Patterns of Patterns

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 25 — 11/17/2009

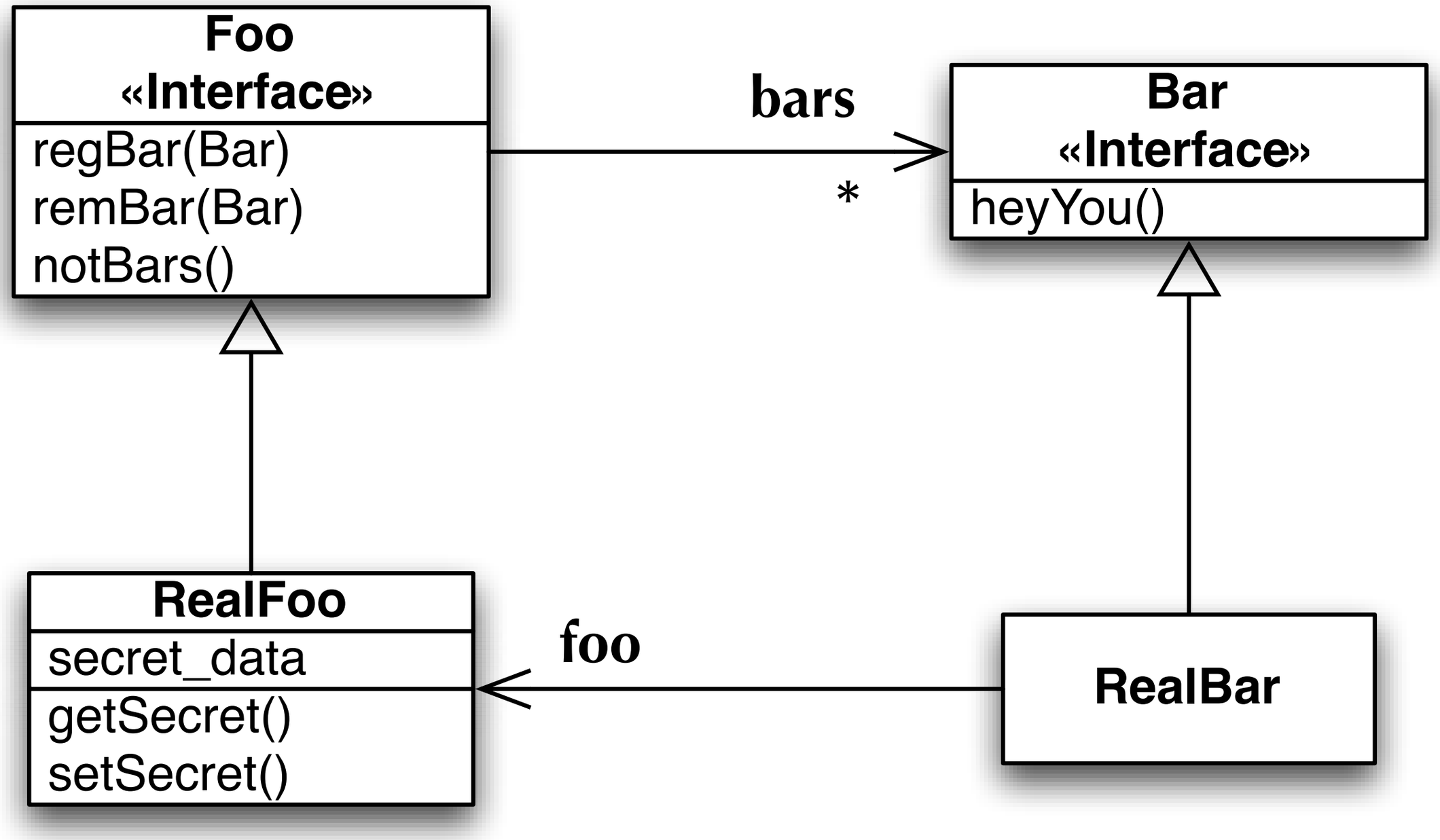
Lecture Goals

- Cover Material from Chapter 12 of the Design Patterns Textbook

Patterns of Patterns

- Patterns can be
    - used together in a single system (we've seen this several times)
    - can be combined to create, in essence, a new pattern
- Chapter 12 does a good job of showing both of these situations in use
    - DuckSimulator Revisited: An example that uses six patterns at once
    - Model View Controller: A pattern that makes use of multiple patterns
- We'll see many examples as we move through this lecture

But first… what pattern is this?



Remember that the names of classes participating in a pattern is unimportant; Its the structure (of the relationships and methods) that's important!

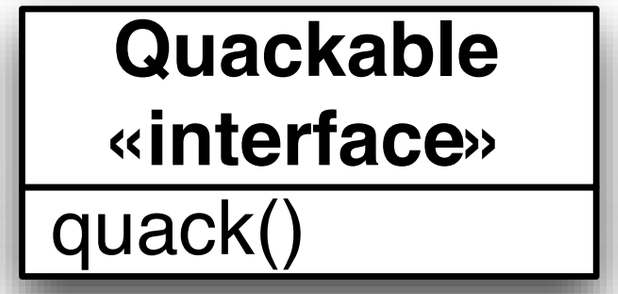
Duck Simulator Revisited

- We've been asked to build a new Duck Simulator by a Park Ranger interested in tracking various types of water fowl, ducks in particular.
- New Requirements
    - Ducks are the focus, but other water fowl (e.g. Geese) can join in too
    - Need to keep track of how many times duck's quack
    - Control duck creation such that all other requirements are met
    - Allow ducks to band together into flocks and subflocks
    - Generate a notification when a duck quacks
- Note: to avoid coding to an implementation, replace all instances of the word "duck" above with the word "Quackable"

