Design Patterns

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 6 — 09/08/2011

Goals of the Lecture

- Introduce the concept of design patterns
  - Explain how it arose from the field of architecture and anthropology
- Discuss why design patterns are important and what advantages they provide
- Present an example of one design pattern
  - We saw an example of another design pattern—Delegation—in Lecture 4 and how it is used in iOS

Design Patterns are Everywhere (I)

- In 1995, a book was published by the "Gang of Four" called Design Patterns
  - It applied the concept of patterns (discussed next) to software design and described 23 of them
    - The authors did not invent these patterns
      - Instead, they included patterns they found in at least 3 "real" software systems.

Design Patterns are Everywhere (II)

- Since that time lots of DP books have been published
  - and more patterns have been cataloged
  - although many pattern authors abandoned the criteria of having to find the pattern in 3 shipping systems
- Unfortunately, many people feel like they should become experts in OO A&D **before** they learn about patterns
  - our book takes a different stance: learning about design patterns will help you become an expert in OO A&D

Cultural Anthropology

- Design Patterns have their intellectual roots in the discipline of cultural anthropology
  - Within a culture, individuals will agree on what is considered good design
    - "Cultures make judgements on good design that transcend individual beliefs"
  - **Patterns** (structures and relationships that appear over and over again in many different well designed objects) **provide an objective basis for judging design**

Christopher Alexander (I)

- Design patterns in software design traces its intellectual roots to work performed in the 1970s by an architect named Christopher Alexander
  - His 1979 book called "The Timeless Way of Building" that asks the question "Is quality objective?"
    - in particular, "What makes us know when an architectural design is good? Is there an objective basis for such a judgement?"
  - His answer was "yes" that it was possible to objectively define "high quality" or "beautiful" buildings

Christopher Alexander (II)

- He studied the problem of identifying what makes a good architectural design by observing all sorts of built structures
  - buildings, towns, streets, homes, community centers, etc.
- When he found an example of a high quality design, he would compare that object to other objects of high quality and look for commonalties
  - especially if both objects were used to solve the same type of problem

Christopher Alexander (III)

- By studying **high quality structures that solve similar problems**, he could discover **similarities between the designs** and these similarities where what he called **patterns**
  - "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."
  - The pattern provides an approach that can be used to achieve a high quality solution to its problem

Four Elements of a Pattern

- Alexander identified four elements to describe a pattern
  - The name of the pattern
  - The purpose of the pattern: what problem it solves
  - How to solve the problem
  - The constraints we have to consider in our solution
- He also felt that multiple patterns applied together can help to solve complex architectural problems

Design Patterns and Software (I)

- Work on design patterns got started when people asked
  - Are there problems in software that occur all the time that can be solved in somewhat the same manner?
  - Was it possible to design software in terms of patterns?
- Many people felt the answer to these questions was "yes" and this initial work eventually influenced the creation of the Design Patterns book by the Gang of Four
  - It catalogued 23 patterns: successful solutions to common problems that occur in software design

Design Patterns and Software (II)

- Design patterns, then, assert that the quality of software systems can be measured objectively
    - What is present in a good quality design (X's) that is not present in a poor quality design?
    - What is present in a poor quality design (Y's) that is not present in a good quality design?
- We would then want to maximize the X's while minimizing the Y's in our own designs

Key Features of a Pattern

- Name
- Intent: The purpose of the pattern
- Problem: What problem does it solve?
- Solution: The approach to take to solve the problem
- Participants: The entities involved in the pattern
- Consequences: The effect the pattern has on your system
- Implementation: Example ways to implement the pattern
- Structure: Class Diagram

Why Study Design Patterns? (I)

- Patterns let us
  - **reuse solutions that have worked in the past**; why waste time reinventing the wheel?
  - **have a shared vocabulary around software design**
    - they allow you to tell a fellow software engineer "I used a Strategy pattern here to allow the algorithm used to compute this calculation to be customizable"
      - You don't have to waste time explaining what you mean since you both know the Strategy pattern

Why Study Design Patterns? (II)

- Design patterns provide you **not with code reuse** but with **experience reuse**
  - **Knowing concepts** such as abstraction, inheritance and polymorphism will **NOT** make you a good designer, unless you use those concepts to **create flexible designs that are maintainable and that can cope with change**
- Design patterns can show you how to apply those concepts to achieve those goals

A Sense of Perspective

- Design Patterns give you a higher-level perspective on
  - the problems that come up in OO A&D work
  - the process of design itself
  - the use of object orientation to solve problems
- You'll be able to think more abstractly and not get bogged down in implementation details too early in the process

The Carpenter Analogy (I)

- The book has an excellent example of what they mean by a "higher-level perspective" by talking about two carpenters having a conversation
  - They can either say
    - Should we use a dovetail joint or a miter joint?
  - or
    - Should I make the joint by cutting down into the wood and then going back up 45 degrees and…

