Design Patterns

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 11

Acknowledgement & Materials Copyright

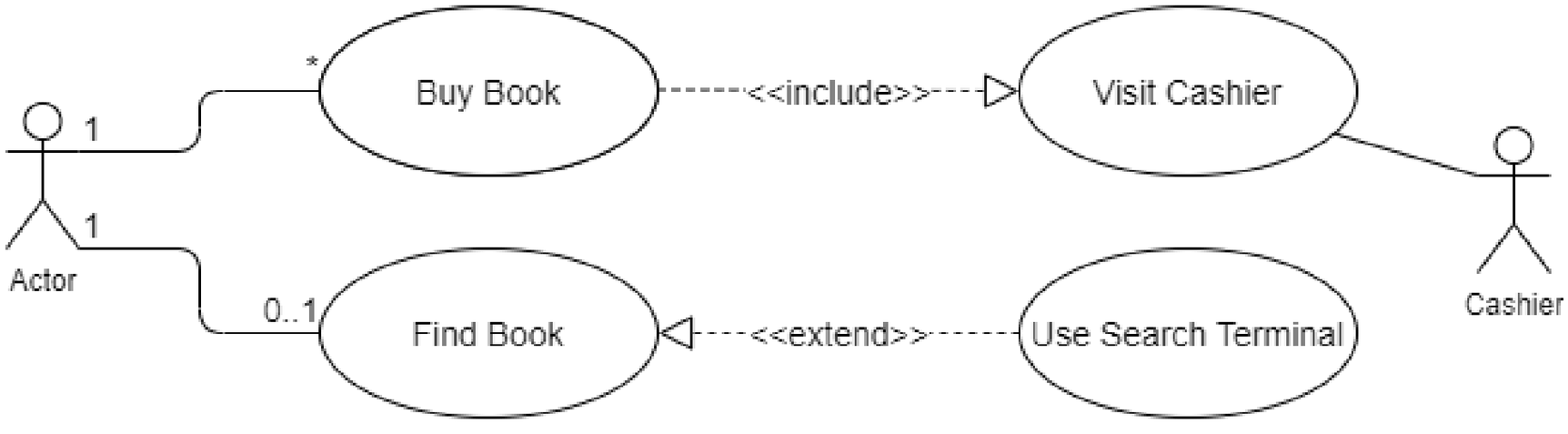
- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Before we start… Quick review

- Structured vs. Functional vs. OO Design & Programming
- Functional Decomposition
- Abstraction
- Encapsulation
- Classes, methods, attributes, instances, objects
- Inheritance (Is-A, superclass to subclass)
- Polymorphism
- Requirements (functional, non-functional, constraints)
- The “V”
- Cohesion & Coupling
- Conceptual, Specification, Implementation Perspectives
- Accessibility (Public, Protected, Private)
- Constructors/Destructors
- Abstract Classes
- Interfaces
- Collaborations
- UML Class, Sequence, State, Activity, Use Case Diagrams
- The WAVE rule for Use Cases
- Association & Multiplicity
- Aggregation, Composition, and Existence Dependency
- Qualification
- Delegation (Has-A)
- Design by Contract
- Object Identity and Equality

Before we start: UML & Use Cases

- There was a good question about the extends and includes line/arrow styles – if you see use case diagrams on the web, you'll see some variation
- The UML 2.5.1 standard (here) says this:
  - Lines indicating relationships between Use Cases can show multiplicity
  - Includes and Extends Use Cases use dashed arrows
  - The Include arrow points from the base Use Case (task) to the included Use Case
  - The Extend arrow points from the extending Use Case towards the extended Use Case



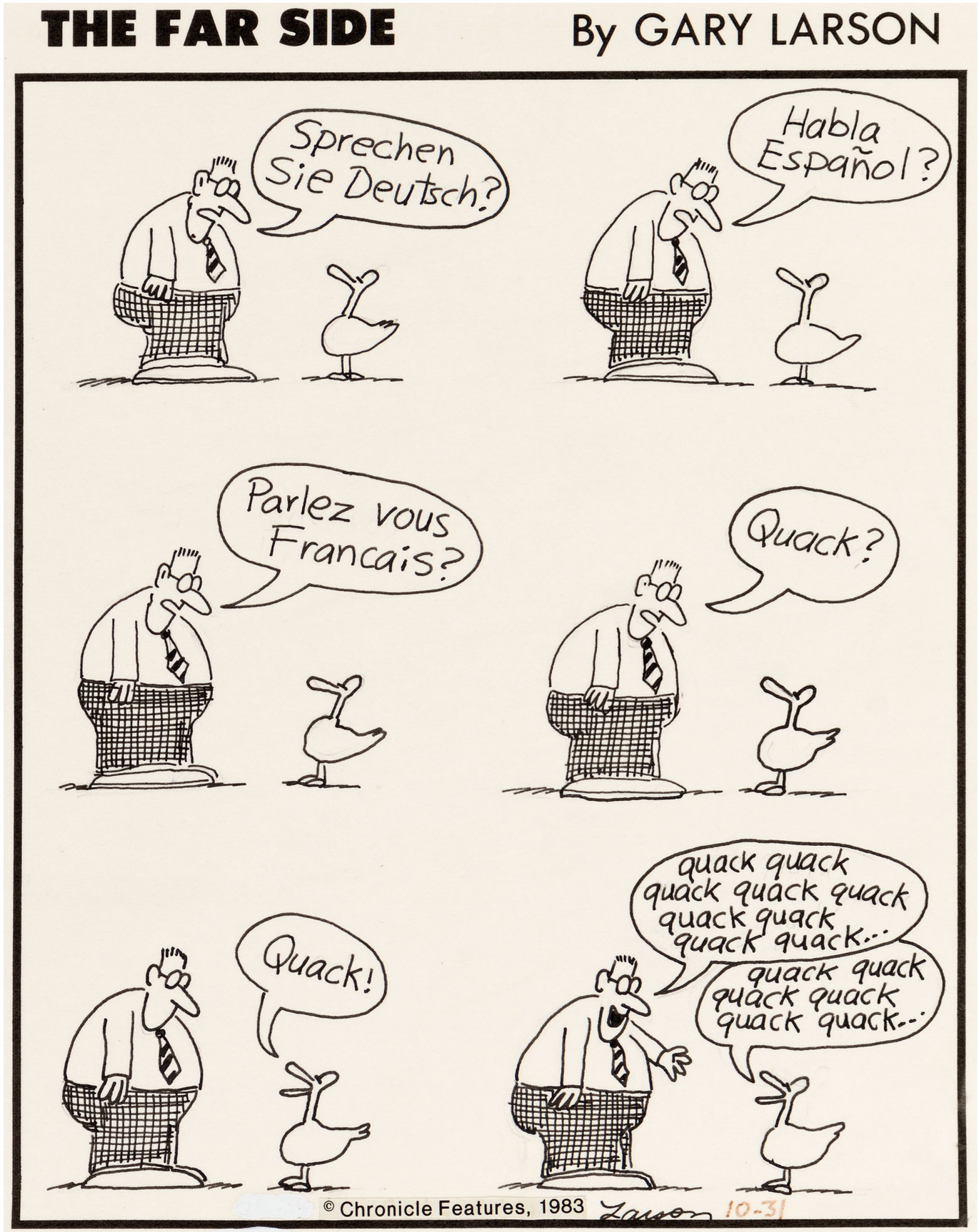
- For use cases we do in class, I am not too concerned about the arrows or the dashes – I want to see include for required tasks and extend for optional tasks