Opportunities for Patterns

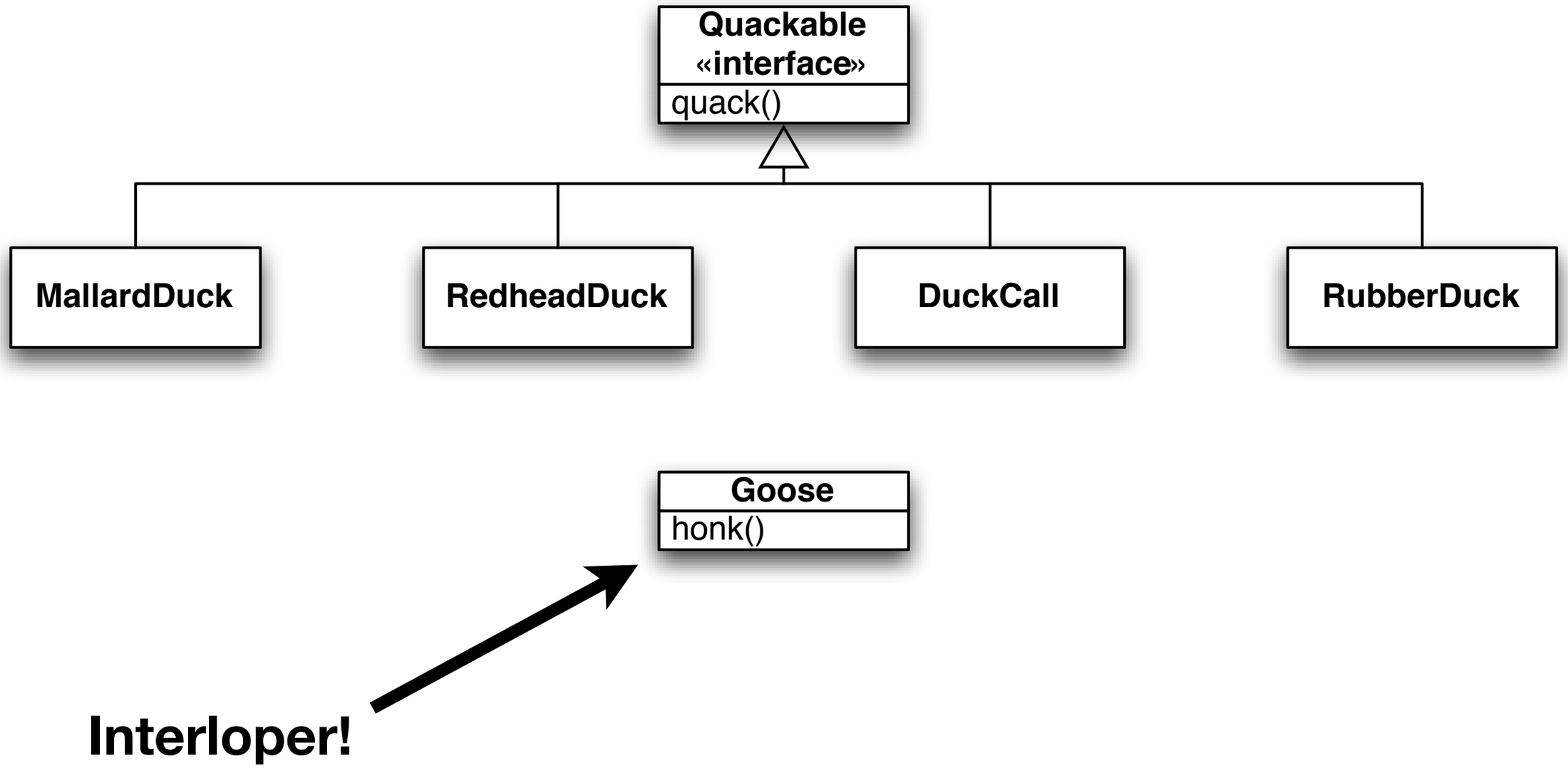
- There are several opportunities for adding patterns to this program
- New Requirements
  - Ducks are the focus, but other water fowl (e.g. Geese) can join in too **(ADAPTER)**
  - Need to keep track of how many times duck's quack **(DECORATOR)**
  - Control duck creation such that all other requirements are met **(FACTORY)**
  - Allow ducks to band together into flocks and subflocks **(COMPOSITE and ITERATOR)**
  - Generate a notification when a duck quacks **(OBSERVER)**
- Lets take a look at this example via a class diagram perspective

