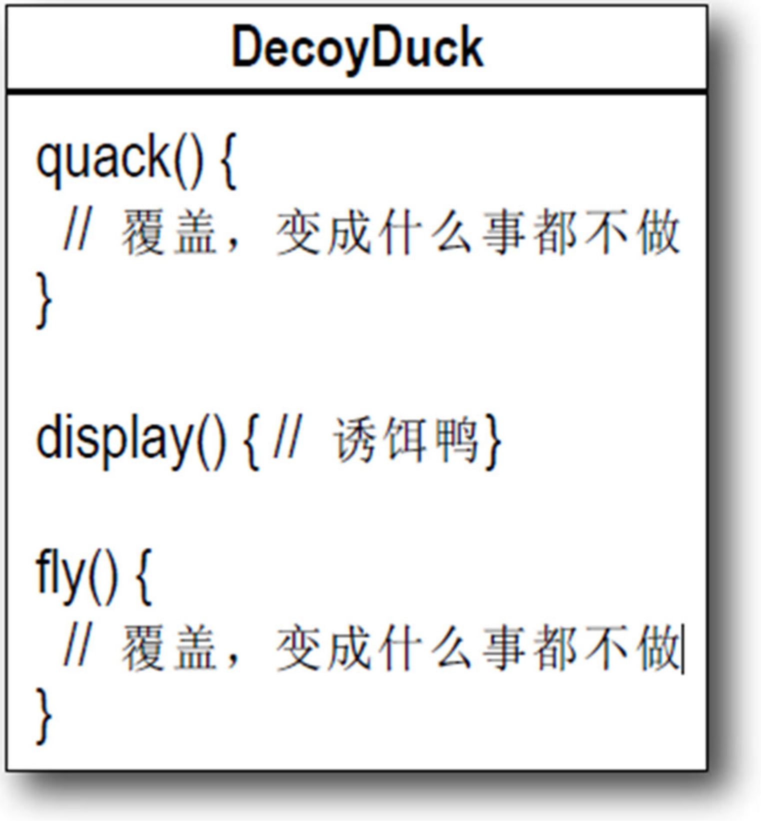
RubberDuck

```
quack() {// 吱吱叫}
display() { // 橡皮鸭 }
fly() {
  // 覆盖，变成什么事都不做
}
```

新需求又来啦

- 加入诱饵鸭(DecoyDuck)
- 不会飞也不会叫

```
class DecoyDuck(Duck):
    def __init__(self, name):
        super().__init__(name)

    def quack(self):
        pass

    def fly(self):
        pass

    def display(self):
        print("I am a DecoyDuck.")
```



DecoyDuck

```
quack() {
  // 覆盖，变成什么事都不做
}

display() { // 诱饵鸭}

fly() {
  // 覆盖，变成什么事都不做
}
```