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 example of design pattern: Strategy

Ducks…

- Warning – this lecture, and other future lectures, contain quite a bit of duck



Imaged by Heritage Auctions, HA.com

Design Patterns are Everywhere

- In 1995, a book was published by the “Gang of Four” (Gamma, Helm, Johnson, Vlissides) 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.
- Since that time lots of Design Pattern 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 more effective in OO A&D

Cultural Anthropology

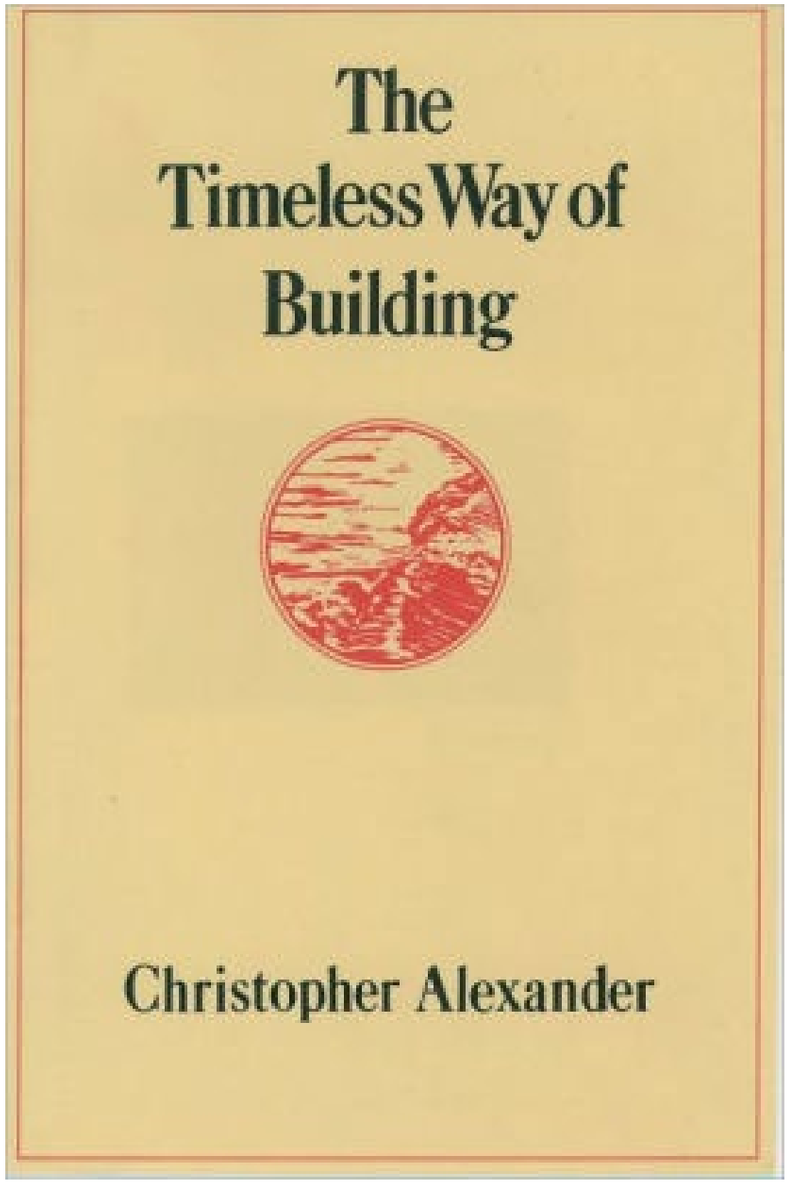
- Design Patterns have their intellectual roots in the discipline of cultural anthropology
  - Within a culture, individuals will agree on what is considered good design
    - “Cultures make judgements on good design that transcend individual beliefs”
  - Patterns (structures and relationships that appear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

As an aside, the folks responsible for user experience analysis and design at a company called Menlo Innovations are called High-Tech Anthropologists: They perform user experience assessments and interviews, but they also spend significant time observing the users in their own environments (natural habitats)

https://menloinnovations.com/our-way/our-process

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
- 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 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

Reminder

- We’re starting in on the material in the Head First Design Patterns book now
- Today’s lecture is taken in part from Chapter 1: Intro to Design Patterns
- The next pattern lecture on the Observer pattern will come from material in Chapter 2… etc.

Design Pattern Relationships

- The diagram is the 23 pattern set from the Gang of Four book
- Our textbook looks in detail at:
  - Strategy, Observer, Decorator, Simple Factory/ Factory/ Abstract Factory, Singleton, Command, Adapter, Façade, Template, Iterator, Composite, State, Proxy, MVC
- It quickly reviews:
  - Bridge, Builder, Chain of Responsibility, Flyweight, Interpreter, Mediator, Memento, Prototype, Visitor, Composite