Step 1: Need an Interface

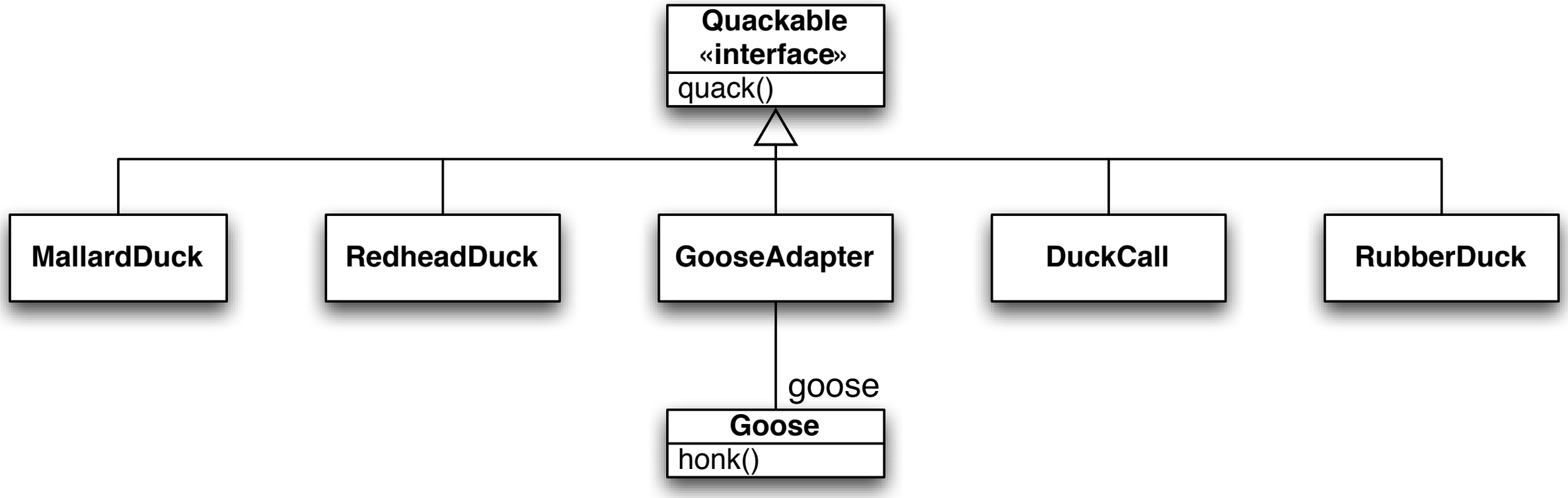


All simulator participants will implement this interface

Step 2: Need Participants

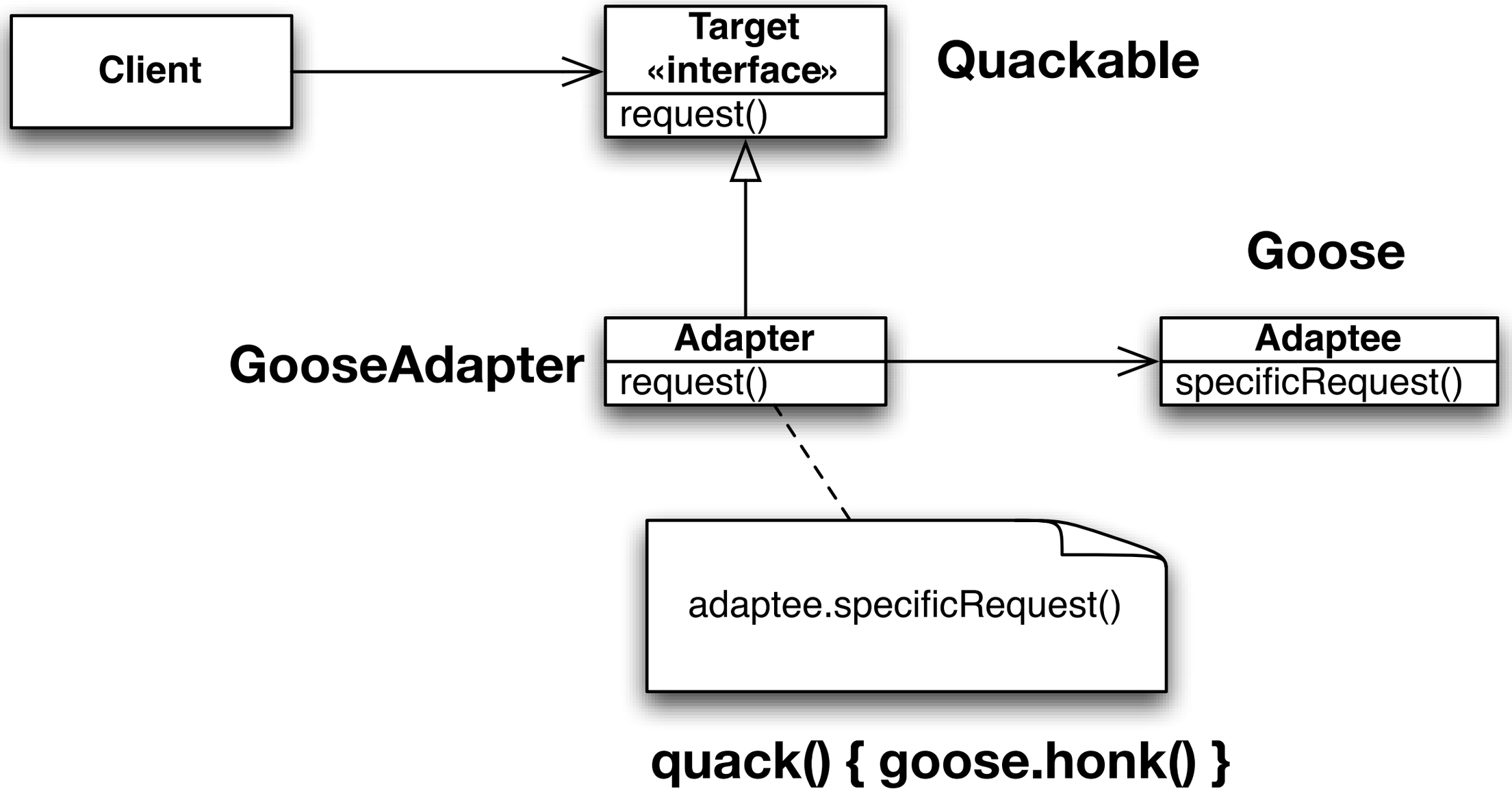


Step 3: Need Adapter

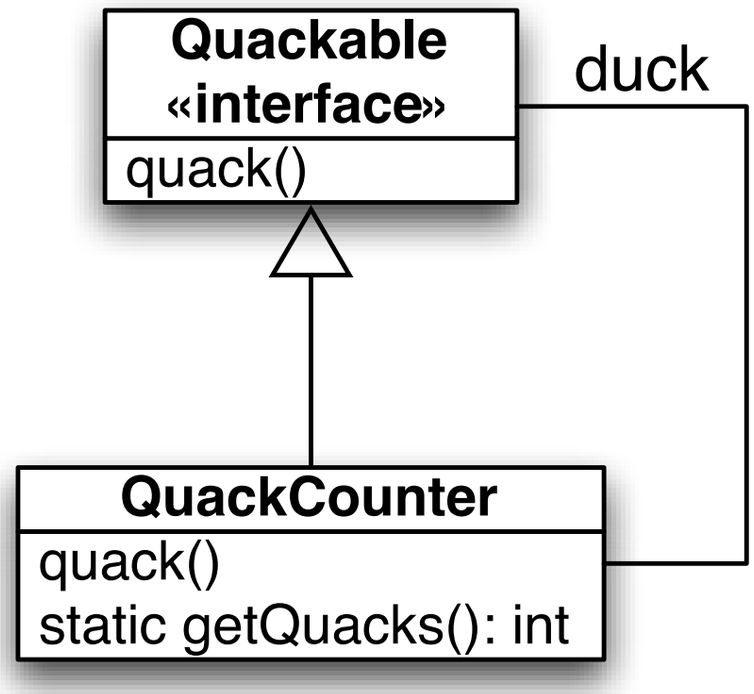


All participants are now Quackables,
allowing us to treat them uniformly

Review: (Object) Adapter Structure



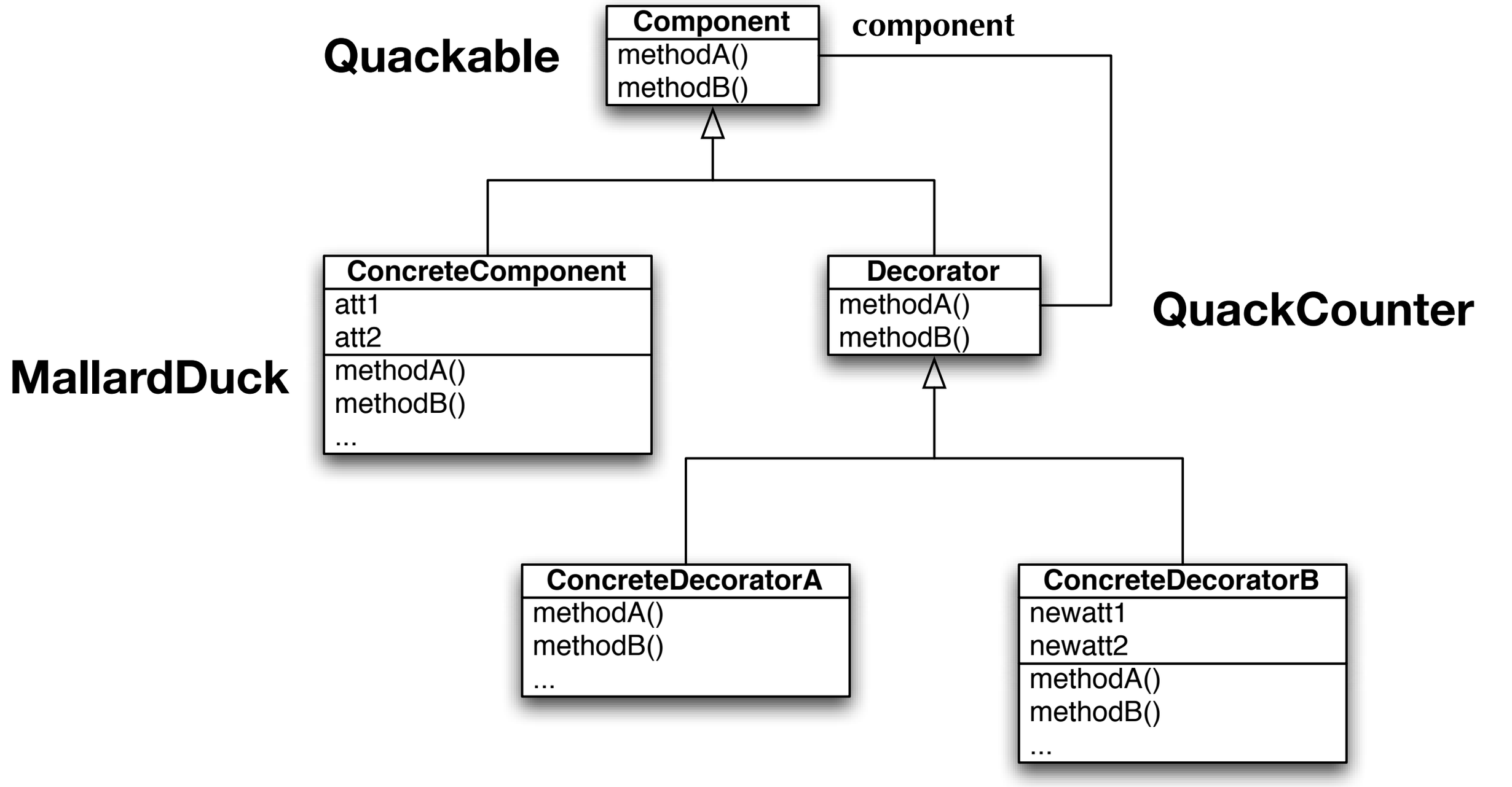
