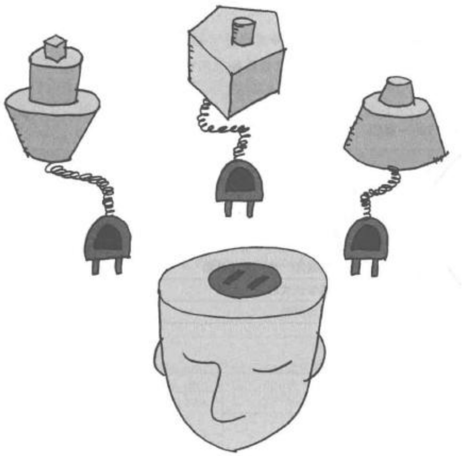
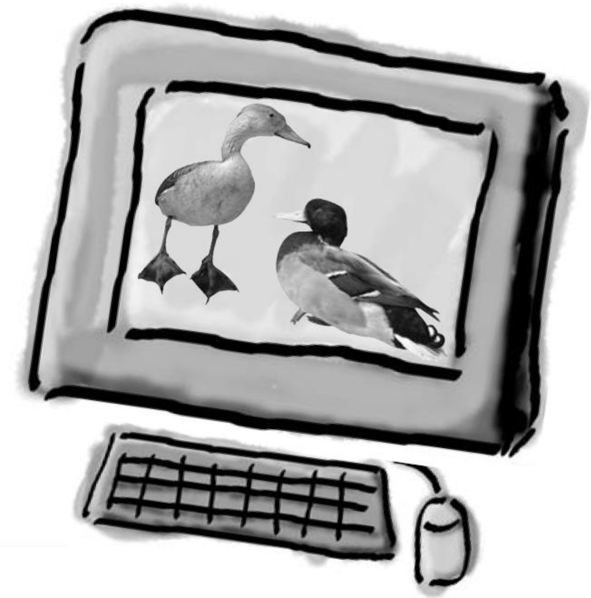
Strategy
(策略, Behavioral Pattern)



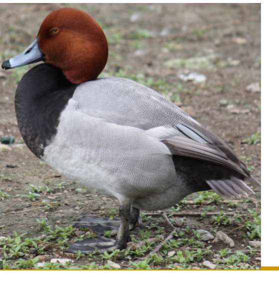
Kai SHI

Problem: A Duck Game

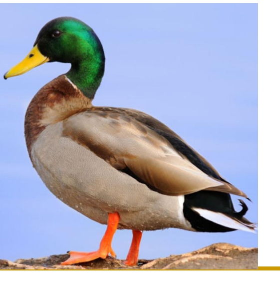


- The game can show a large variety of duck species swimming and making quacking sounds.
- Draw class diagram NOW