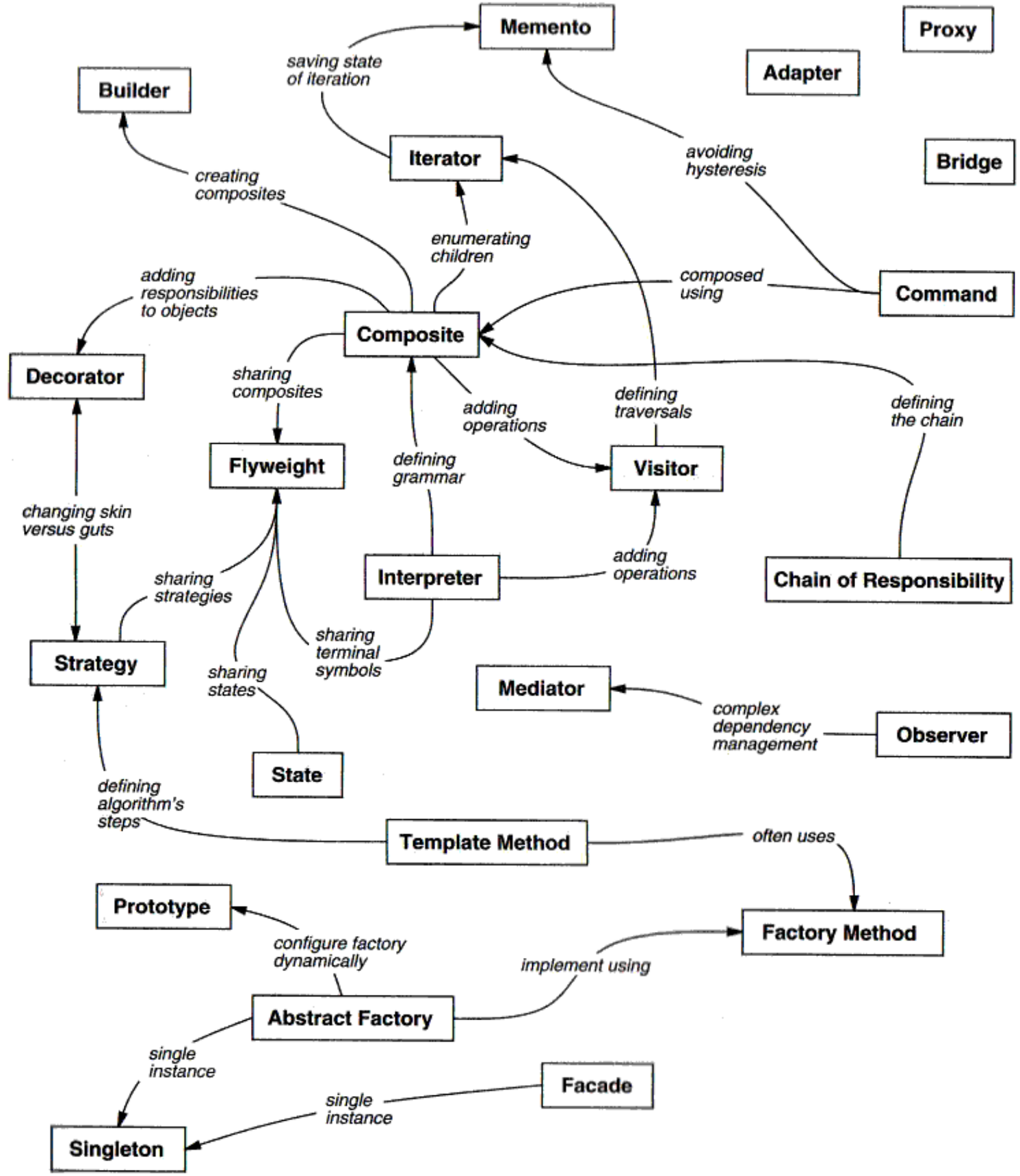


Figure 1.1: Design pattern relationships

Design Patterns and Software (II)

- Design patterns 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?

- 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
- 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

- In the previous textbook there is a good 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 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

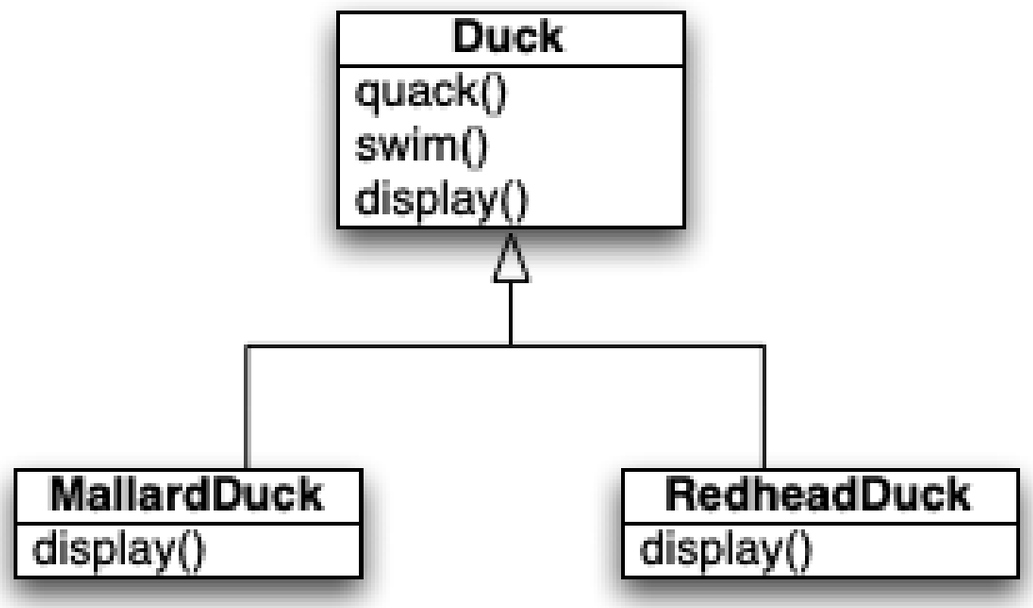
- “I have this one object with some important information and these other objects over here need to know when its information changes. These other objects come and go. I’m thinking I should separate out the notification and client registration functionality from the functionality of the object and just 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

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

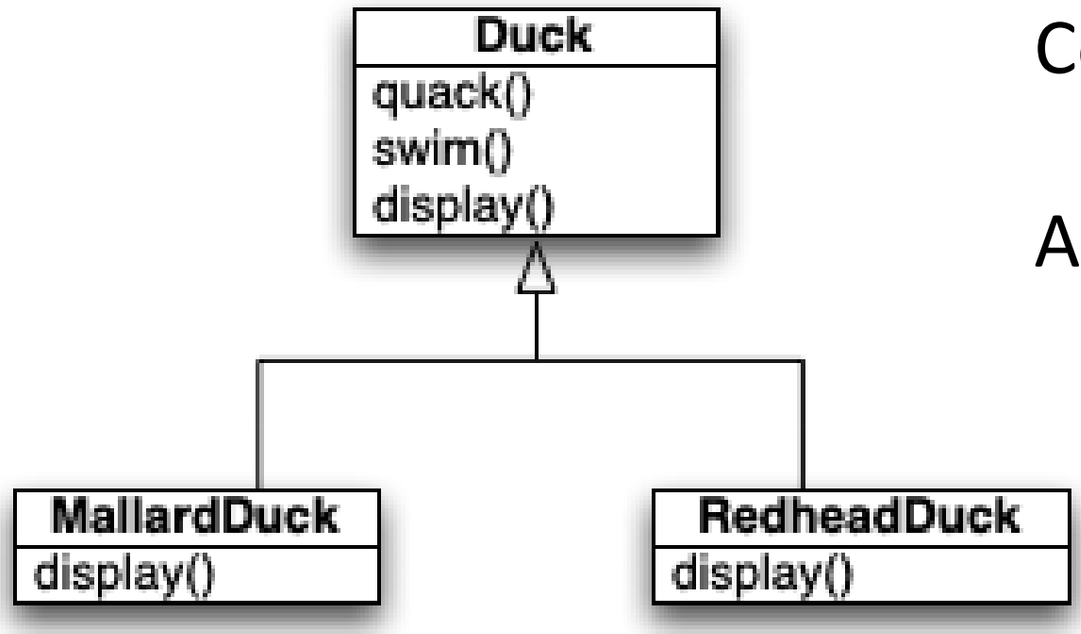
Design Pattern by Example

- The textbook uses the following Duck example
- SimUDuck: a “duck pond simulator” that can show a wide variety of duck species swimming and quacking
  - Initial State