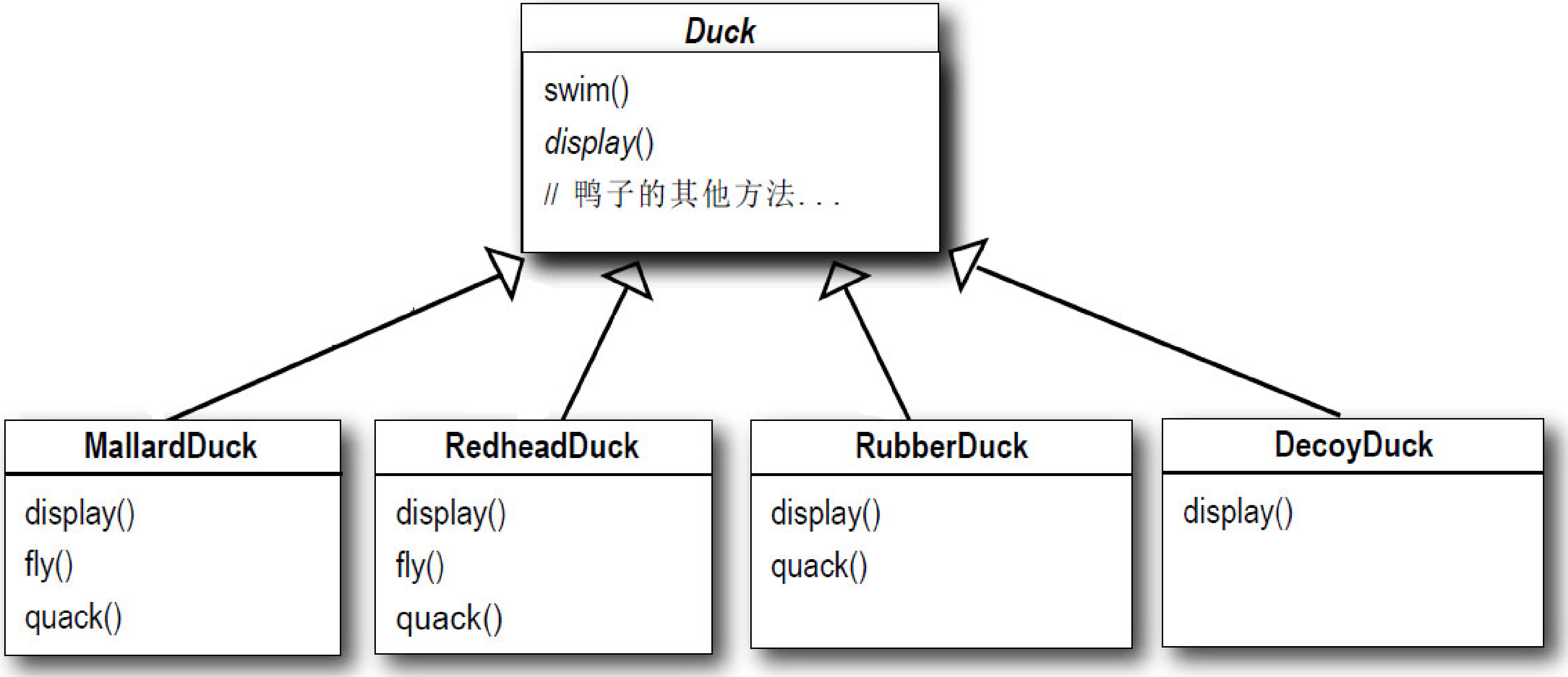
更多的需求正在路上...

- 有五种鸭子会飞，且会嘎嘎叫
- 有三种鸭子能飞得比较高，但不会叫
- 给鸭子增加奔跑的行为
- ……
- 为了复用而使用继承，并不完美

多选题

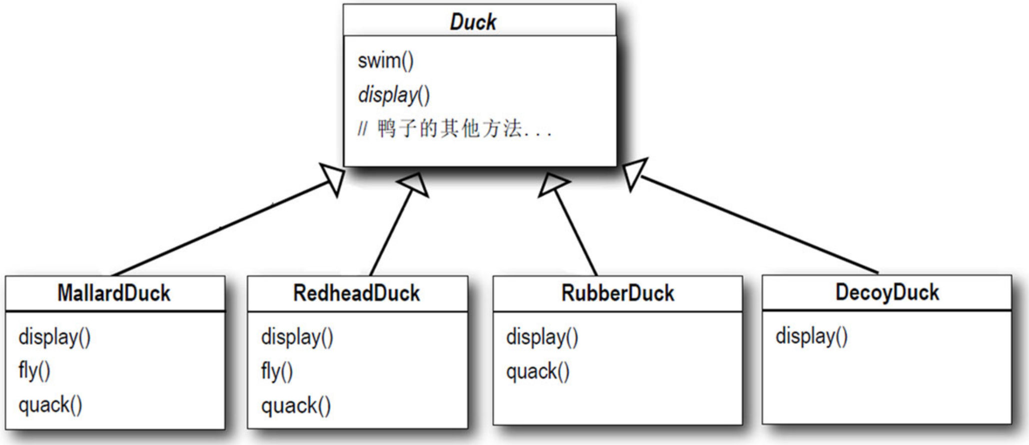
- 利用继承来提供Duck的行为，这会导致下列哪些缺点?
- A. 代码在多个子类中重复
- B. 运行时鸭子的行为不容易改变
- C. 我们不能让鸭子跳舞
- D. 很难知道鸭子的全部具体行为
- E. 鸭子不能同时又飞又叫
- F. 改变会牵一发而动全身，造成其他鸭子不想要的改变

将行为放在子类中?


Duck
swim()
display()
// 鸭子的其他方法...
MallardDuck
display()
fly()
quack()
RedheadDuck
display()
fly()
quack()
RubberDuck
display()
quack()
DecoyDuck
display()

将行为放在子类中?