The Carpenter Analogy (II)

- The former is at a high-level and enables a richer conversation about the problem at hand
  - The latter gets bogged down in the details of cutting the wood such that you don't know what problem is being solved
- The former relies on the carpenter's **shared knowledge**
  - They know that dovetail joints are higher quality than miter joints but with higher costs
  - Knowing that, they can debate whether the higher quality is needed in the situation they are in

The Carpenter Analogy in Software

- "I have this one object with some important information and these other objects over here need to know when its information changes. These other objects come and go. I'm thinking I should separate out the notification and client registration functionality from the functionality of the object and just let it focus on storing and manipulating its information. Do you agree?"
- vs.
- "I'm thinking of using the Observer pattern. Do you agree?"

Other Advantages (I)

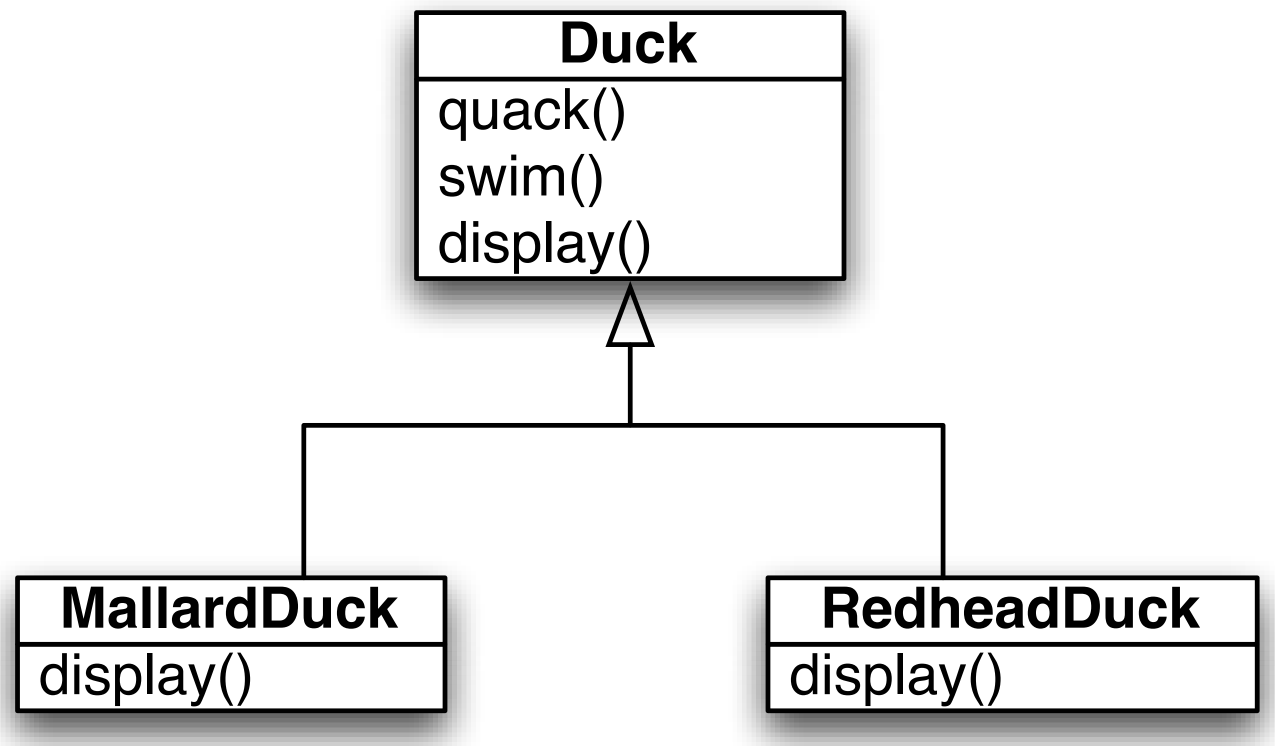
- Improved Motivation of Individual Learning in Team Environments
  - Junior developers see that the design patterns discussed by more senior developers are valuable and are motivated to learn them
- Improved maintainability
  - Many design patterns make systems easy to extend, leading to increased maintainability

Other Advantages (II)

- Design patterns lead to a deeper understanding of core OO principles
- They reinforce useful design heuristics such as
  - code to an interface
  - favor delegation over inheritance
  - find what varies and encapsulate it
- Since they favor delegation, they help you avoid the creation of large inheritance hierarchies, reducing complexity