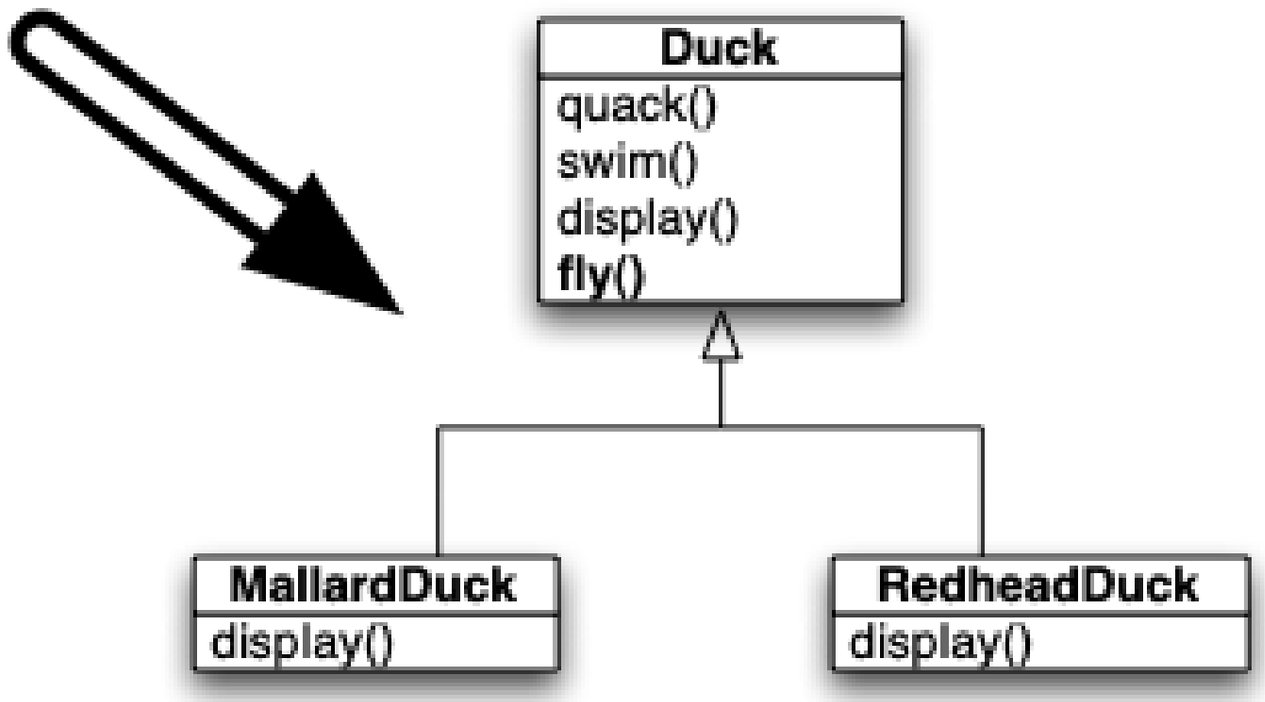
- But a request has arrived to allow ducks to also fly.