- 代码无法复用
- 某一个行为出现bug，需要到所有子类中修改



POP vs. OOP

Procedure Oriented

- 编程任务明确
- 效率高
- 数据与操作分离，二者联系松散
- 数据缺乏保护
- 功能易变，程序维护困难
- 难以复用

Object Oriented

- 操作依附于数据，联系紧密
- 隐藏操作所需数据，加强数据保护
- 易于代码复用与维护(?)，低耦合(?)
- 开销大

采用良好的OO软件设计原则

设计原则

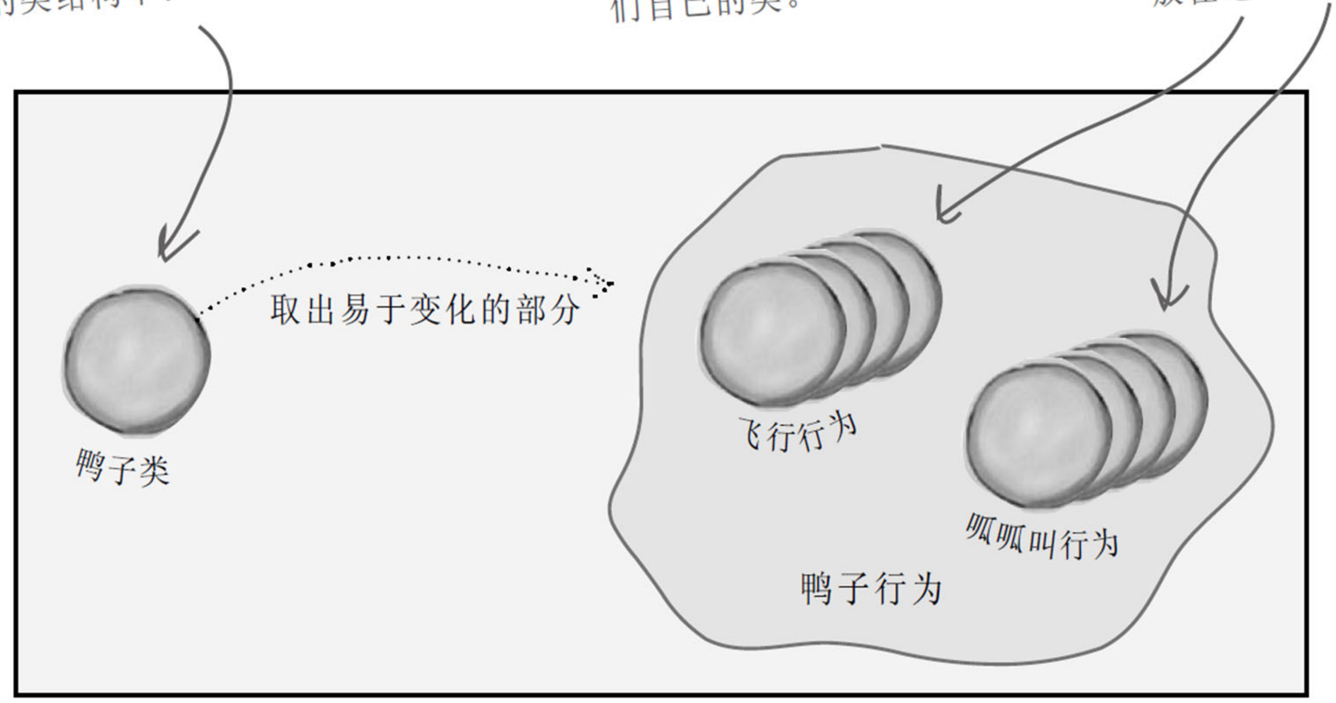
- 将应用中可能需要变化之处独立出来，不和那些不需要变化的代码混在一起

分开变化和不会变化的部分

Duck 类仍是所有鸭子的超类，但是飞行和呱呱叫的行为已经被取出，放在别的类结构中。

现在飞行和呱呱叫都有它们自己的类。

多种行为的实现被放在这里。



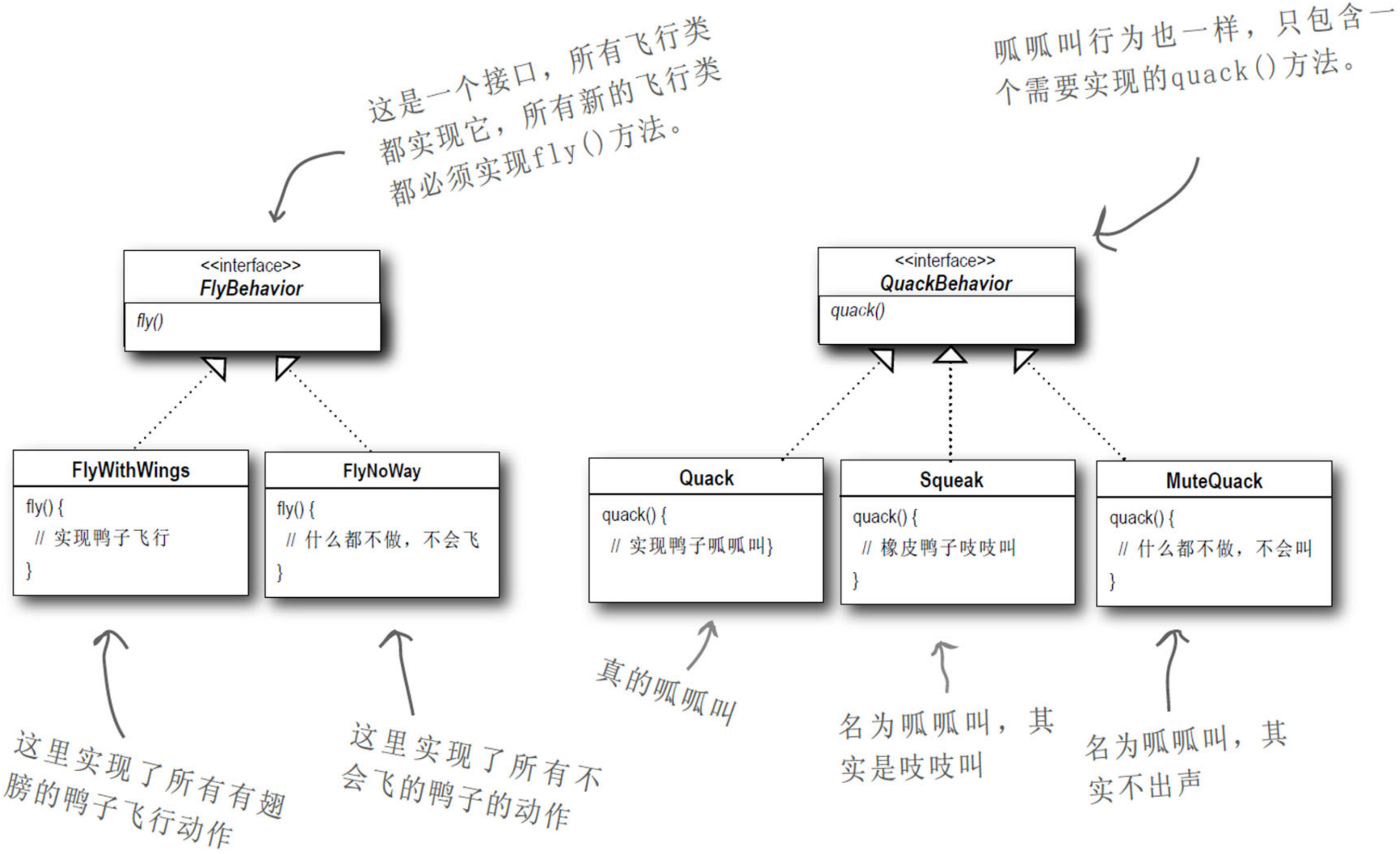
分开变化和不会变化的部分

- 将鸭子的行为放在分开的类中
- 这些类专门提供某一种行为的实现
- 鸭子类不再需要知道行为的实现细节
- 鸭子类不再负责实现FlyBehavior和QuackBehavior

设计原则

- 将应用中可能需要变化之处独立出来，不和那些不需要变化的代码混在一起
- 针对接口编程，而不是针对实现编程

实现鸭子的行为



这是一个接口，所有飞行类都实现它，所有新的飞行类都必须实现fly()方法。

呱呱叫行为也一样，只包含一个需要实现的quack()方法。

```
fly() {
 // 实现鸭子飞行
}
```

```
fly() {
 // 什么都不做，不会飞
}
```

```
quack() {
 // 实现鸭子呱呱叫}
```

```
quack() {
 // 橡皮鸭子吱吱叫
}
```

```
quack() {
 // 什么都不做，不会叫
}
```