Step 4: Use Decorator to Add Quack Counting



Note: two relationships between QuackCounter and Quackable
What do they mean?

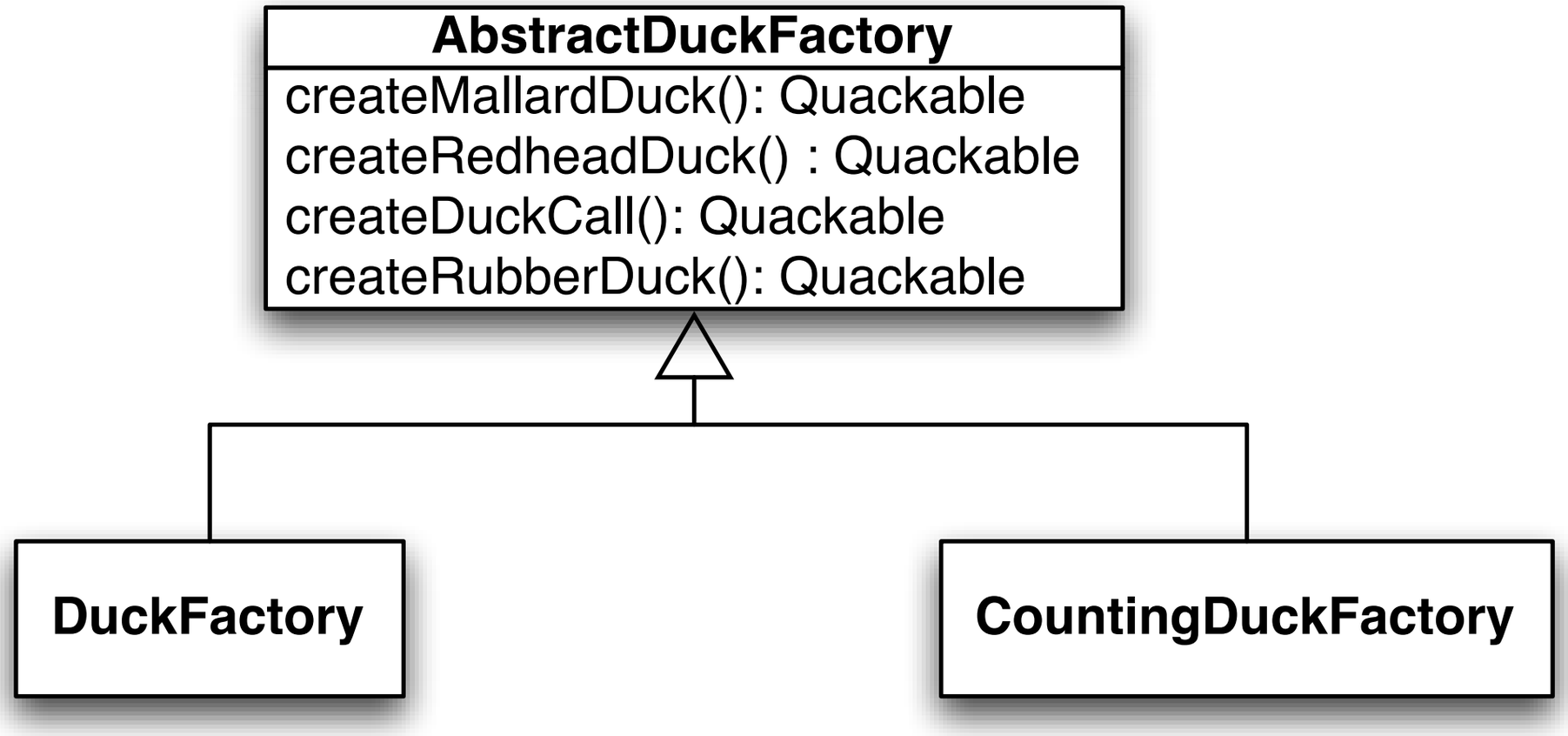
Previous classes/relationships are all still there… just elided for clarity

Review: Decorator Structure



No need for abstract Decorator interface in this situation; note that QuackCounter follows ConcreteDecorators, as it adds state and methods on top of the original interface.