Easy

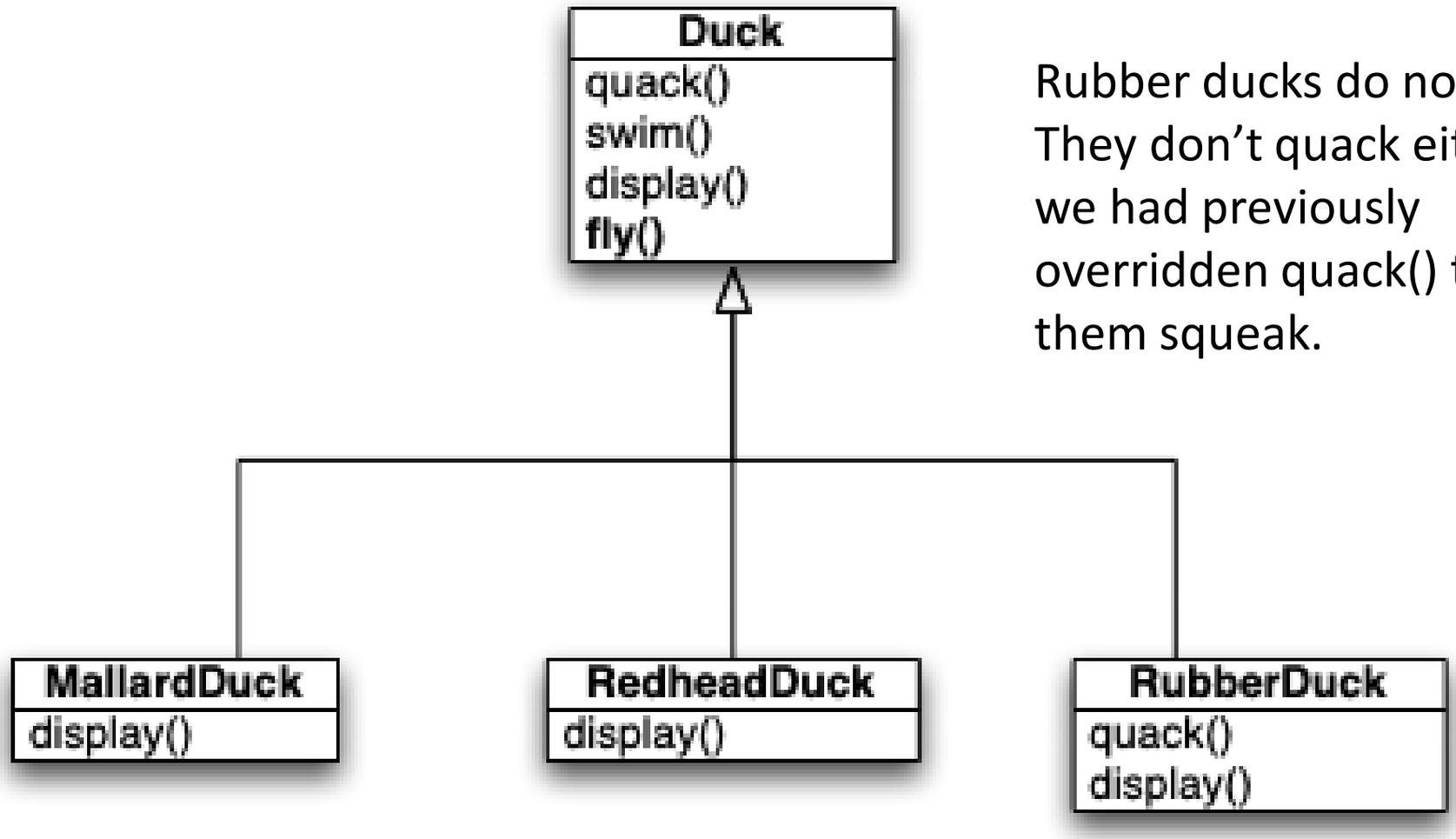


Code Reuse via Inheritance

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



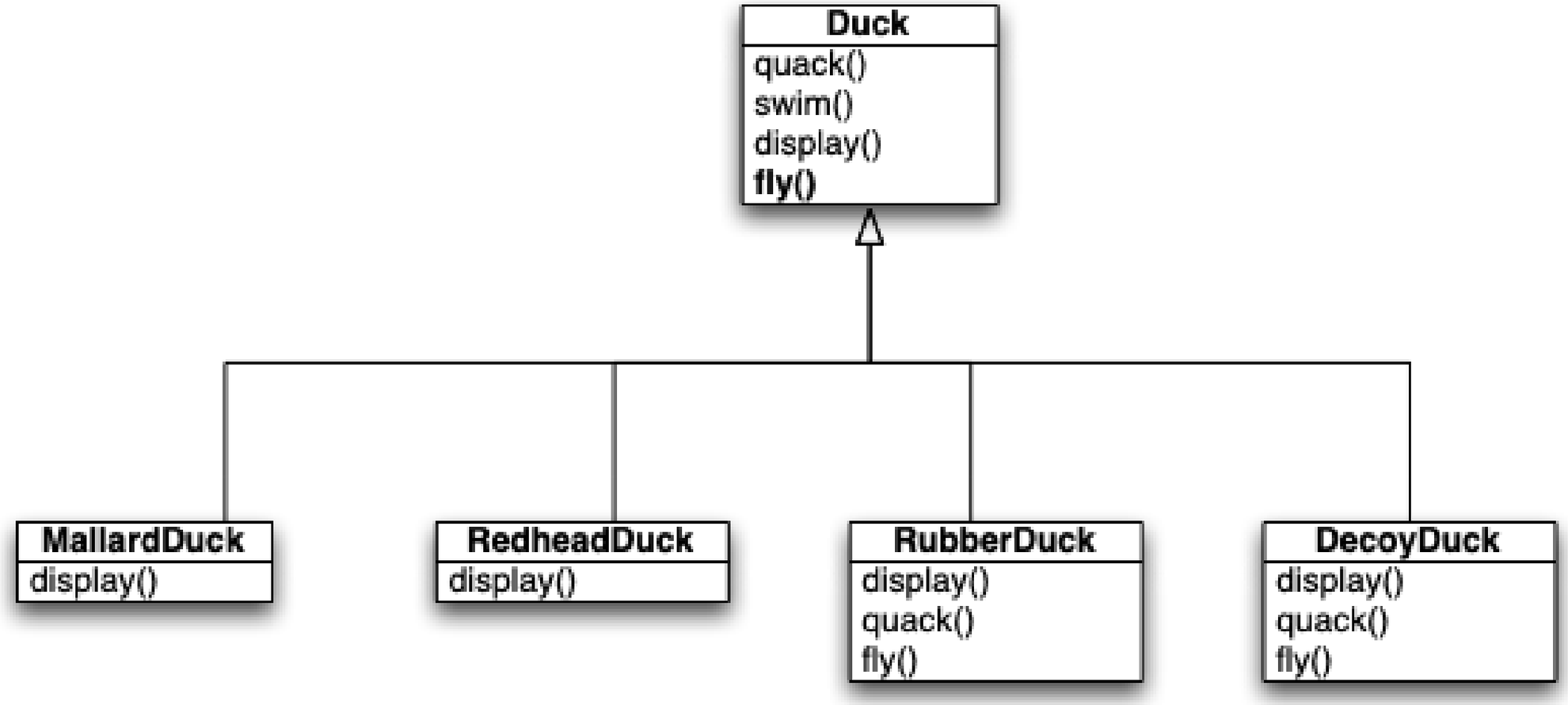
Whoops!



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...

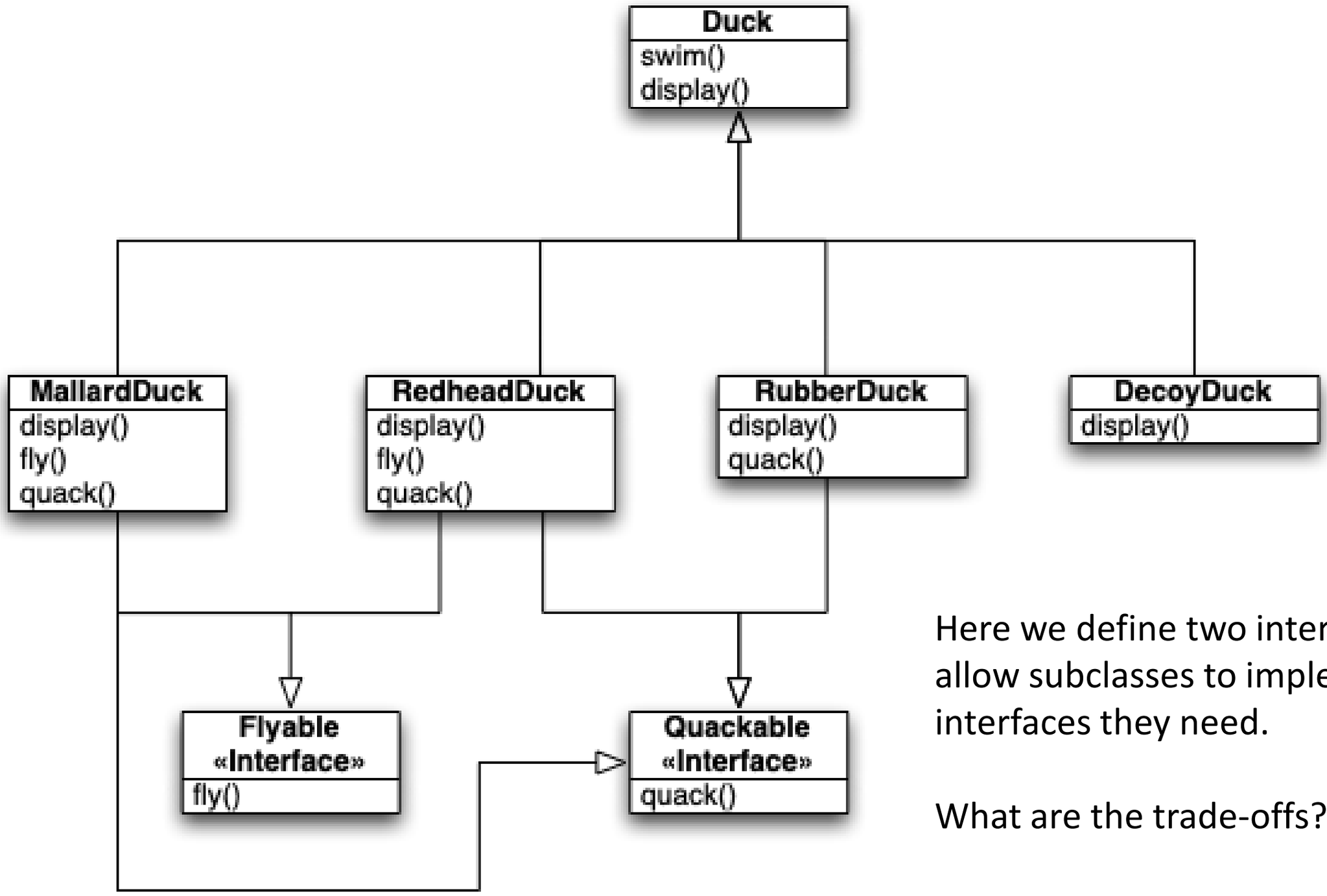
Double Whoops!