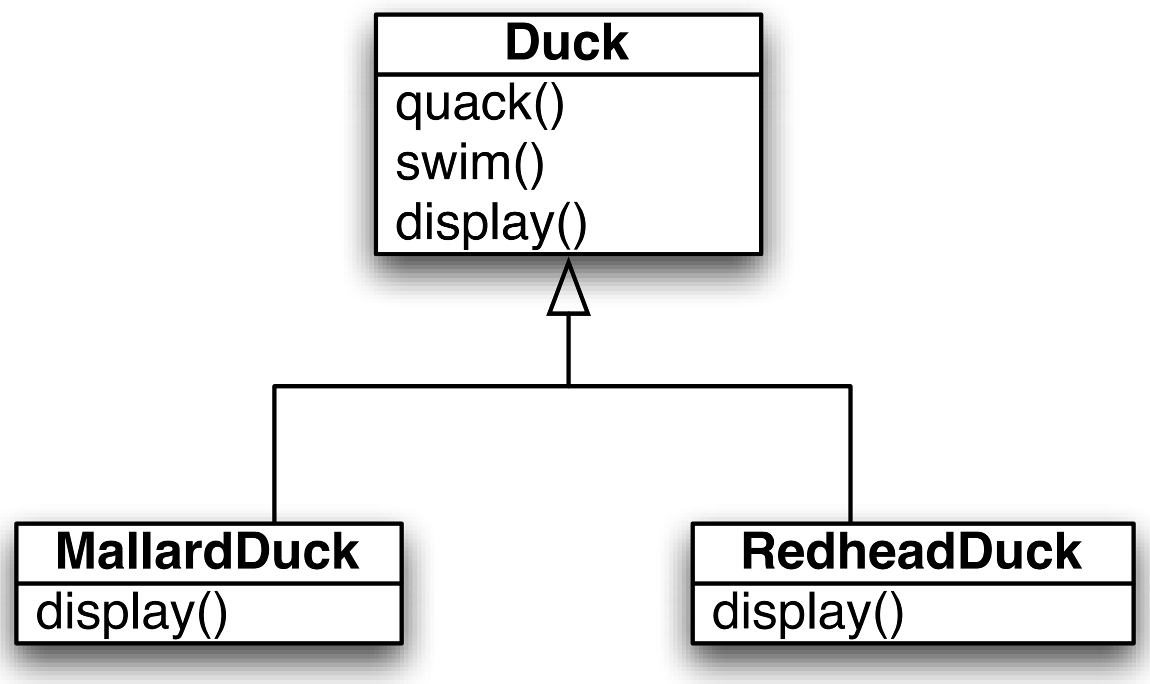
Design Pattern by Example

- SimUDuck: a “duck pond simulator” that can show a wide variety of duck species swimming and quacking
  - Initial State



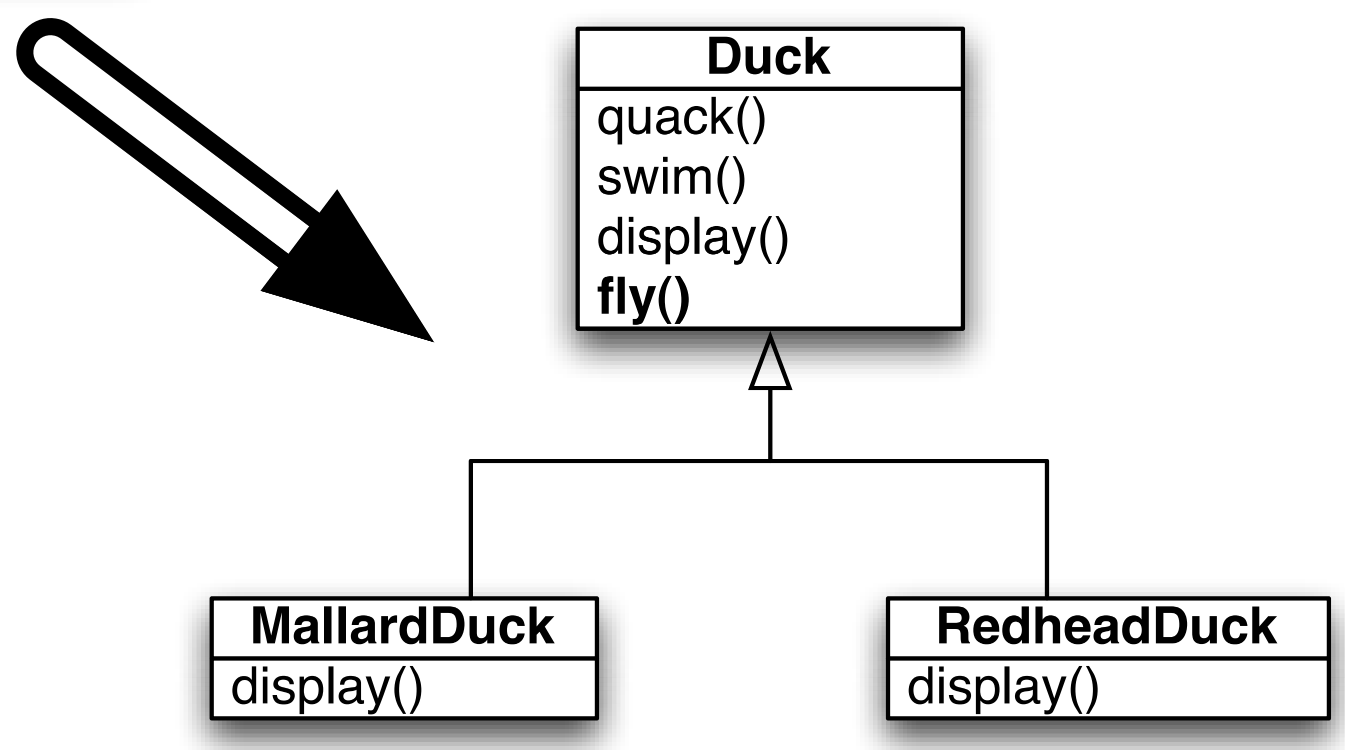
  - But a request has arrived to allow ducks to also fly. (We need to stay ahead of the competition!)