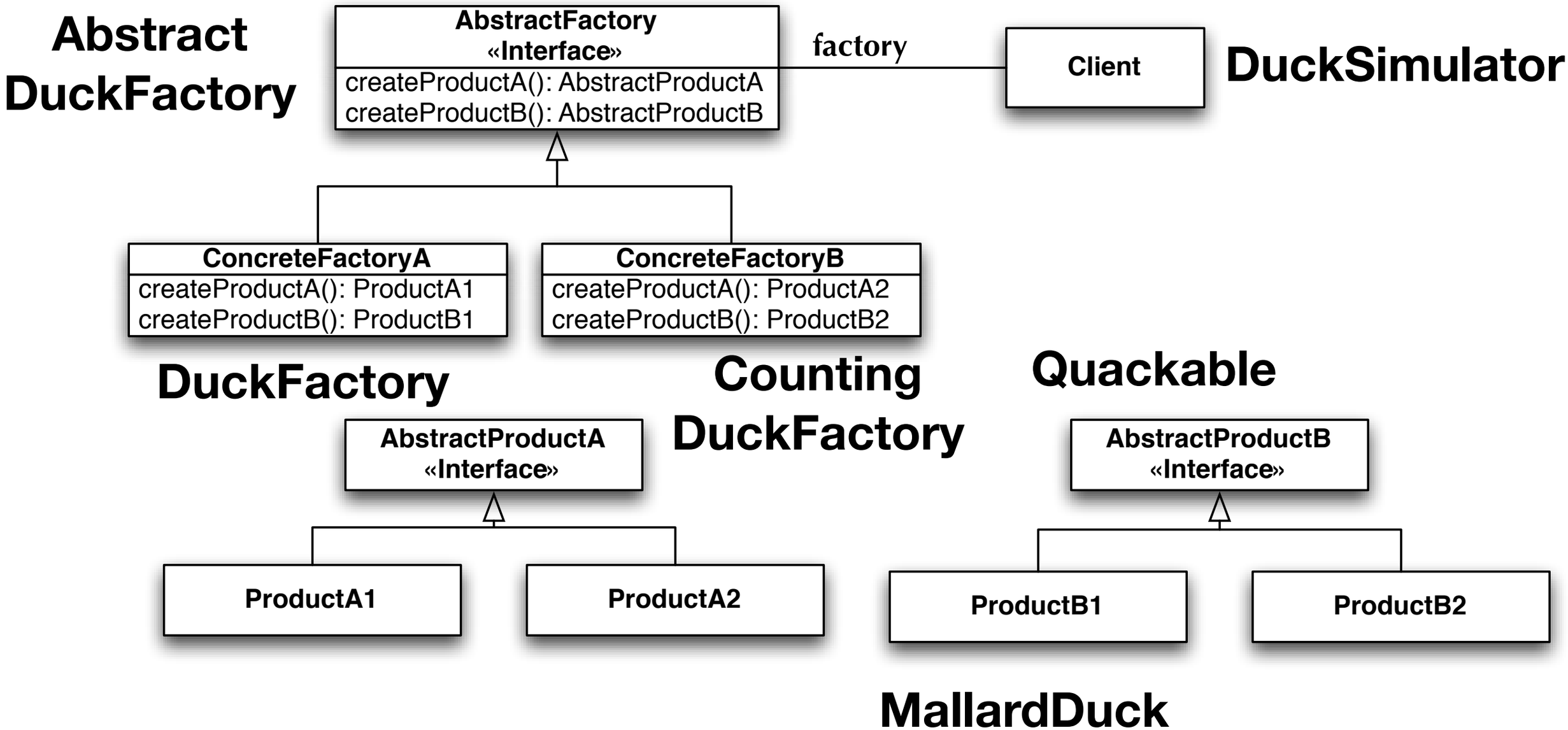
Step 5: Add Factory to Control Duck Creation



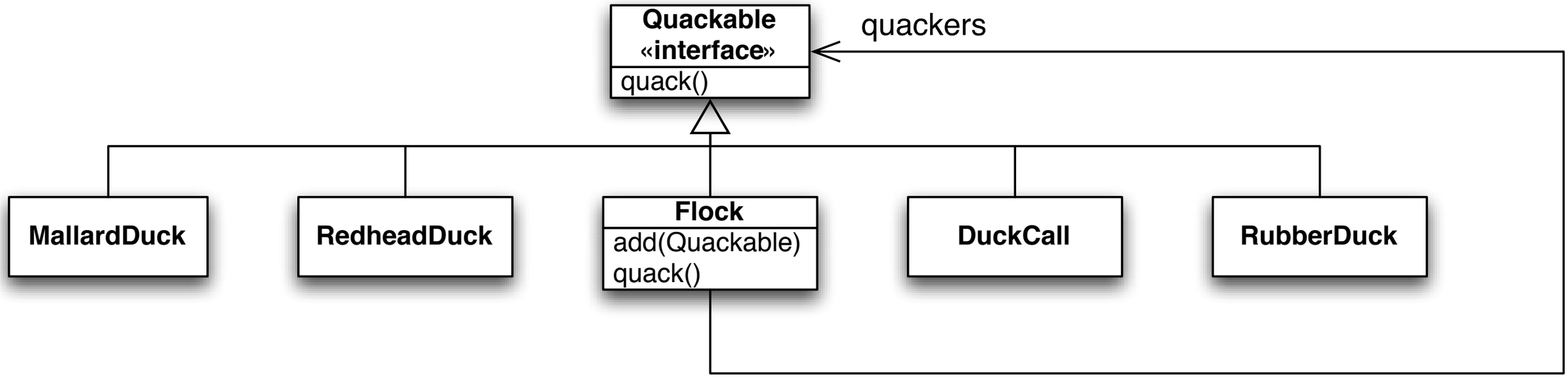
CountingDuckFactory returns ducks that are automatically wrapped by the QuackCounter developed in Step 4

This code is used by a method in DuckSimulator (not previously shown) that accepts an instance of AbstractDuckFactory as a parameter. **Demonstration**.