…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

A glimpse ahead…

- Head First Design Patterns: OO Principles
  - Encapsulate what varies
  - Favor composition over inheritance
  - Program to interfaces not implementations
  - Strive for loosely coupled designs between objects that interact
  - Classes should be open for extension, but closed for modification
  - Depend on abstractions, not concrete classes
  - Only talk to your friends
  - Don’t call us, we’ll call you
  - A class should have only one reason to change

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 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

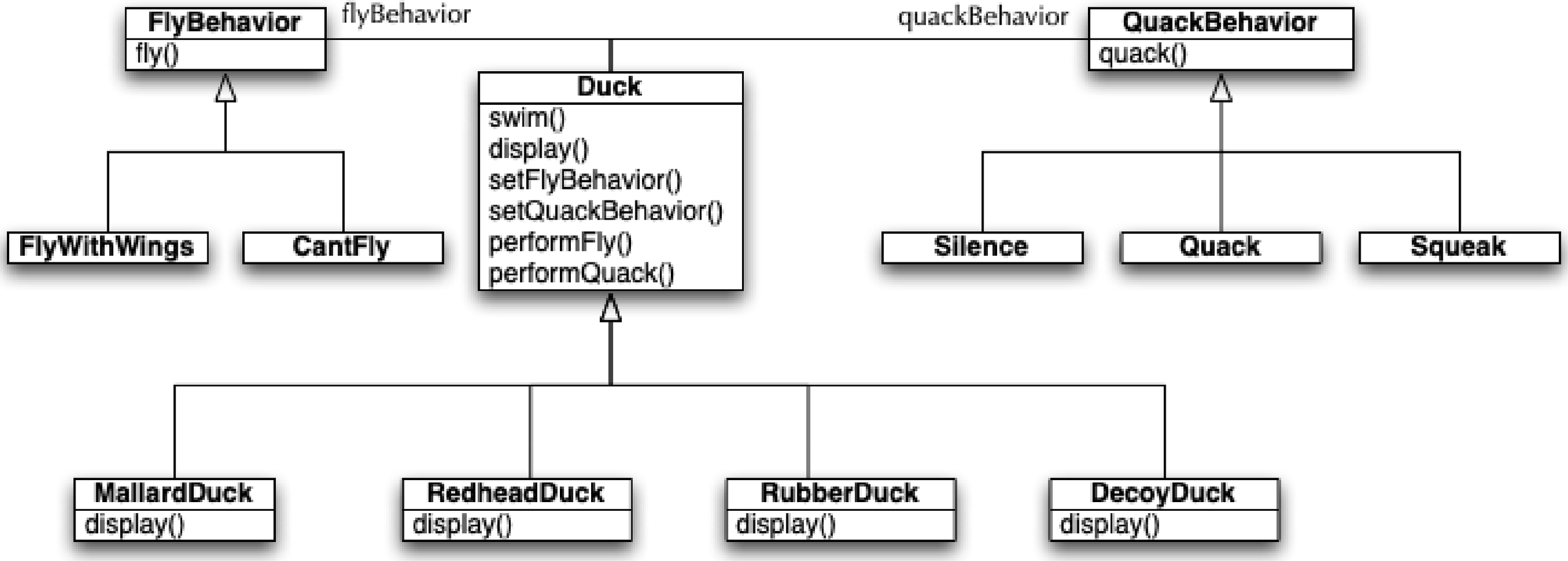
“Code to Interface” Does NOT Imply a Java Interface

- 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 delegation over inheritance"

New Class Diagram



FlyBehavior and QuackBehavior define a set of behaviors that provide behavior to Duck. Duck delegates to 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

```
1  public abstract class Duck {
2      FlyBehavior flyBehavior;
3      QuackBehavior quackBehavior;
4
5      public Duck() {
6      }
7
8      public void setFlyBehavior (FlyBehavior fb) {
9          flyBehavior = fb;
10     }
11
12     public void setQuackBehavior(QuackBehavior qb) {
13         quackBehavior = qb;
14     }
15
16     abstract void display();
17
18     public void performFly() {
19         flyBehavior.fly();
20     }
21
22     public void performQuack() {
23         quackBehavior.quack();
24     }
25
26     public void swim() {
27         System.out.println("All ducks float, even decoys!");
28     }
29 }
30
```

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

DuckSimulator.java (Part 1)

```
17    public static void main(String[] args) {
18
19      List<Duck> ducks  = new LinkedList<Duck>();
20
21      Duck model = new ModelDuck();
22
23      ducks.add(new DecoyDuck());
24      ducks.add(new MallardDuck());
25      ducks.add(new RedHeadDuck());
26      ducks.add(new RubberDuck());
27      ducks.add(model);
28
29      processDucks(ducks);
30
31      // change the Model Duck's behavior dynamically
32      model.setFlyBehavior(new FlyRocketPowered());
33      model.setQuackBehavior(new Squeak());
34
35      processDucks(ducks);
36    }
37  }
38
```

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)

```
1  import java.util.LinkedList;
2  import java.util.List;
3
4  public class DuckSimulator {
5
6    public static void processDucks(List<Duck> ducks) {
7      for (Duck d : ducks) {
8        System.out.println("----------------------------------");
9        System.out.println("Name: " + d.getClass().getName());
10       d.display();
11       d.performQuack();
12       d.performFly();
13       d.swim();
14     }
15   }
```