Easy

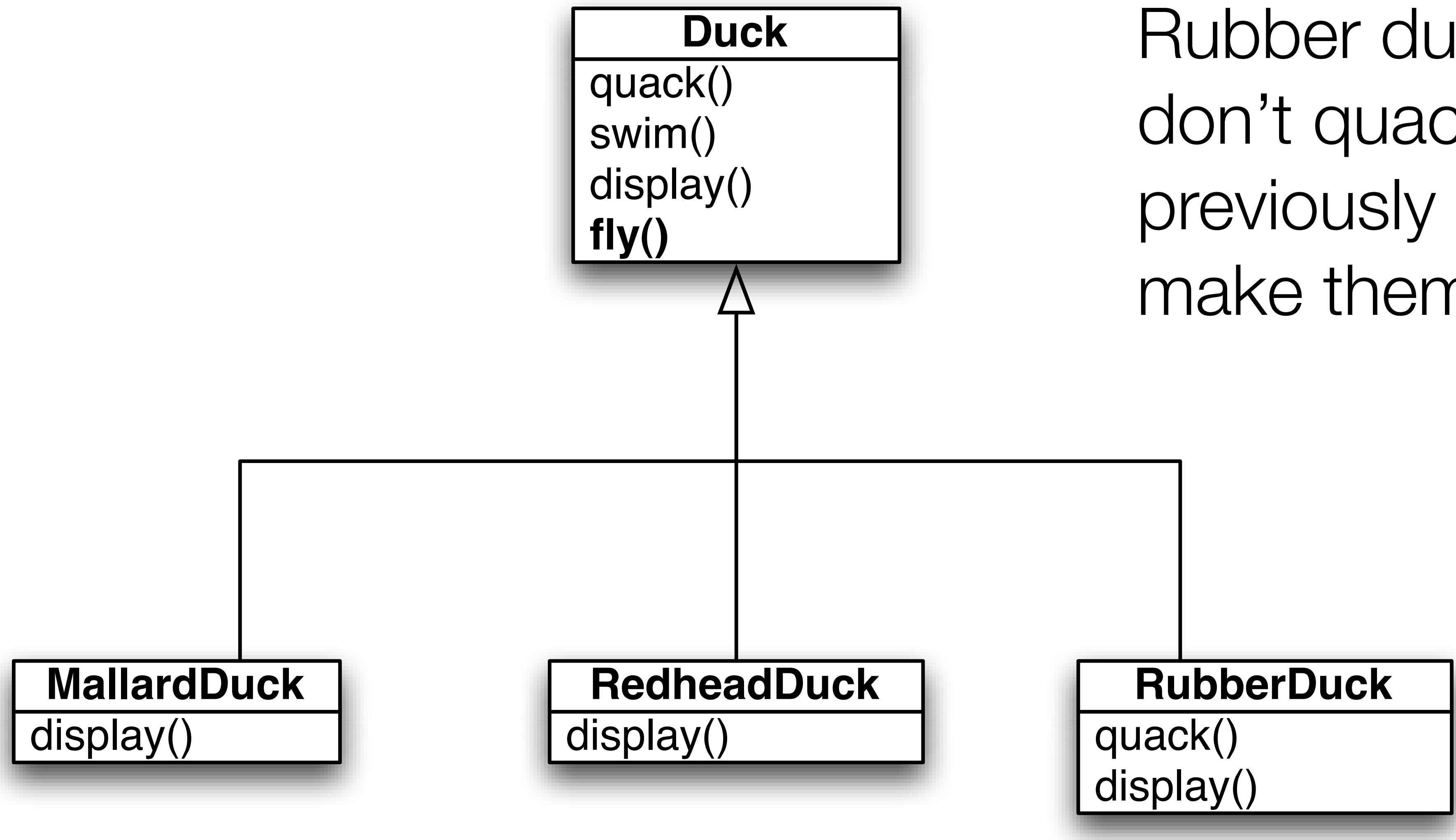


Code Reuse via Inheritance

Add fly() to Duck; all ducks can now fly



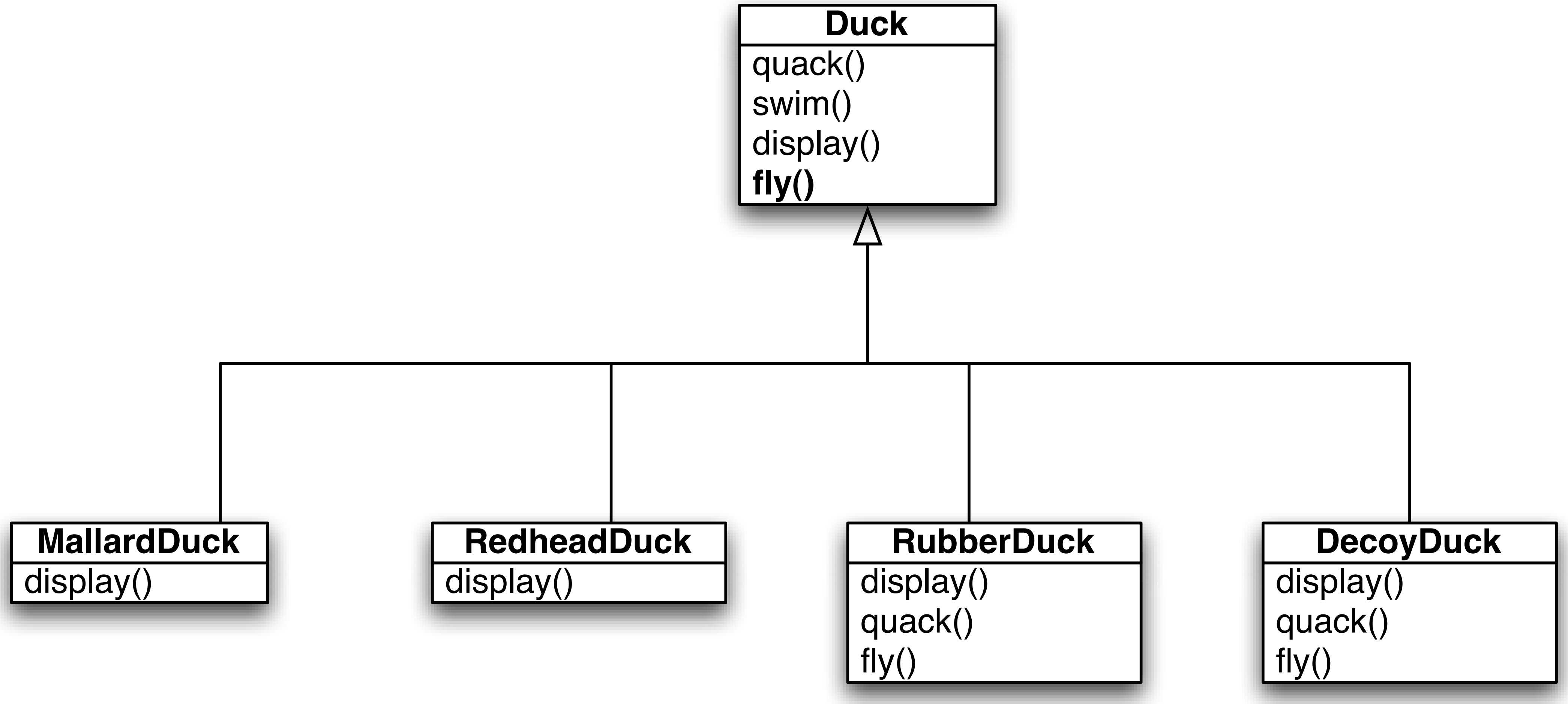
Whoops!



Rubber ducks do not fly! They don't quack either, so we had previously overridden quack() to make them squeak.

We could override fly() in RubberDuck to make it do nothing, but that's less than ideal, especially...