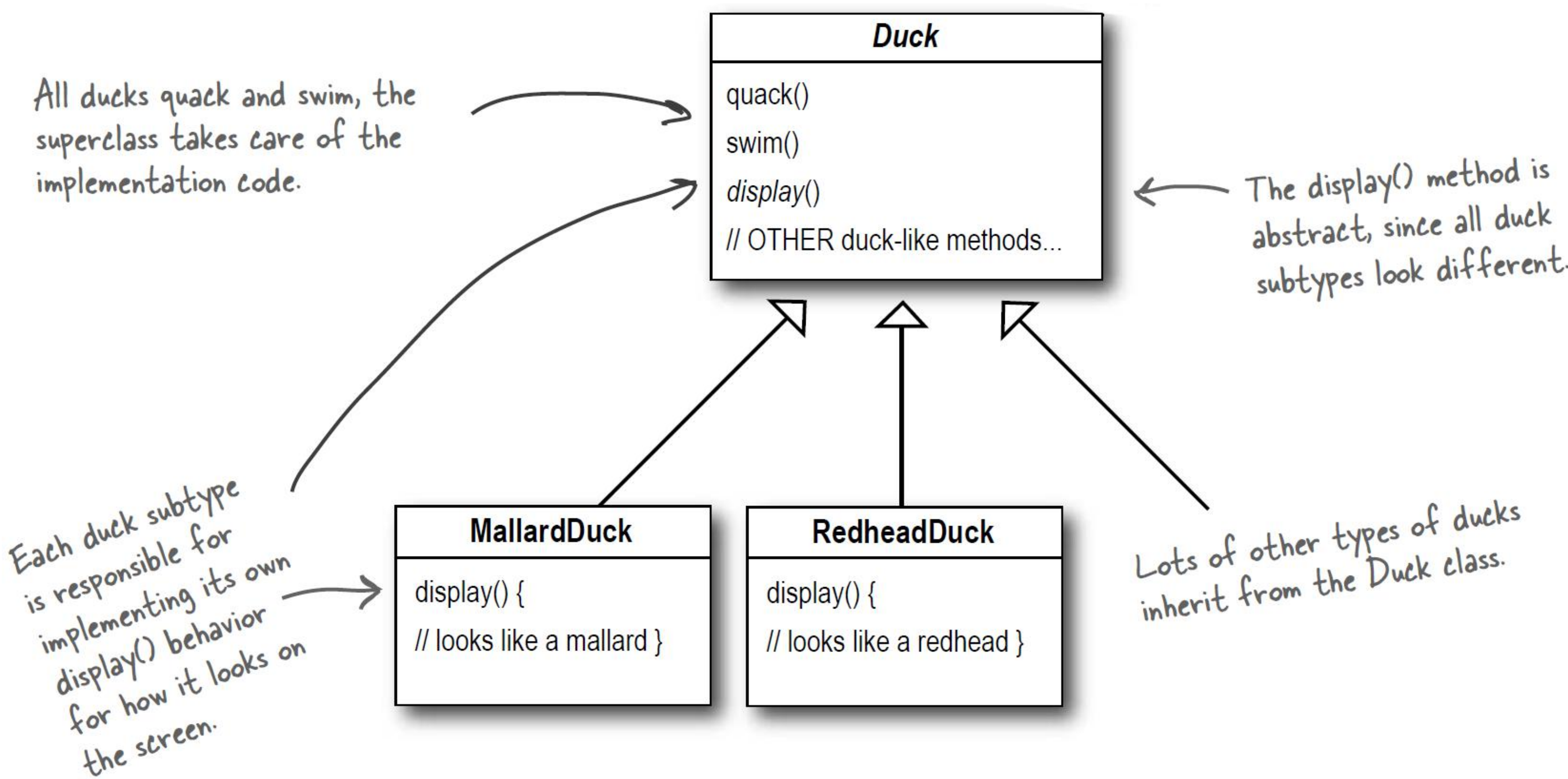
Redhead duck
（红头鸭）



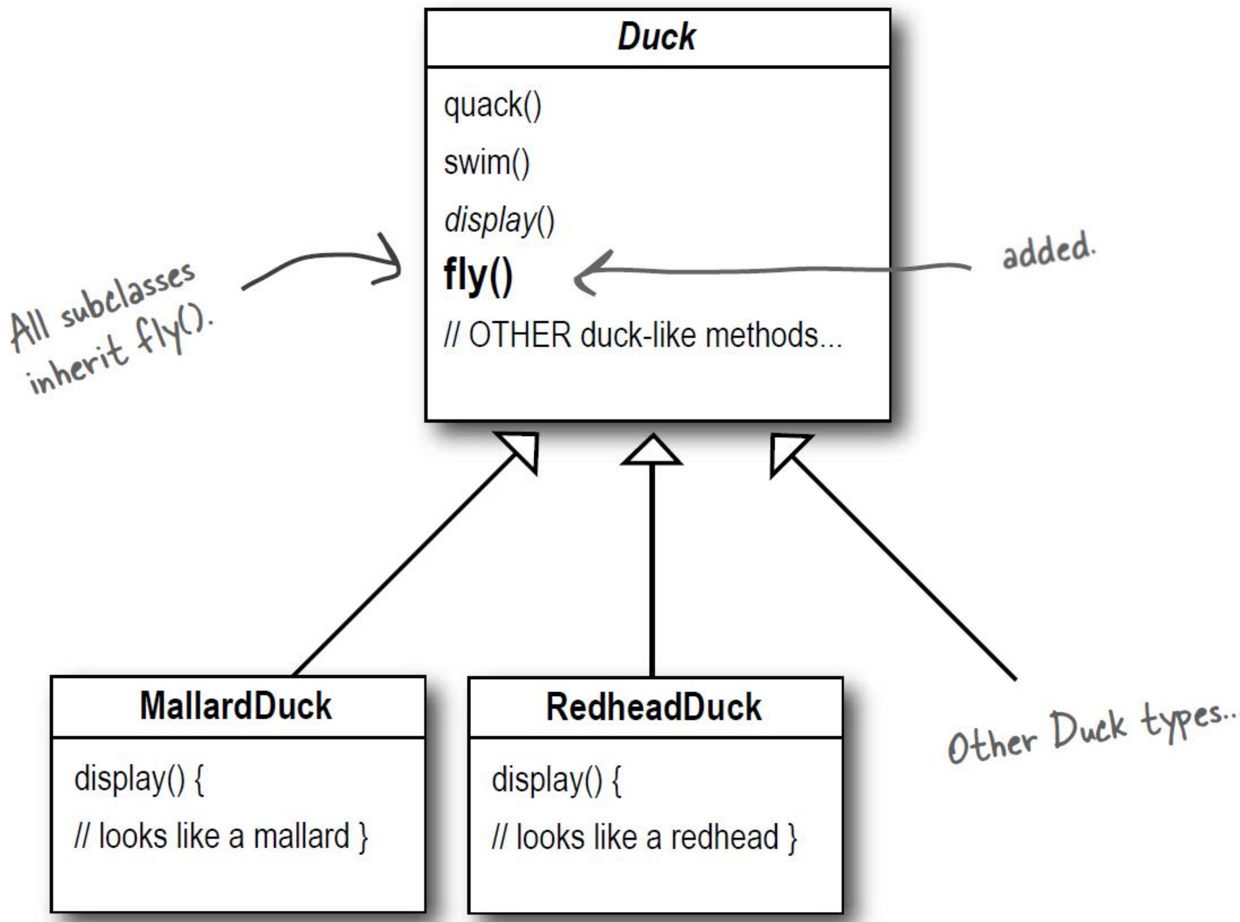
Mallard duck
（绿头鸭）



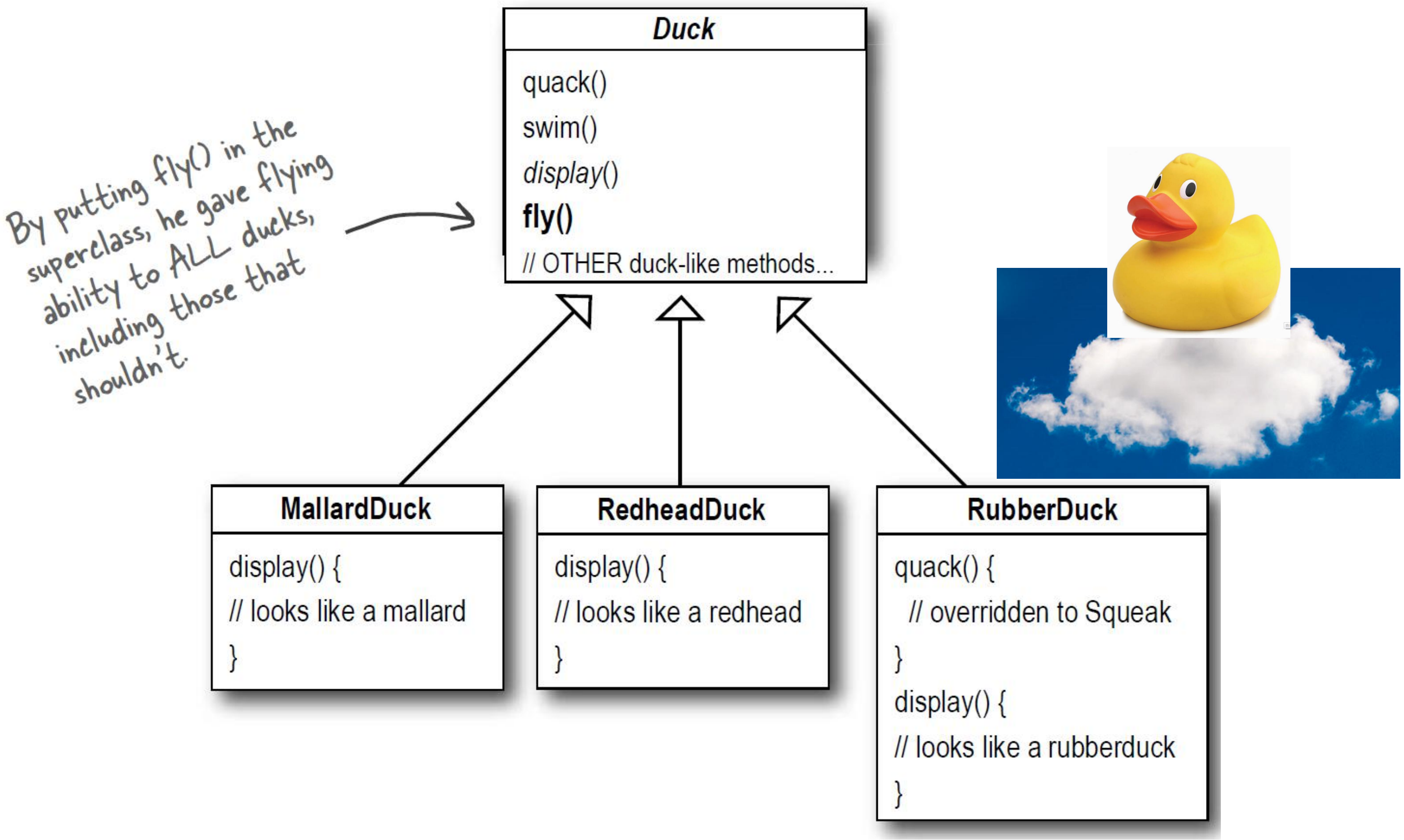
First Try: Class Diagram



Change: The Ducks Could Fly



Problem: Rubber Duckies fly



Solution: override the fly() method in rubber duck

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