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

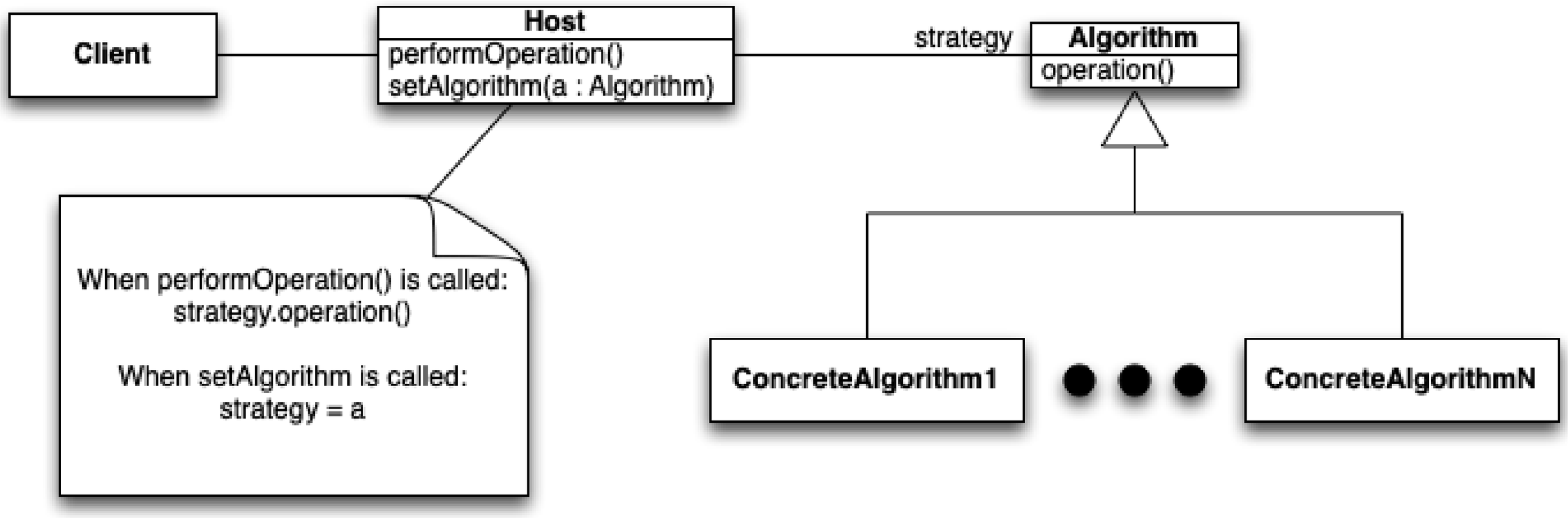
Not Completely Decoupled

- 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**
    - **Favor Delegation 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.
- We'll talk about Strategy a bit again later

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

Applying Strategy

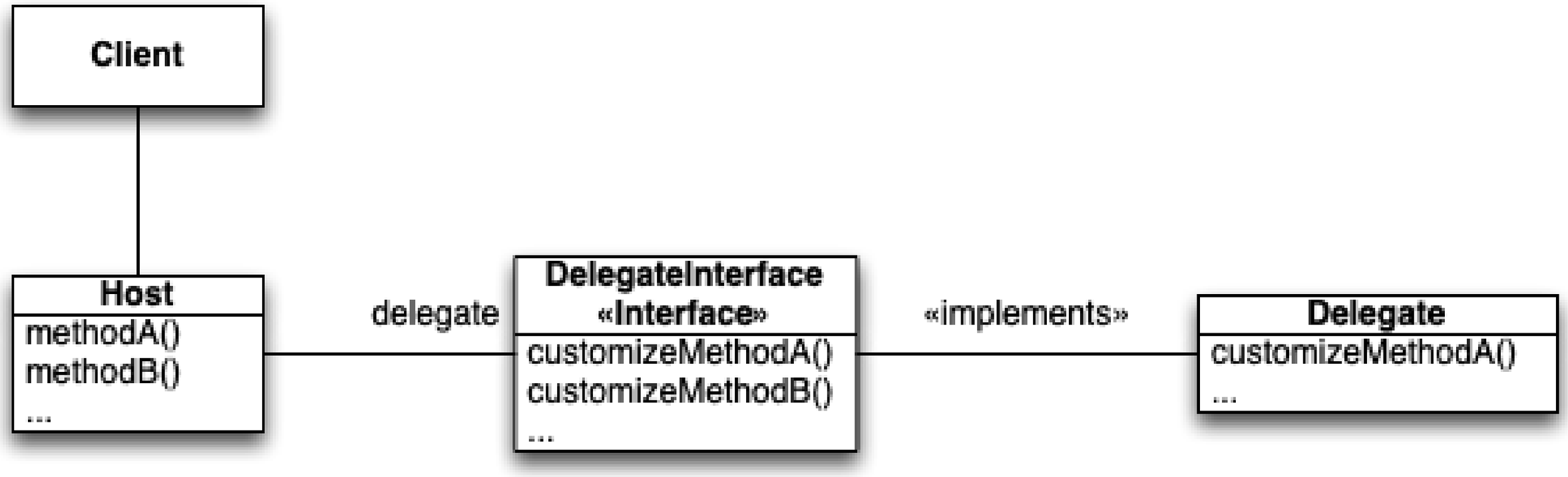
Use Strategy if…

- Many related classes differ only by behavior; Strategy allows configuring a single class instance with a selected behavior
- Varying an algorithm used in a class with an external and focused hierarchy of referenced algorithms
- Encapsulating data that clients shouldn't know about, but the algorithm uses; Strategy helps protect complex, algorithm-specific data structures
- Eliminate conditionals by composing selected Strategy algorithms with object instantiations
  - In many cases, a complex conditional statement should trigger a code smell – how else can I handle these alternatives
- Does increase the number of classes in the solution and the objects in use
- Great detailed Strategy (and all patterns) reference at:
  - http://www.cs.unc.edu/~stotts/GOF/hires/pat5ifso.htm

Review of Delegation

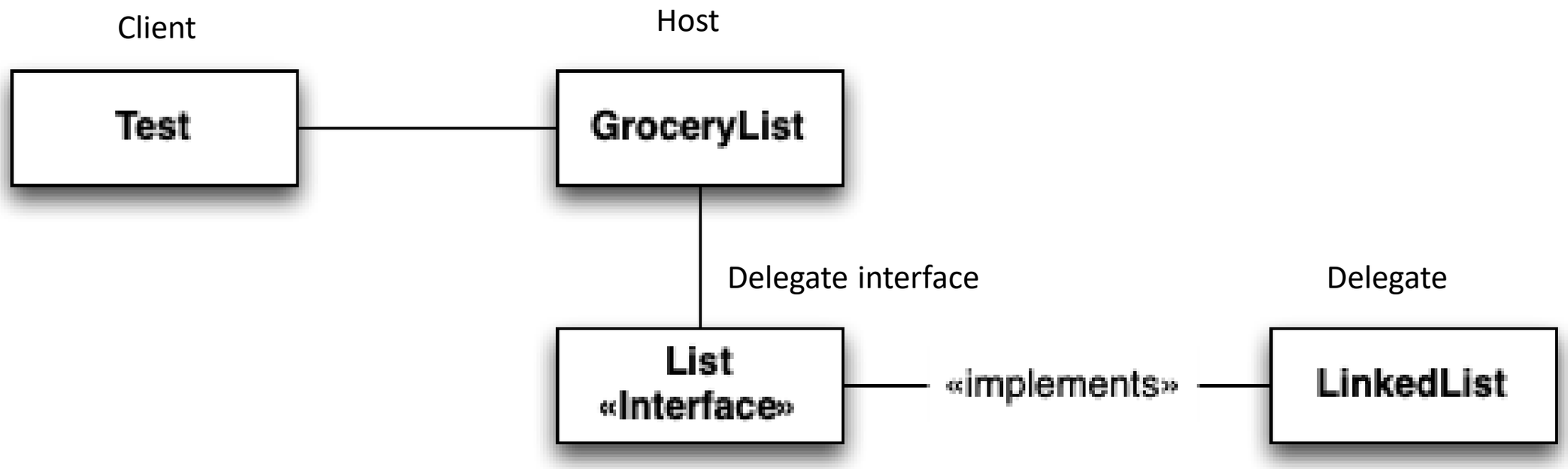
- Purpose of Delegate:
  - Allow an object's behavior to be customized without forcing a developer to create a subclass that overrides default behavior
- Structure
  - Host object; Delegate Interface; Delegate object; Client
  - Client invokes method on Host; Host checks to see if Delegate handles this method; if so, it routes the call to the Delegate; if not, it provides default behavior for the method