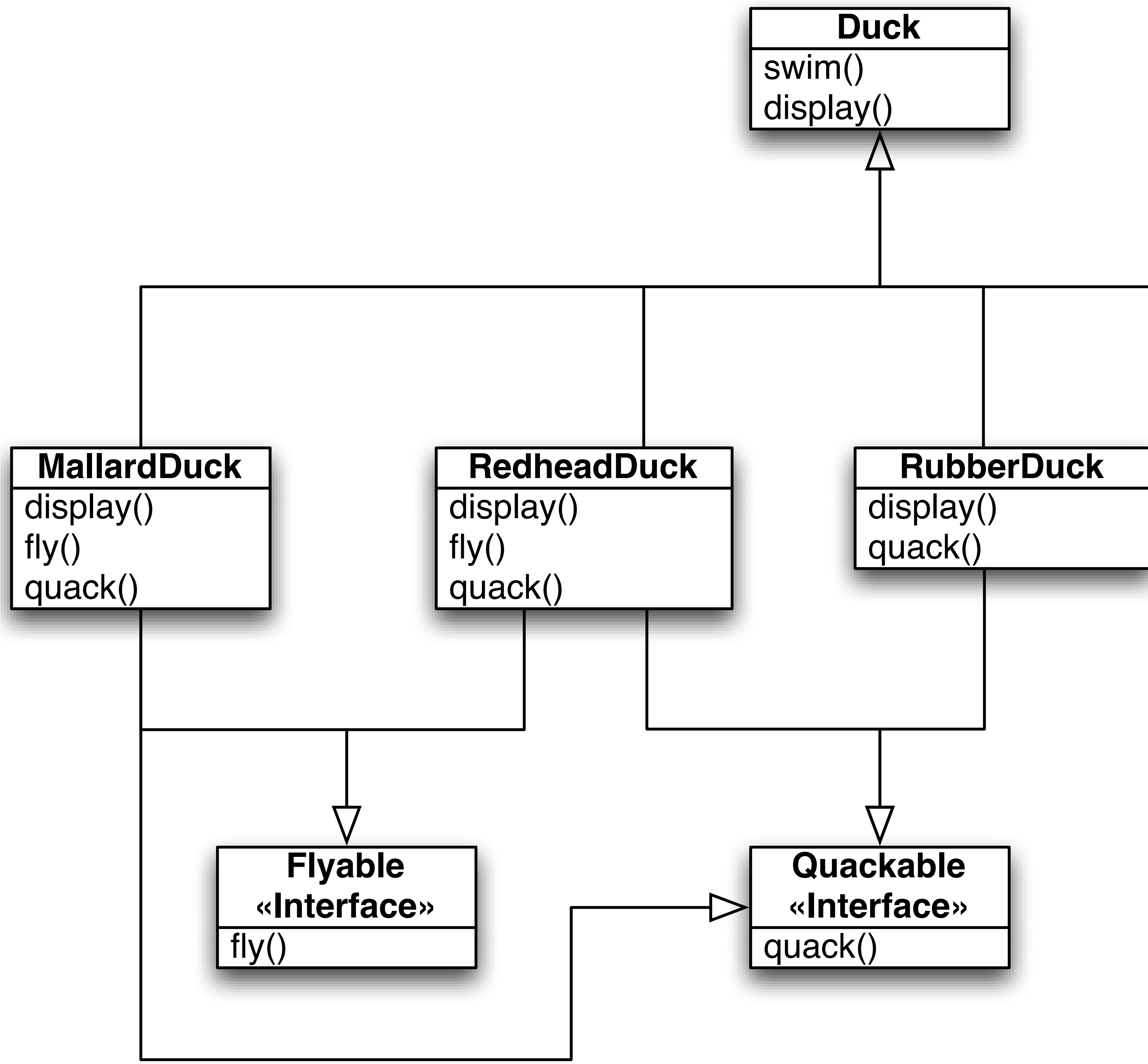
Double Whoops!



…when we might always find other Duck subclasses that would have to do the same thing.

What was supposed to be a good instance of **reuse via inheritance** has turned into a **maintenance headache**!

What about an Interface?



Here we define two interfaces and allow subclasses to implement the interfaces they need.

What are the trade-offs?

Design Trade-Offs

- With inheritance, we get
  - code reuse, only one fly() and quack() method vs. multiple (pro)
  - common behavior in root class, not so common after all (con)
- With interfaces, we get
  - specificity: only those subclasses that need a fly() method get it (pro)
  - no code re-use: since interfaces only define signatures (con)
- Use of abstract base class over an interface? Could do it, but only in languages that support multiple inheritance
  - In this approach, you implement Flyable and Quackable as abstract base classes and then have Duck subclasses use multiple inheritance

OO Principles to the Rescue!

- Encapsulate What Varies
  - For this particular problem, the “what varies” is the behaviors between Duck subclasses
    - We need to pull out behaviors that vary across subclasses and put them in their own classes (i.e. encapsulate them)
- The result: fewer unintended consequences from code changes (such as when we added fly() to Duck) and more flexible code

Basic Idea

- Take any behavior that varies across Duck subclasses and pull them out of Duck
  - Duck’s will no longer have fly() and quack() methods directly
  - Create two sets of classes, one that implements fly behaviors and one that implements quack behaviors
- Code to an Interface
  - We’ll make use of the “code to an interface” principle and make sure that each member of the two sets implements a particular interface
    - For QuackBehavior, we’ll have Quack, Squeak, Silence
    - For FlyBehavior, we’ll have FlyWithWings, CantFly, FlyWhenThrown, …
- Additional benefits
  - Other classes can gain access to these behaviors (if that makes sense) and we can add additional behaviors without impacting other classes