Change: add wooden decoy (诱饵) ducks to the program

DecoyDuck

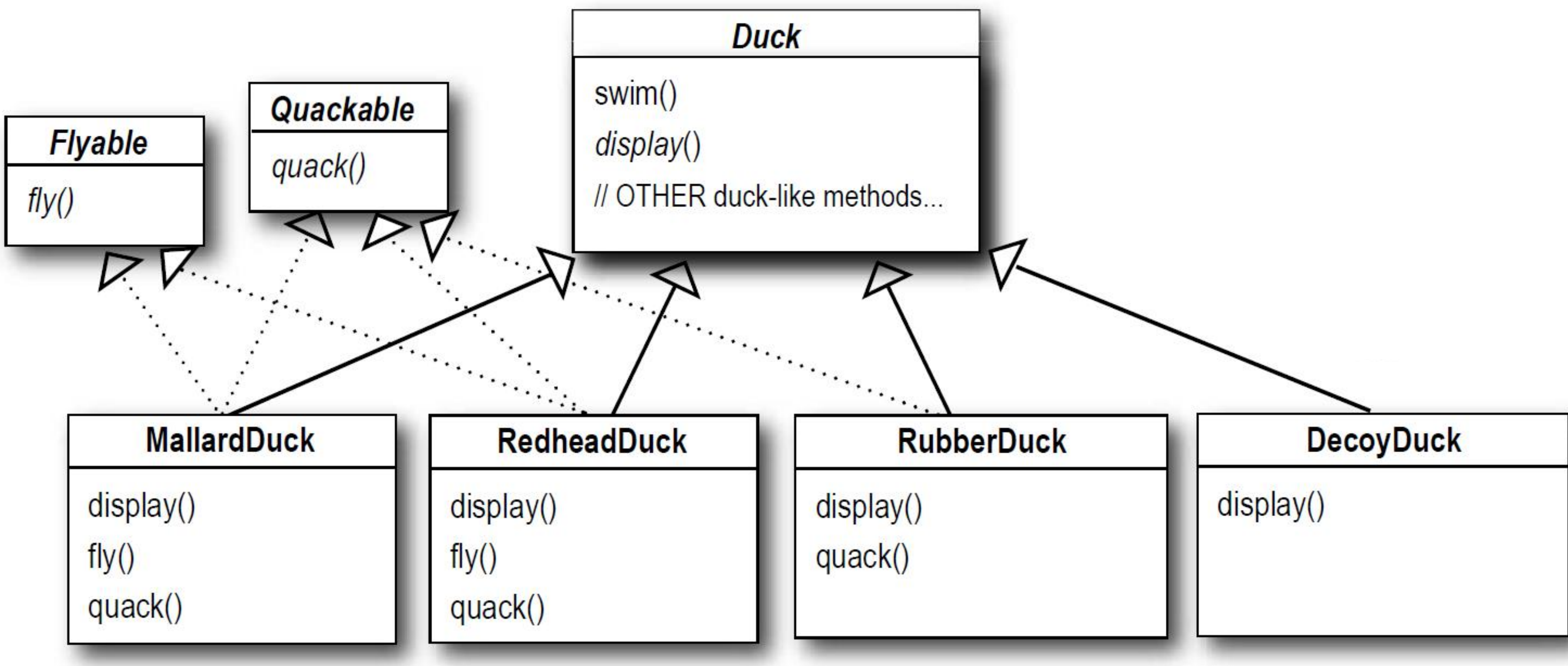
```
quack() {
  // override to do nothing
}

display() { // decoy duck}

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

Possibly override fly() and quack() for every new Duck subclass that's ever added to the program... forever.
Bad!