Review: Abstract Factory Structure



Step 6: Add support for Flocks with Composite



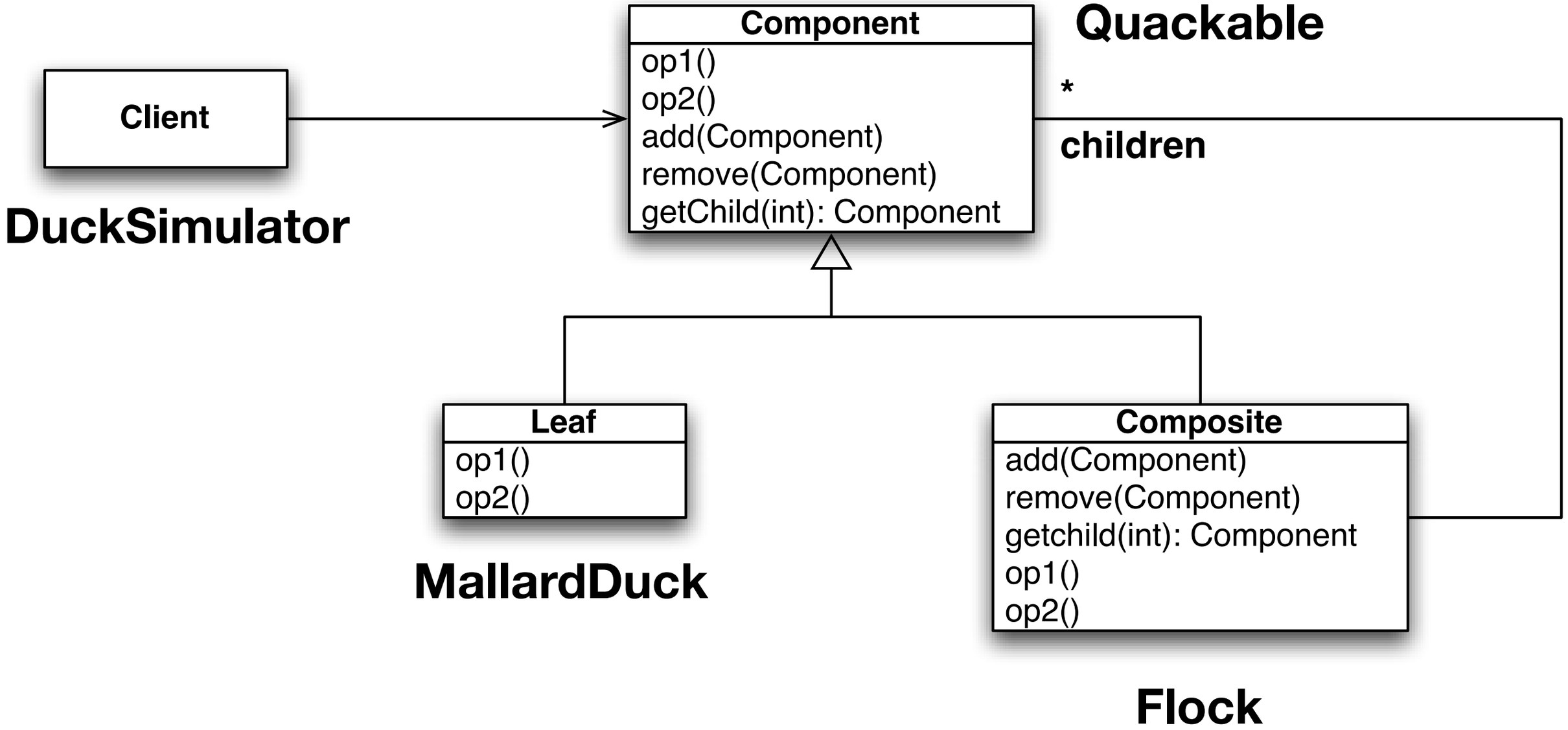
Note: Iterator pattern is hiding inside of Flock.quack(); **Demonstration**

Note: This is a variation on Composite, in which the Leaf and Composite classes have different interfaces;

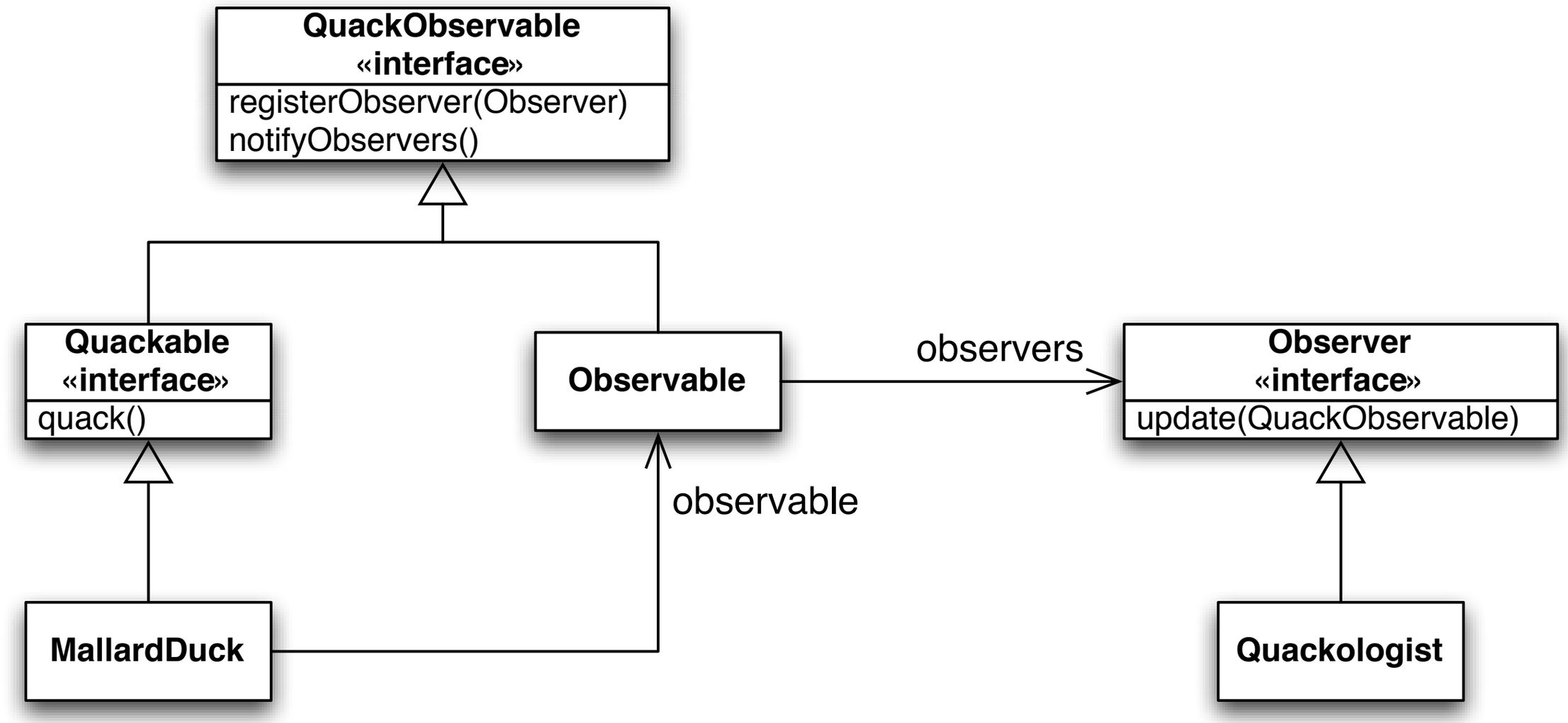
Only Flock has the “add(Quackable)” method.