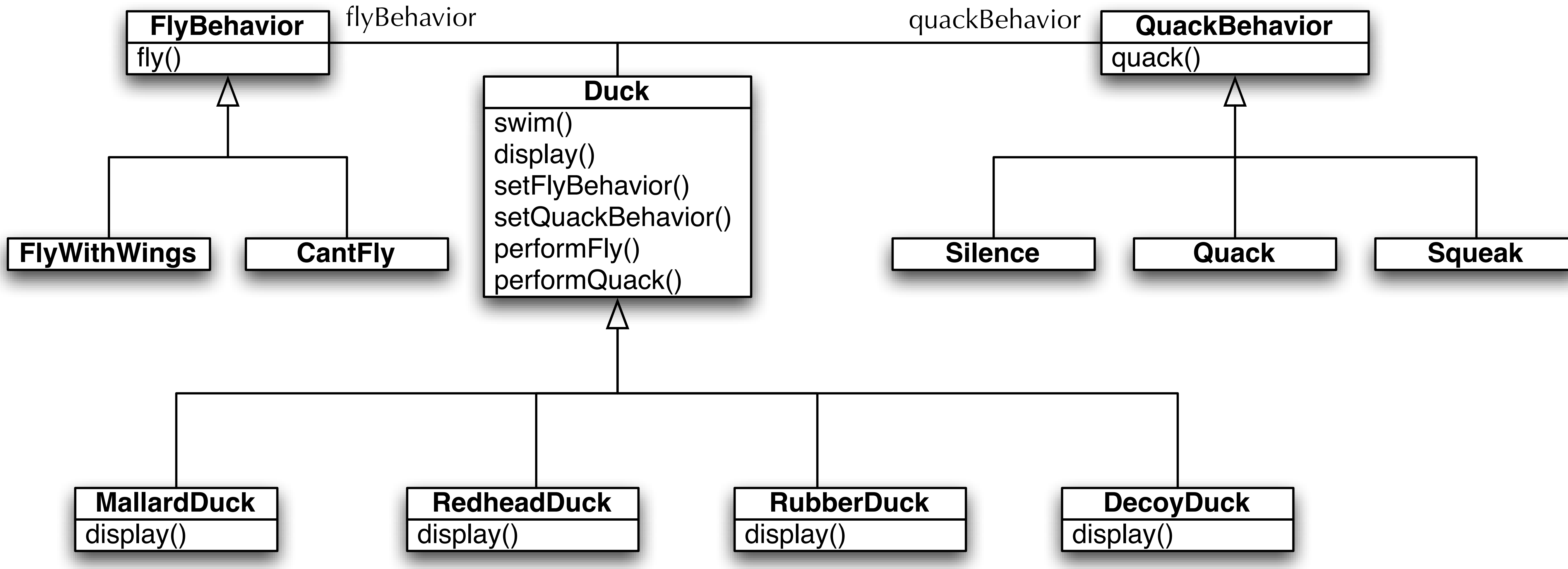
“Code to Interface” Does NOT Imply Java Interface

- We are overloading the word “interface” when we say “code to an interface”
  - We can implement “code to an interface” by defining a Java interface and then have various classes implement that interface
  - Or, we can “code to a supertype” and instead define an abstract base class which classes can access via inheritance.
- When we say “code to an interface” it implies that the object that is using the interface will have a variable whose type is the supertype (whether its an interface or abstract base class) and thus
  - can point at any implementation of that supertype
  - and is shielded from their specific class names
    - A Duck will point to a fly behavior with a variable of type FlyBehavior NOT FlyWithWings; the code will be more loosely coupled as a result

Bringing It All Together: Delegation

- To take advantage of these new behaviors, we must modify Duck to delegate its flying and quacking behaviors to these other classes
    - rather than implementing this behavior internally
- We'll add two attributes that store the desired behavior and we'll rename fly() and quack() to performFly() and performQuack()
    - this last step is meant to address the issue of it not making sense for a DecoyDuck to have methods like fly() and quack() directly as part of its interface
        - Instead, it inherits these methods and plugs-in CantFly and Silence behaviors to make sure that it does the right thing if those methods are invoked
- This is an instance of the principle "Favor composition over inheritance"

New Class Diagram



FlyBehavior and QuackBehavior define a set of behaviors that provide behavior to Duck. Duck is composing each set of behaviors and can switch among them dynamically, if needed. While each subclass now has a performFly() and performQuack() method, at least the user interface is uniform and those methods can point to null behaviors when required.