A New Design

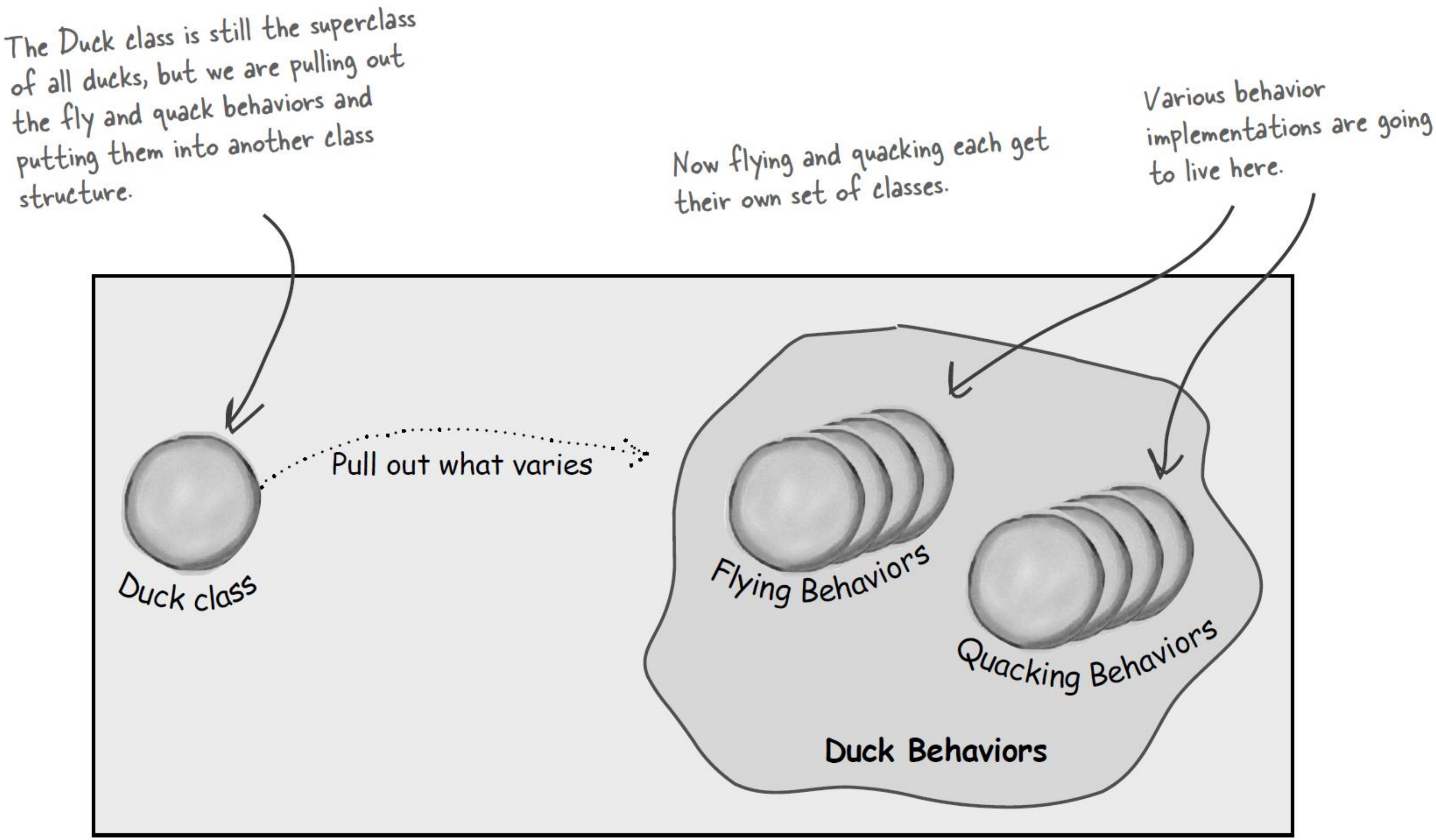


What do YOU think about this design?

Design Principle

- **Separating what changes from what stays the same**
  - (把可能变化的代码独立出来，不要和不变化的代码混在一起。)
  - Identify the aspects of your application that vary and separate them from what stays the same.
  - Take what varies and "encapsulate" it so it won't affect the rest of your code.

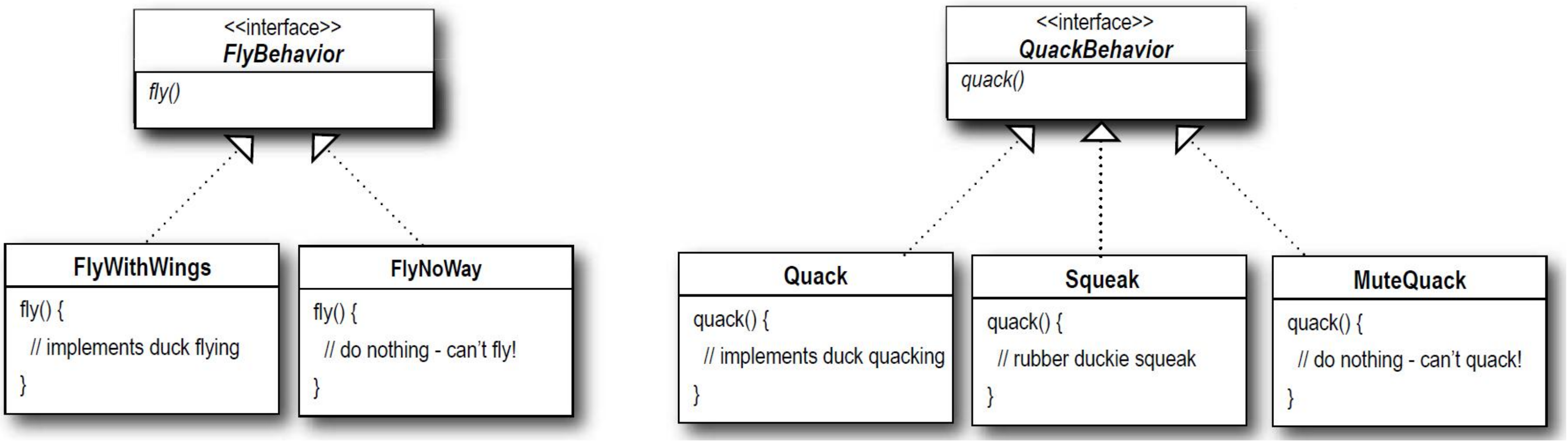
fly() and quack() are the parts of the Duck class that vary across ducks.
To separate these behaviors from the Duck class, we'll pull both methods out of the Duck class and create a new set of classes to represent each behavior.