Client code has to distinguish between Flocks and Quackables as a result. Resulting code is “safer” but less transparent.

Review: Composite Structure

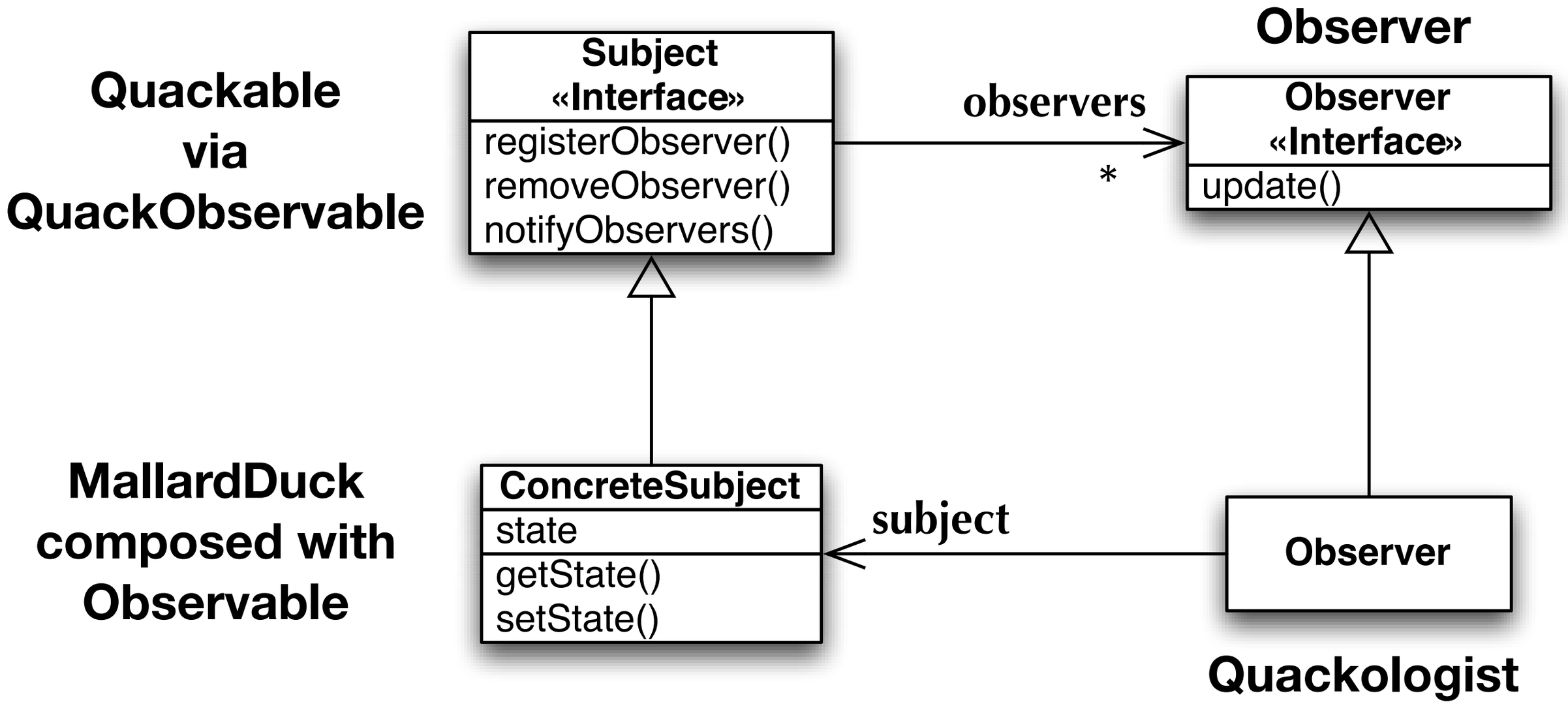


Step 7: Add Quack Notification via Observer



Cool implementation of the Observer pattern. All Quackables are made Subjects by having Quackable inherit from QuackObserver. To avoid duplication of code, an Observable helper class is implemented and composed with each ConcreteQuackable class. Flock does not make use of the Observable helper class directly; instead it delegates those calls down to its leaf nodes. **Demonstration**.

Review: Observer Structure



Counting Roles

- As you can see, a single class will play multiple roles in a design
  - Quackable defines the shared interface for five of the patterns
  - Each Quackable implementation has four roles to play: Leaf, ConcreteSubject, ConcreteComponent, ConcreteProduct
- You should now see why names do not matter in patterns
  - Imagine giving MallardDuck the following name:
    - MallardDuckLeafConcreteSubjectComponentProduct
  - !!!
- Instead, its the structure of the relationships between classes and the behaviors implemented in their methods that make a pattern REAL
  - And when these patterns live in your code, they provide multiple extension points throughout your design. Need a new product, no problem. Need a new observer, no problem. Need a new dynamic behavior, no problem.