这里实现了所有有翅膀的鸭子飞行动作

这里实现了所有不会飞的鸭子的动作

真的呱呱叫

名为呱呱叫，其实是吱吱叫

名为呱呱叫，其实不出声

实现鸭子的飞行行为

```python
class FlyBehavior:
    def fly(self, name):
        raise NotImplementedError("FlyBehavior is not implemented")

class FlyWithWings(FlyBehavior):
    def fly(self, name):
        print(name + " is flying.")

class FlyNoWay(FlyBehavior):
    def fly(self, name):
        print("I can't fly.")
```

实现鸭子的叫行为

```python
class QuackBehavior:
    def quack(self, name):
        raise NotImplementedError("QuackBehavior is not implemented")

class Quack(QuackBehavior):
    def quack(self, name):
        print(name + " is quacking.")

class MuteQuack(QuackBehavior):
    def quack(self, name):
        print("Silence")

class Squeak(QuackBehavior):
    def quack(self, name):
        print(name + " is squeaking.")
```

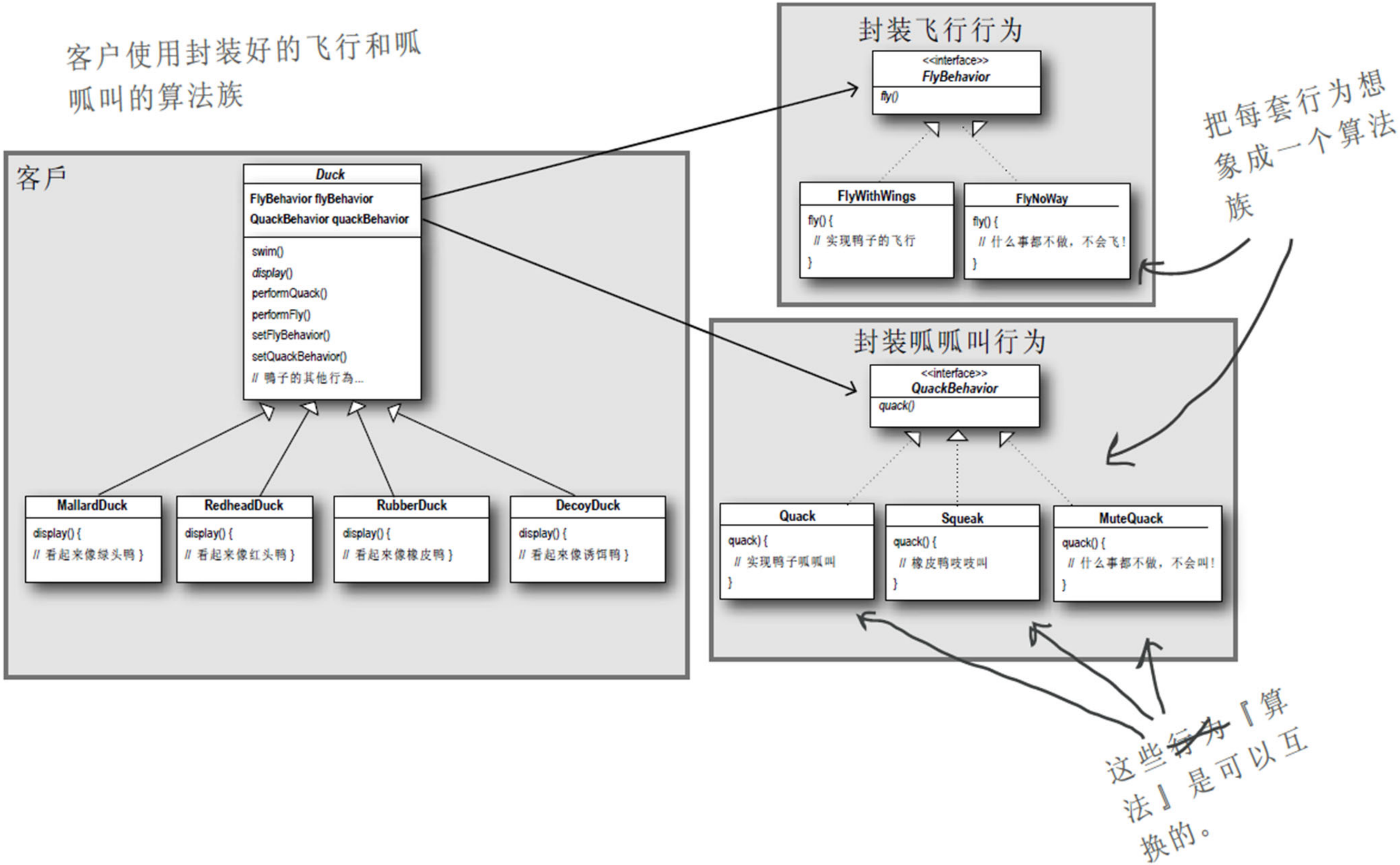
实现鸭子

```python
class Duck:
    def __init__(self, name, flyBehavior, quackBehavior):
        self.name = name
        self.flyBehavior = flyBehavior
        self.quackBehavior = quackBehavior

    def swim(self):
        print(self.name + " is swimming.")

    def performQuack(self):
        self.quackBehavior.quack(self.name)

    def performFly(self):
        self.flyBehavior.fly(self.name)

    def display(self):
        print("I am a Duck.")
```