Design Principle

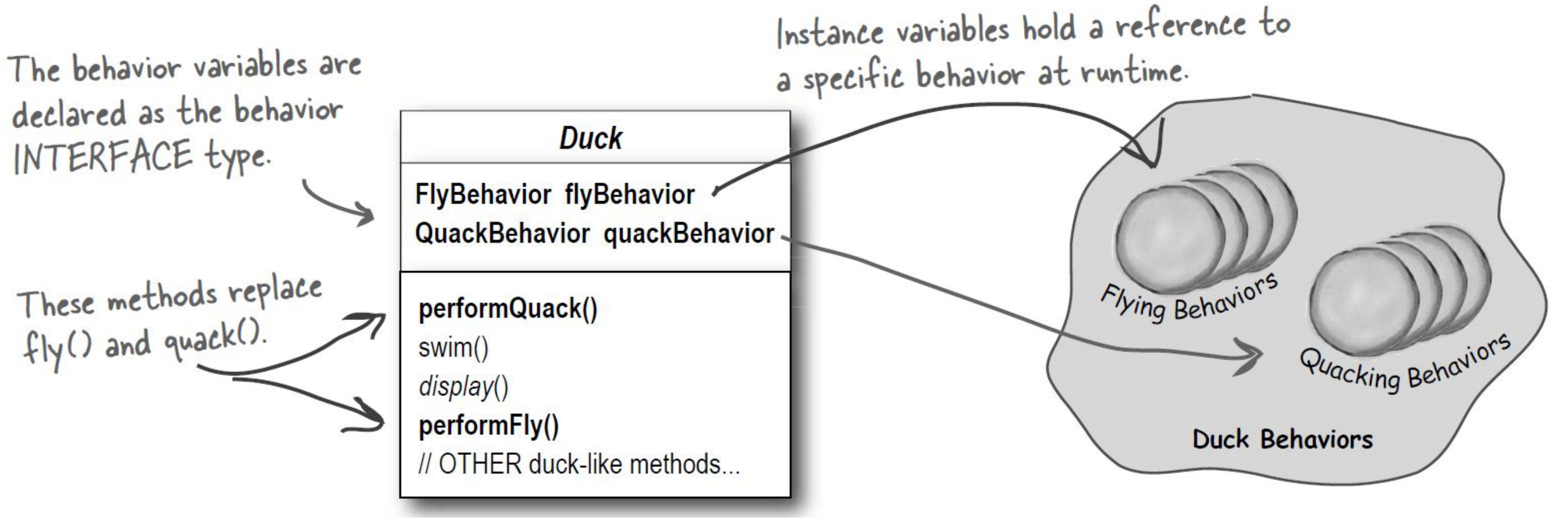
- **Program to an interface, not an implementation.**
  - (针对接口编程，而不是针对实现编程。)
  - This principle is to satisfy DIP: Dependence Inversion Principle

Class Diagrams of Behaviors

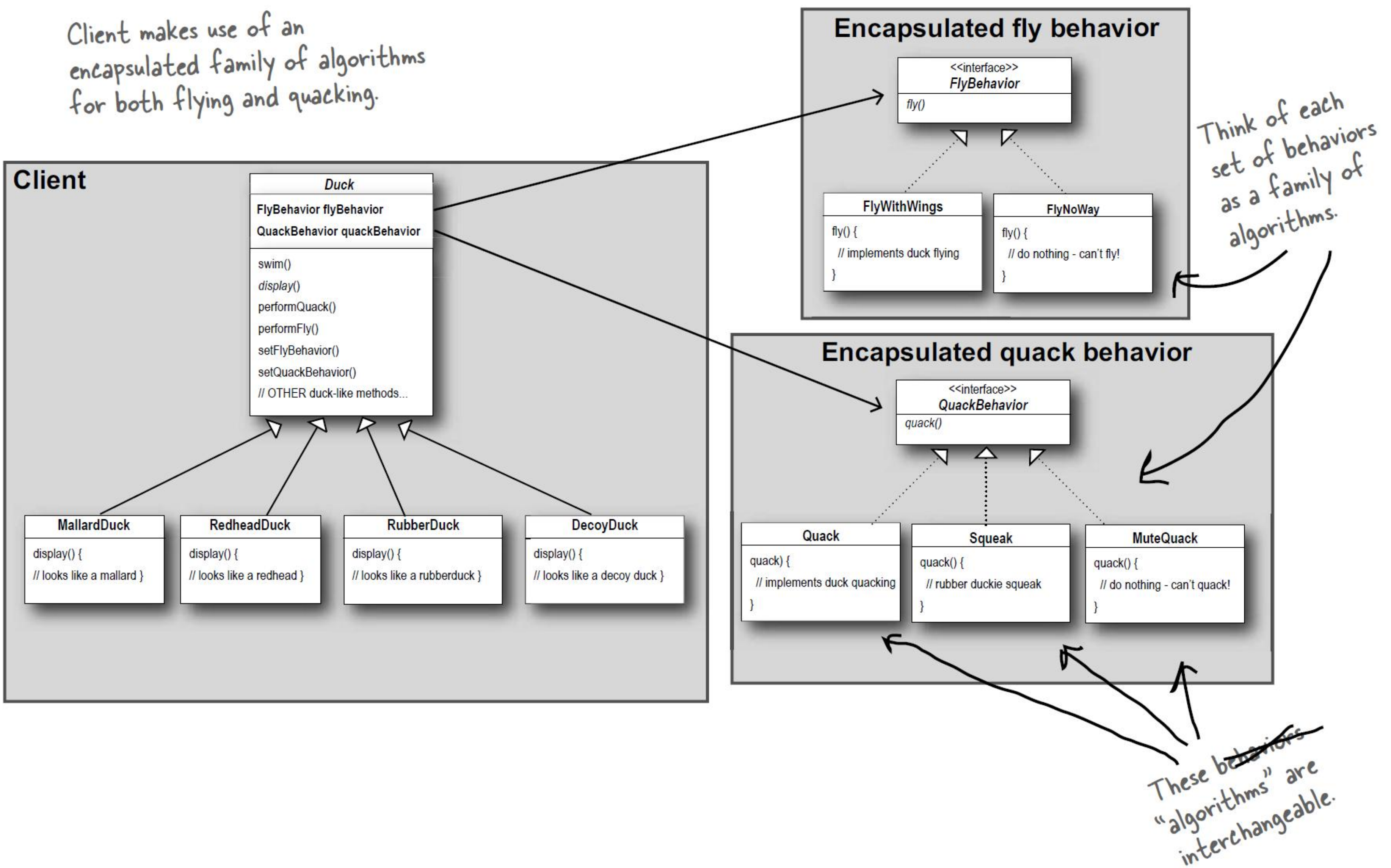


- With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in Duck classes!
- We can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.