Structure of Delegate



Here, if Client invokes methodA() on Host, the Delegate's customizeMethodA() will be invoked at some point to help customize Host's behavior for methodA

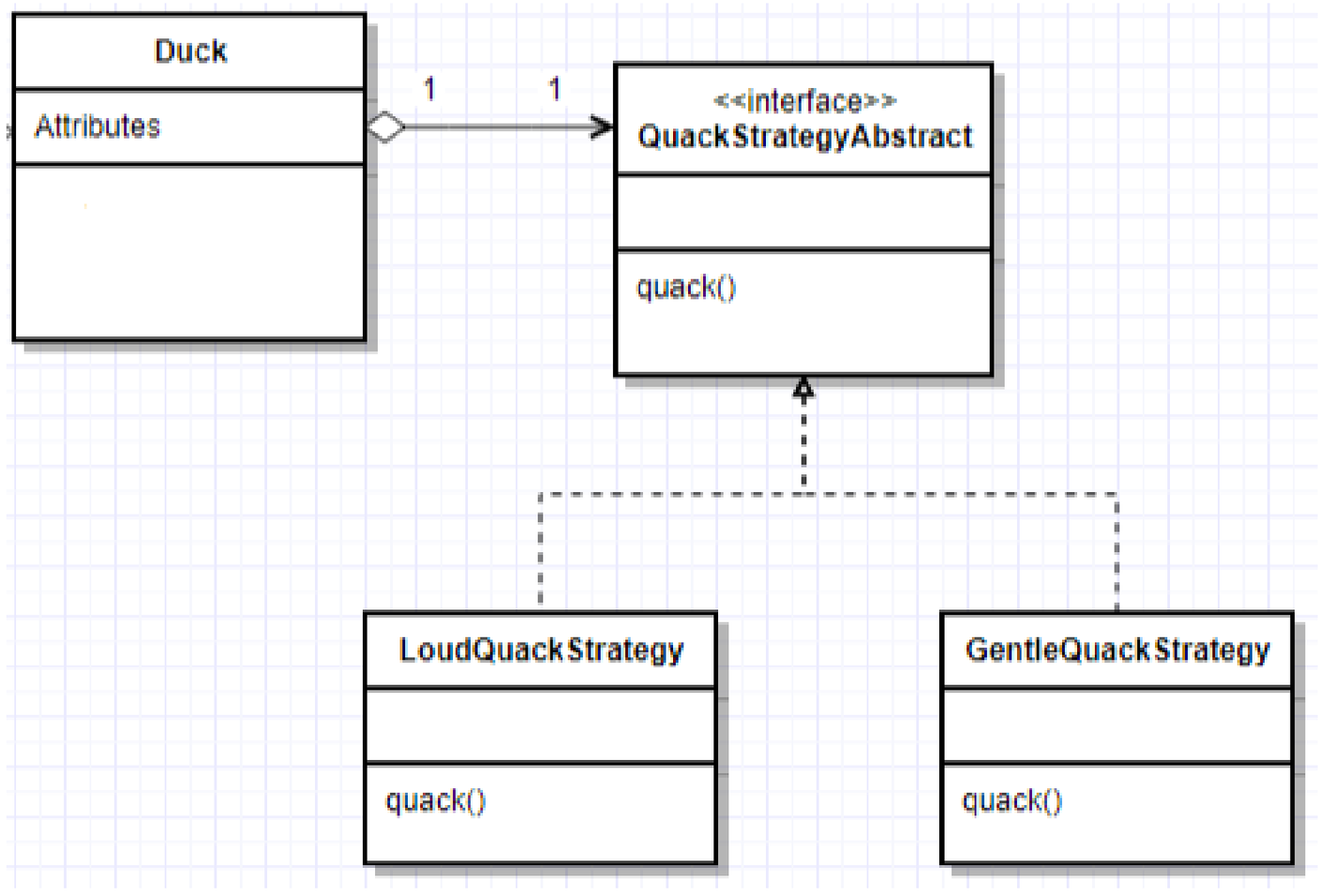
We've seen this before: GroceryList with Delegation



What Strategy looks like in Python

In this example of using Strategy, we're

- moving the quack behavior out of the Duck class
- Making an interface for the abstract strategy for the behavior (actually an abstract class)
- Making concrete strategies to reference in Duck instances for the quack method implementation



Example from: https://medium.com/@sheikhsajid/design-patterns-in-python-part-1-the-strategy-pattern-54b24897233e

Python - Strategy

This module, strategy.py, will hold our strategy implementation.

We have two concrete implementations of the quack strategy, loud and gentle.

Python doesn’t directly have interfaces, but it does have an abstract class library (abc).

```
import abc  # Python's built-in abstract class library

class QuackStrategyAbstract(object):
    """You do not need to know about metaclasses.
    Just know that this is how you define abstract
    classes in Python."""
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def quack(self):
        """Required Method"""

class LoudQuackStrategy(QuackStrategyAbstract):
    def quack(self):
        print "QUACK! QUACK!!"

class GentleQuackStrategy(QuackStrategyAbstract):
    def quack(self):
        print "quack!"
```

Python - Strategy

In our main duck.py code, we'll import the strategy definitions, assign the strategies to new classes of ducks, and then instantiated ducks can use the assigned strategies. We could assign the strategy at runtime as well.

```
from strategy import LoudQuackStrategy
from strategy import GentleQuackStrategy

loud_quack = LoudQuackStrategy()
gentle_quack = GentleQuackStrategy()

class Duck(object):
    def __init__(self, quack_strategy):
        self._quack_strategy = quack_strategy

    def quack(self):
        self._quack_strategy.quack()
```

```
# Types of Ducks
class ToyDuck(Duck):
    def __init__(self):
        super(ToyDuck, self).__init__(gentle_quack)

class RobotDuck(Duck):
    def __init__(self):
        super(RobotDuck, self).__init__(loud_quack)

robo = RobotDuck()
robo.quack()  # QUACK! QUACK!!
```

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
- Strategy Pattern
  - For families of related algorithms
  - Alternative to inheritance and subclassing
  - Can eliminate conditionals
  - Can increase number of objects