Duck.java

```java
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }

    public void setFlyBehavior (FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

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

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

Note: “code to interface”, delegation, encapsulation, and ability to change behaviors dynamically

DuckSimulator.java (Part 1)

```java
  public static void main(String[] args) {

    List<Duck> ducks  = new LinkedList<Duck>();

    Duck model = new ModelDuck();

    ducks.add(new DecoyDuck());
    ducks.add(new MallardDuck());
    ducks.add(new RedHeadDuck());
    ducks.add(new RubberDuck());
    ducks.add(model);

    processDucks(ducks);

    // change the Model Duck's behavior dynamically
    model.setFlyBehavior(new FlyRocketPowered());
    model.setQuackBehavior(new Squeak());

    processDucks(ducks);
  }
}

```

Note: all variables are of type Duck, not the specific subtypes; “code to interface” in action

Note: here we see the power of delegation. We can change behaviors at run-time

DuckSimulator.java (Part 2)

```java
import java.util.LinkedList;
import java.util.List;

public class DuckSimulator {

  public static void processDucks(List<Duck> ducks) {
    for (Duck d : ducks) {
      System.out.println("----------------------------------------");
      System.out.println("Name: " + d.getClass().getName());
      d.display();
      d.performQuack();
      d.performFly();
      d.swim();
    }
  }
```

Because of abstraction and polymorphism, processDucks() consists of nice, clean, robust & extensible code!

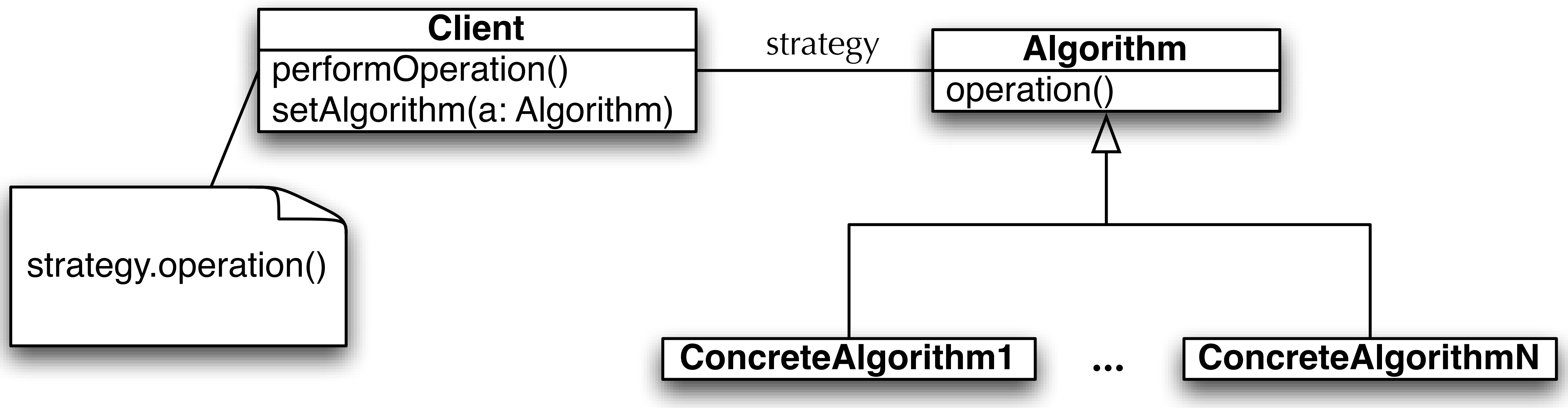
Not Completely Decoupled

- Is DuckSimulator completely decoupled from the Duck subclasses?
  - All of its variables are of type Duck
- No!
  - The subclasses are still coded into DuckSimulator
    - Duck mallard = **new MallardDuck()**;
- This is a type of coupling… fortunately, we can eliminate this type of coupling if needed, using a pattern called **Factory**.
  - We'll see Factory in action later this semester

Meet the Strategy Design Pattern

- The solution that we applied to this design problem is known as the Strategy Design Pattern
  - It features the following OO design concepts/principles:
    - Encapsulate What Varies
    - Code to an Interface
    - Delegation
    - Favor Composition over Inheritance
- Definition: The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it

Structure of Strategy



Algorithm is pulled out of Client. Client only makes use of the public interface of Algorithm and is not tied to concrete subclasses.

Client can change its behavior by switching among the various concrete algorithms

Wrapping Up

- Design Patterns
  - let us reuse existing, high-quality solutions to commonly recurring problems
  - establish a shared vocabulary to improve communication among teams
    - as well as raise the level of our engineering discipline
  - Provide designers with a higher perspective on the problems that occur within design and how to discuss them, how to solve them, how to consider trade-offs

Coming Up Next

- Homework 3 Assigned on Friday
- Lecture 7: Facade and Adapter
  - Read Chapters 6 and 7 of the textbook
- Homework 3 Due on the following Friday
- Lecture 8: Expanding Horizons
  - Read Chapter 8 of the textbook