Class Diagrams of Duck



Class Diagram of ALL



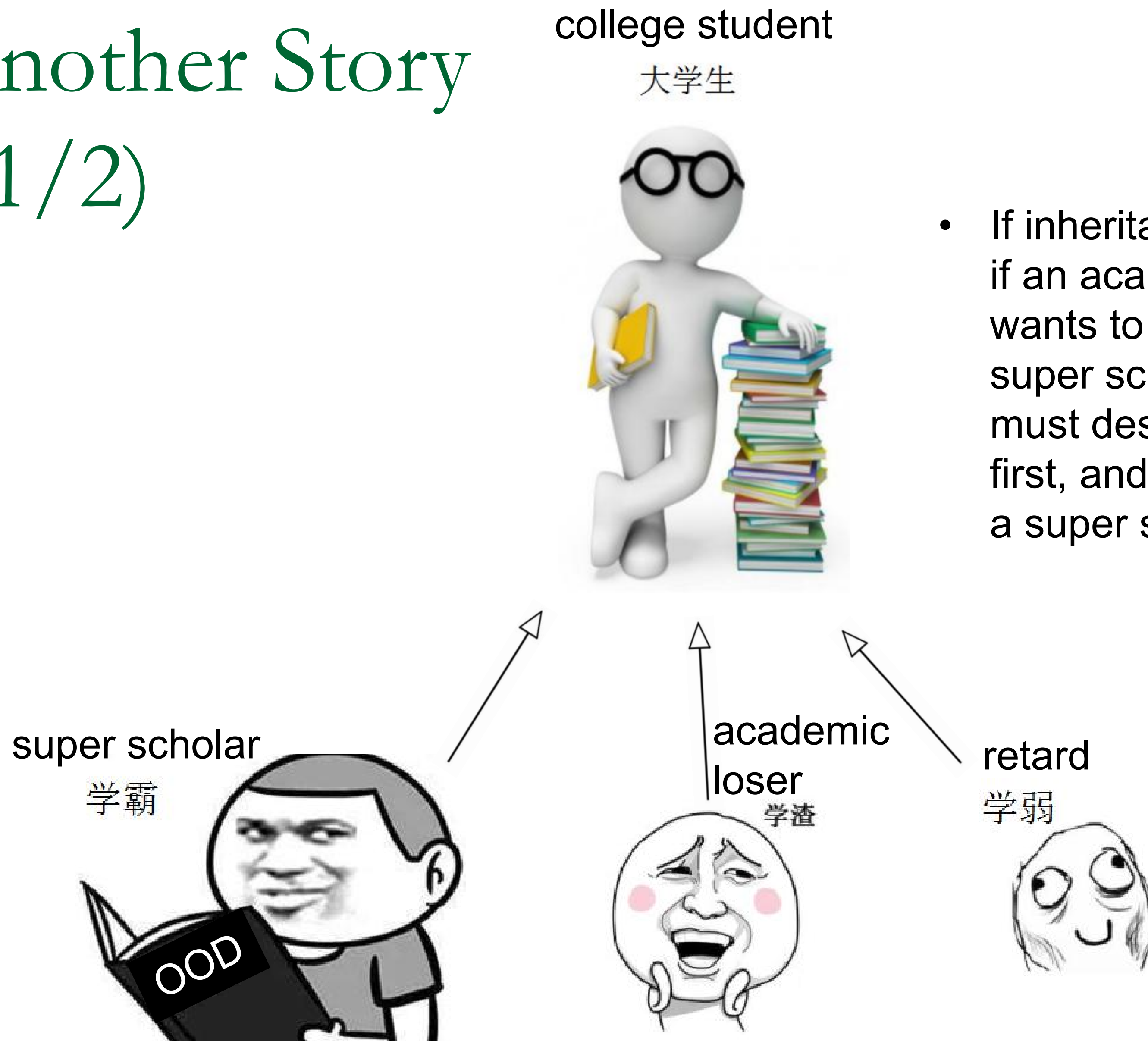
View Code Now

- code: net.dp.strategy
  - 先看ADuck.java

Design Principle

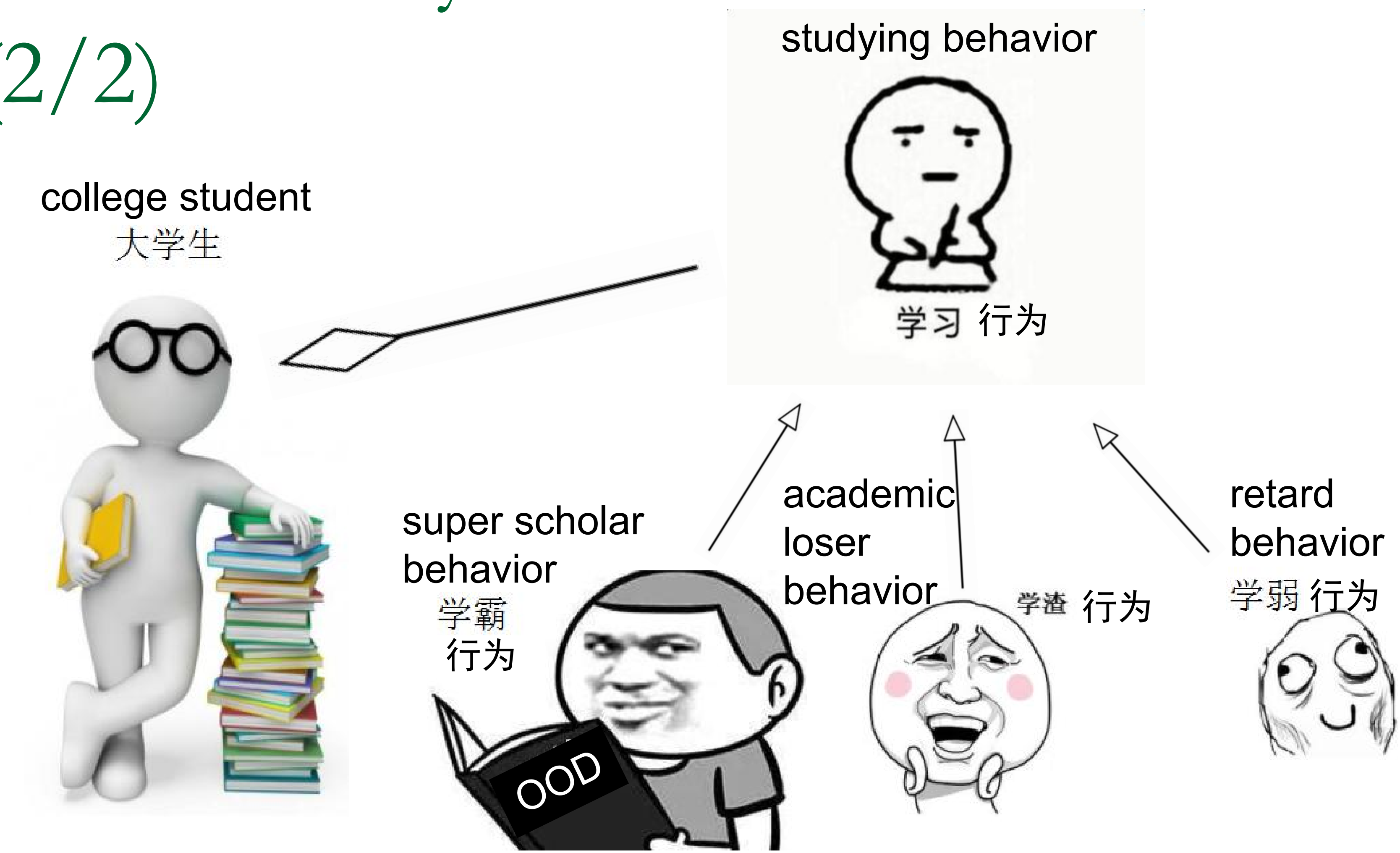
- **Favor composition over inheritance.**
  - (多用组合，少用继承。)
  - HAS-A can be better than IS-A
  - Creating systems using composition gives more flexibility. Not only does it encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime.