整合鸭子的行为

省略display方法

```python
class MallardDuck(Duck):
    def __init__(self, name):
        super().__init__(name, FlyWithWings(), Quack())

class RubberDuck(Duck):
    def __init__(self, name):
        super().__init__(name, FlyNoWay(), Squeak())

class DecoyDuck(Duck):
    def __init__(self, name):
        super().__init__(name, FlyNoWay(), MuteQuack())
```



客户使用封装好的飞行和呱呱叫的算法族

客戶

```
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// 鴨子的其他行為...
```

```
MallardDuck
display() {
// 看起來像绿头鸭 }
```

```
RedheadDuck
display() {
// 看起來像红头鸭 }
```

```
RubberDuck
display() {
// 看起來像橡皮鸭 }
```

```
DecoyDuck
display() {
// 看起來像诱饵鸭 }
```

封装飞行行为

```
<<interface>>
FlyBehavior
fly()
```

```
FlyWithWings
fly() {
 // 实现鸭子的飞行
}
```

```
FlyNoWay
fly() {
 // 什么事都不做，不会飞！
}
```

把每套行为想象成一个算法族

封装呱呱叫行为

```
<<interface>>
QuackBehavior
quack()
```

```
Quack
quack) {
 // 实现鸭子呱呱叫
}
```

```
Squeak
quack() {
 // 橡皮鸭吱吱叫
}
```

```
MuteQuack
quack() {
 // 什么事都不做，不会叫！
}
```

这些~~行为~~『算法』是可以互换的。

封装行为的大局观

- 飞行和呱呱叫的行为可以被其他对象复用，因为这些行为已经与鸭子类无关了
- 可以很方便的新增一些行为，而不会影响既有的行为类，也不会影响使用到飞行行为的鸭子类
- 利用继承的“复用”，但没有继承所带来的各种问题

“有一个”可能比“是一个”更好

- 每一个鸭子都有一个FlyBehavior和一个QuackBehavior
- 将飞行和呱呱叫委托给它们代为处理
- Composition
- 鸭子的行为不是继承来的，而是和适当的行为对象组合来的

Composition

Colloquially, composition means

"If you want to reuse some behavior, put that behavior in a class, create an object of that class, include it as an attribute, and call its methods when the behavior is needed"

- Composition does not break encapsulation, and does not affect the types (all public interfaces remain unchanged)
- No need to involve in possibly complex hierarchy, and easy to understand and implement