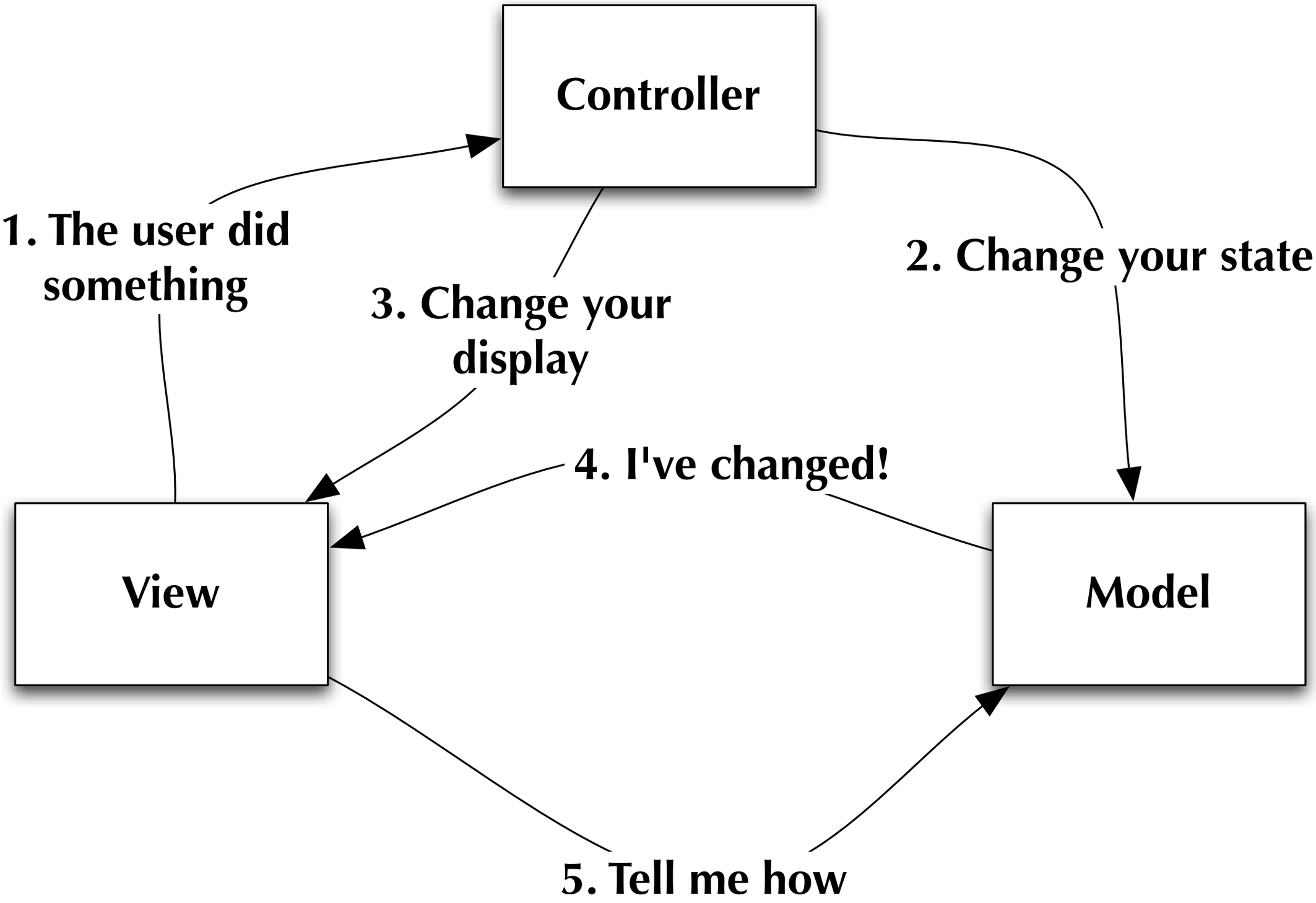
Model-View-Controller: A Pattern of Patterns

- Model-View-Controller (MVC) is a ubiquitous pattern that allows information (stored in models) to be viewed in a number of different ways (views), with each view aided by code that handles user input or notifies the view of updates to its associated models (controllers)
    - Speaking broadly
        - tools/frameworks for creating views are ubiquitous
            - the widgets of any GUI toolkit, templates in Web frameworks, etc.
        - data storage frameworks abound for handling models
            - generic data structures + persistence mechanisms (files, RDBMs, …)
        - controllers are almost ALWAYS written by hand
            - lone exception (that I know of) is Apple's Cocoa Bindings
                - ability to specify a binding between a value maintained by a widget and a similar value in your application's model

MVC Roles

- As mentioned, MVC is a pattern for manipulating information that may be displayed in more than one view
  - Model: data structure(s) being manipulated
    - may be capable of notifying observers of state changes
  - View: a visualization of the data structure
    - having more than one view is fine
    - MVC keeps all views in sync as the model changes
- Controller: handle user input on views
  - make changes to model as appropriate
  - more than one controller means more than one “interaction style” is available

MVC: Structure



MVC: Hidden Patterns

- Observer pattern used on models
  - Views keep track of changes on models via the observer pattern
    - A variation is to have controllers observe the models and notify views as appropriate
- View and Controller make use of the Strategy pattern
  - When an event occurs on the view, it delegates to its current controller
    - Want to switch from direct manipulation to audio only? Switch controllers
- Views (typically) implement the composite pattern
  - In GUI frameworks, tell the root level of a view to update and all of its sub-components (panels, buttons, scroll bars, etc.) update as well
- Others: Events are often handled via the Command pattern, views can be dynamically augmented with the decorator pattern, etc.

MVC Examples

- DJView
  - Example of one model, one controller, two views
  - Allows you to set a value called “beats per minute” and watch a progress bar “pulse” to that particular value
    - In book, referenced a bunch of midi-related code that did not work on my machine: was supposed to play a “beat track” that matched the specified tempo
      - I ripped that code out and substituted a thread that emits “beats” at the specified rate
- Heart Controller
  - Shows how previous behavior can be altered by changing the model and controller classes… now progress bar “pulse” mimics a human heart

Wrapping Up

- We've shown two ways in which "patterns of patterns" can appear in code
  - The first is when you use multiple patterns in a single design
    - Each individual pattern focuses on one thing, but the combined effect is to provide you with a design that has multiple extension points
  - The second is when two or more patterns are combined into a solution that solves a recurring or general problem
    - MVC is such a pattern (also known as a Compound pattern) that makes use of Observer, Strategy, and Composite to provide a generic solution to the problem of visualizing and interacting with the information stored in an application's model classes

Ken’s Corner (I)

- Go
- New programming language by Google
- Available at <http://golang.org/>
- Aims to be
    - Simple, fast, safe, concurrent, and fun!
- Is also open source!
    - (comes with the code for at least three compilers!)

Ken's Corner (II)

- Intriguing language because (among other things) it drops inheritance and instead enables interesting software design via a combination of types, interfaces and methods
- From package "io"

```
type Reader interface {
    Read(p []byte) (n int, err os.Error);
}
```

- This defines a new type which is an interface that has one method: read()
- From package "os"

```
type File struct { // private fields }

func (file *File) Read(b []byte) (n int, err Error)
```

- Because the function Read has been implemented for File, File is automatically a Reader… no explicit declaration has to be made!

Ken’s Corner (III)

- Other interesting features
  - syntax for declaring types is “backwards”
    - var s string = “”
    - type T struct { a, b int }
      - b []byte
- Compiler can infer types, so you can take the declaration of s and do:
  - s := “”
- That’s enough for Go to figure out that s is a variable of type string.
- Concurrency using goroutines and channels (example)

Coming Up Next

- Lecture 26: Refactoring, Part 1
- Lecture 27: Refactoring, Part 2