Another Story (1/2)



- If inheritance is used, if an academic loser wants to become a super scholar, he must destruct himself first, and re-instantiate a super scholar.

Another Story (2/2)



Strategy Pattern

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
  - (针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。客户决定使用哪个算法。)

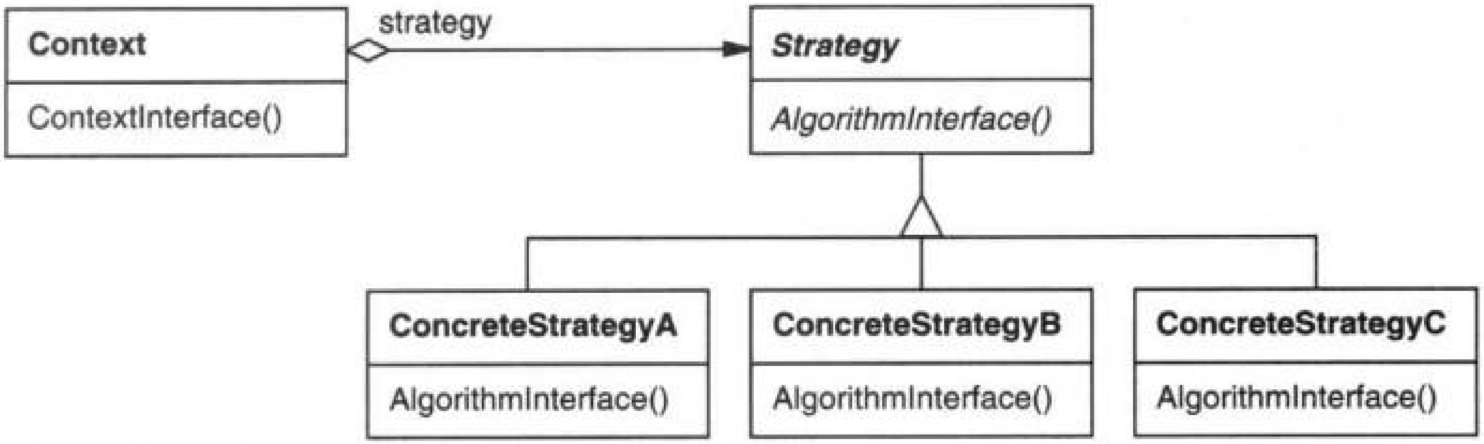
Strategy Pattern

- Also Known As
  - Policy
- Motivation
  - If clients support multiple algorithms, it makes clients bigger and harder to maintain.
  - Different algorithms will be appropriate at different times. We don't want to support multiple algorithms if we don't use them all.
  - It's difficult to add new algorithms and vary existing ones.

Applicability: Use strategy pattern when:

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
  - For example, you might define algorithms reflecting different space/time trade-offs.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class. (减少if语句的使用)

Class Diagram



Participants

- Strategy
  - Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy
  - implements the algorithm using the Strategy interface.
- Context
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Collaborations

- Strategy and Context interact to implement the chosen algorithm.
  - A context may pass all data required by the algorithm to the strategy when the algorithm is called.
  - Alternatively, the context can pass itself as an argument to strategy operations. That lets the strategy call back on the context as required.
- A Context forwards requests from its clients to its strategy.
  - There is often a family of ConcreteStrategy classes for a client to choose from.
  - Clients usually create and pass a ConcreteStrategy object to the context.
  - Clients interact with the context exclusively (唯一、排外地).