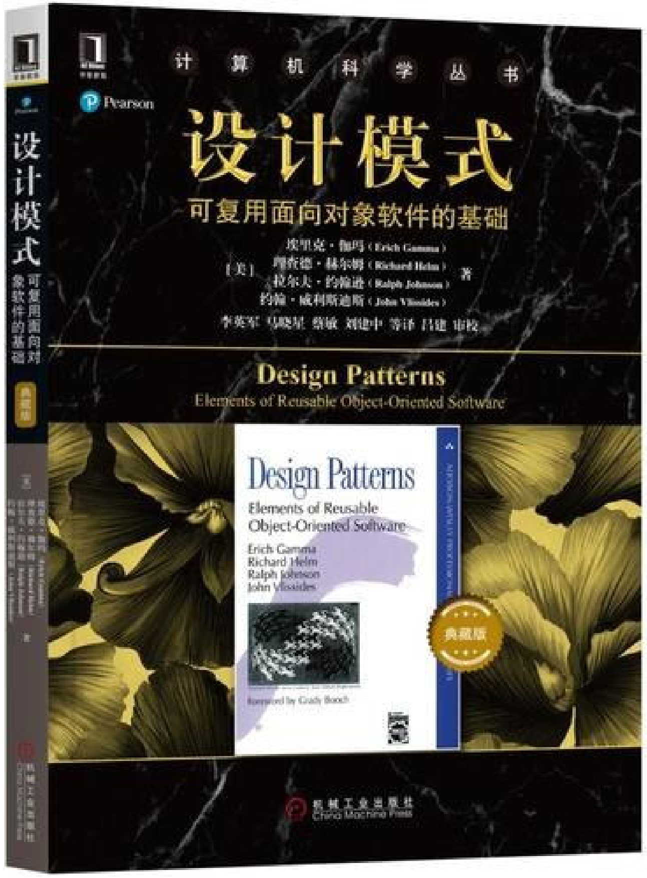
设计原则

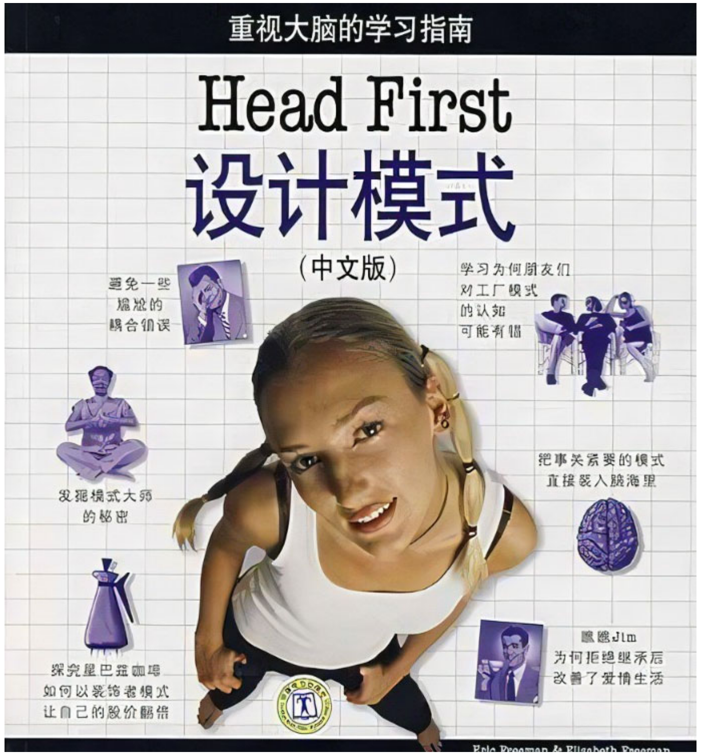
- 将应用中可能需要变化之处独立出来，不和那些不需要变化的代码混在一起
- 针对接口编程，而不是针对实现编程
- 多用组合，少用继承

设计模式（Design Pattern）

- 恭喜你！学会了第一个设计模式：策略模式（Strategy Pattern）
- 策略模式定义了算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

设计模式（Design Pattern）


计算机科学丛书
Pearson
设计模式
可复用面向对象软件的基础
［美］ 埃里克·伽玛（Erich Gamma） 理查德·赫尔姆（Richard Helm） 拉尔夫·约翰逊（Ralph Johnson） 约翰·威利斯迪斯（John Vlissides） 著
李英军 马晓星 蔡敏 刘建中 等译 吕建 审校
Design Patterns
Elements of Reusable Object-Oriented Software
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
典藏版
机械工业出版社
China Machine Press


重视大脑的学习指南
Head First
设计模式
（中文版）
避免一些尴尬的耦合错误
学习为何朋友们对工厂模式的认知可能有错
发掘模式大师的秘密
把重要的模式直接装入脑海里
探究星巴兹咖啡如何以装饰者模式让自己的股价翻倍
瞧瞧Jim为何拒绝继承后改善了爱情生活
Eric Freeman & Elisabeth Freeman

DDL提醒

- Lab08: 2022-12-03 23:59:00
- Proj03: 2022-12-04 23:59:00
- HW08: 2022-12-06 23:59:00