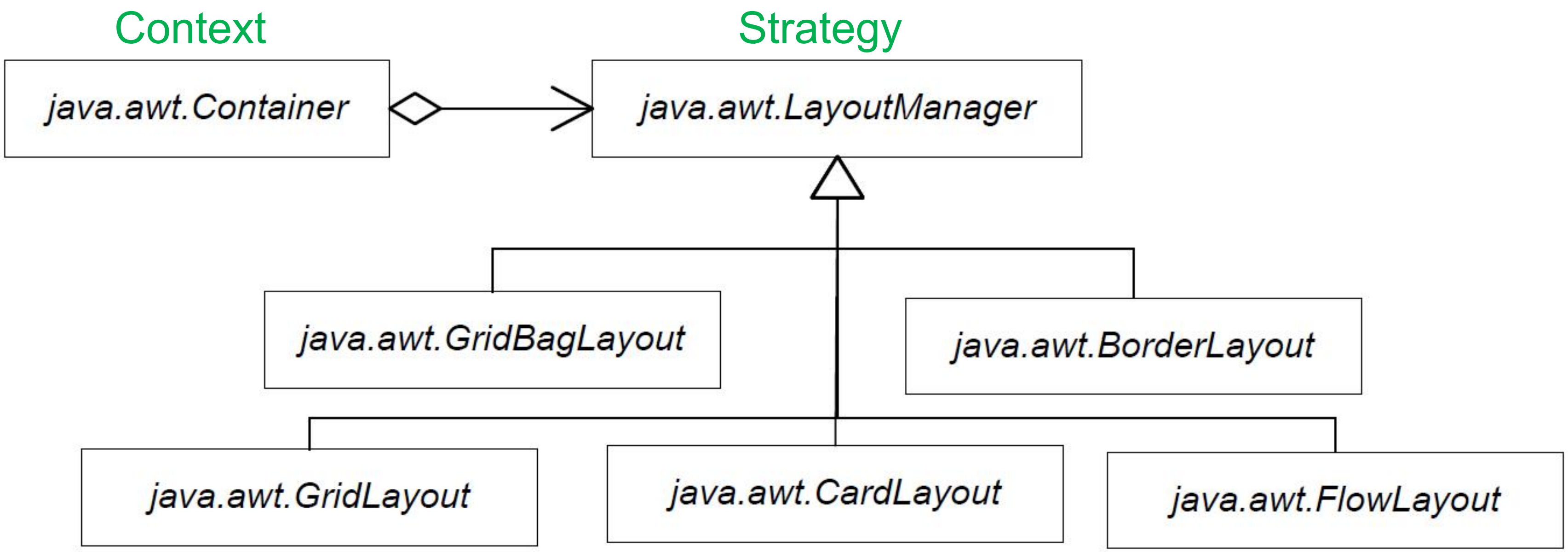
Consequences: benefits

- Families of related algorithms
  - Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse.
- An alternative to subclassing.
  - Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend. (CRP)
- Strategies eliminate conditional statements (if语句).
  - The Strategy pattern offers an alternative to conditional statements for selecting desired behavior.
- A choice of implementations.
  - Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.

Consequences: drawbacks

- Clients must be aware of different Strategies.
  - Client must understand how Strategies differ before it can select the appropriate one. (增加了客户使用难度)
- Communication overhead between Strategy and Context.
  - The Strategy is shared by all ConcreteStrategy classes whether they are simple or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed by Context;
- Increased number of objects.
  - Shared strategies which not maintain state across invocations can reduce the number of objects. (will learn that in Flyweight pattern)

Example: LayoutManager in AWT



Extension 1: **Passing data** between Context and Strategy

- The Strategy and Context must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
  - One approach is to have Context pass data in parameters to Strategy operations.
    - Simple approach
    - This keeps Strategy and Context decoupled.
    - Context might pass data the Strategy doesn't need.
  - A context pass itself as an argument, and the strategy requests data from the context explicitly.
    - Strategy can store a reference to its context, eliminating the need to pass anything at all.
    - Context must define a more elaborate interface to its data, which couples Strategy and Context more closely.

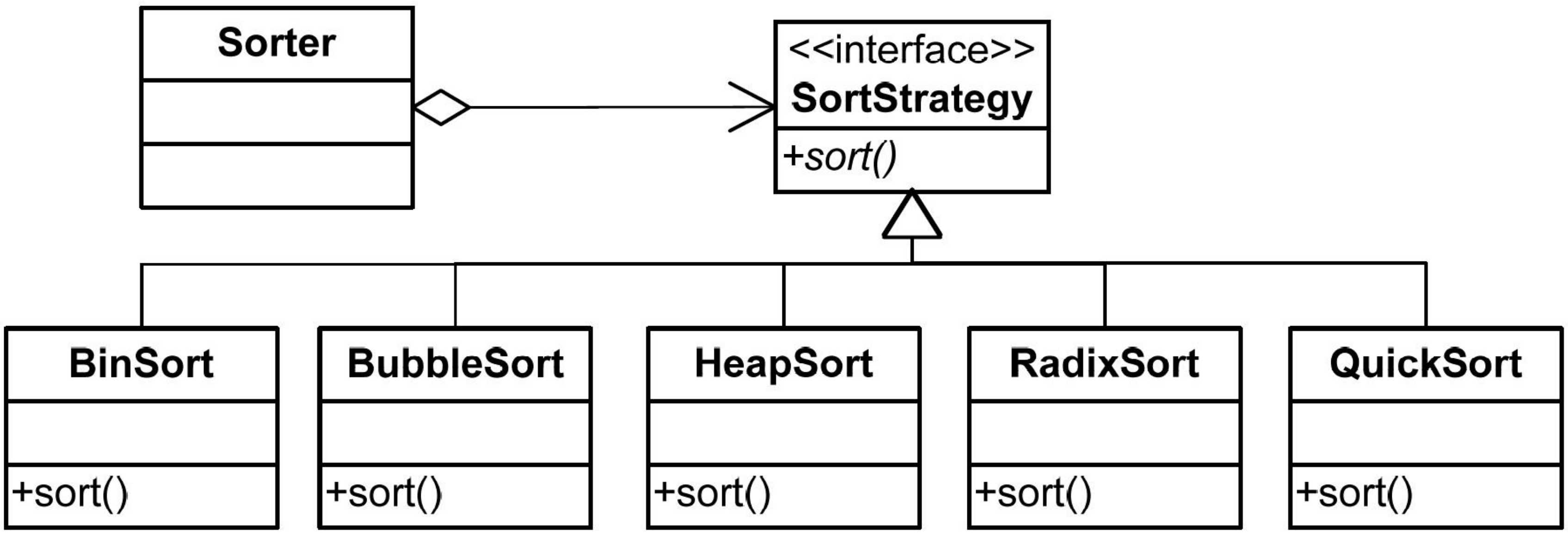
Extension 2: **Making** strategy objects **optional**

- The Context class may be simplified if it's meaningful not to have a Strategy object.
- Context checks to see if it has a Strategy object before accessing it.
  - If there is one, then Context uses it normally.
  - If there isn't a strategy, then Context carries out default behavior.
- Or it can be treated as an default Strategy is set up to the Context .

Another Question: Sorter System

- An container *data* stores some Strings.
- A sorter could sort these Strings using different sorting algorithms.
- Draw class diagram NOW.

Class Diagram



- Sample code: strategy.sort

Practice After Class: