MVC and Compound Patterns

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 23

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

Example in Head First

- The MVC pattern is covered in Chapter 12 in Head First Design Patterns…
- Interestingly, MVC is not a pattern recognized as such in the Gang of Four Design Patterns book
    - They discuss it briefly as a framework in Smalltalk that combines other patterns

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 (a Compound pattern)
- Two examples
  - DuckSimulator Revisited: An example that uses six patterns at once
    - Comes from Head First Design Patterns
  - Model View Controller: A pattern that makes use of multiple patterns

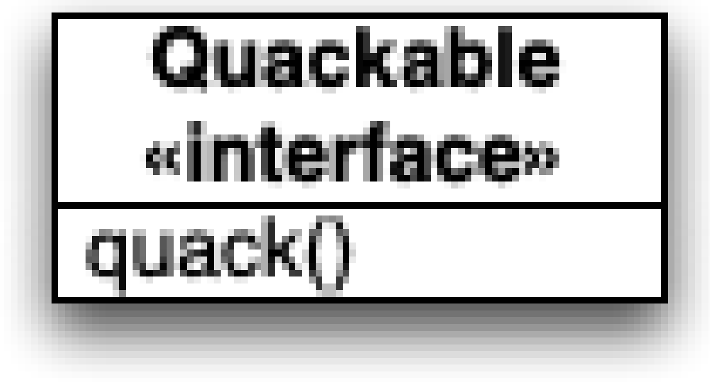
Duck Simulator Revisited

- 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

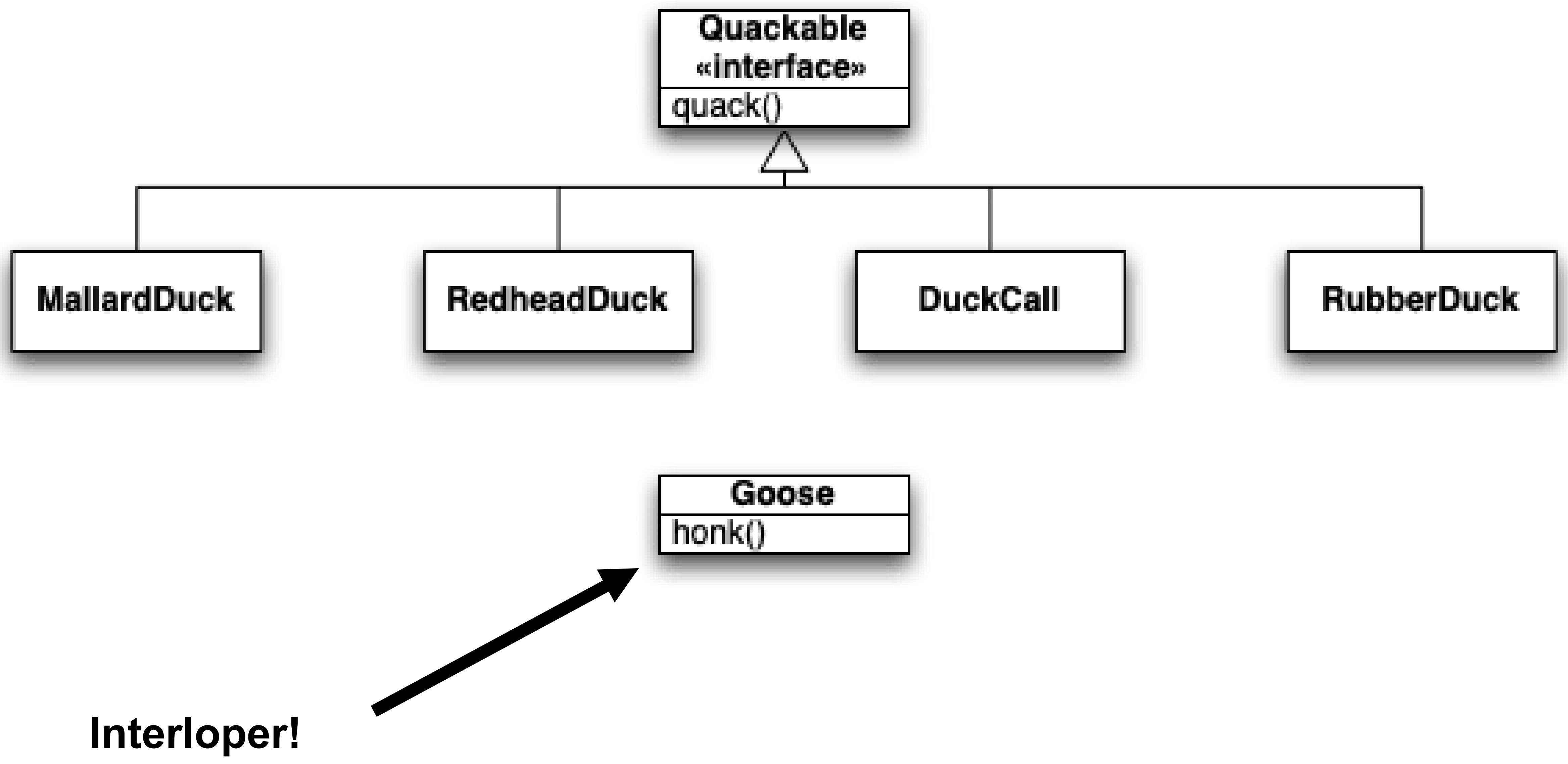
- There are several opportunities for adding patterns to this program
- New Requirements
  - Ducks are the focus, but other water fowl (e.g. Geese) can join in too **(ADAPTER)**
  - Need to keep track of how many times duck's quack **(DECORATOR)**
  - Control duck creation such that all other requirements 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

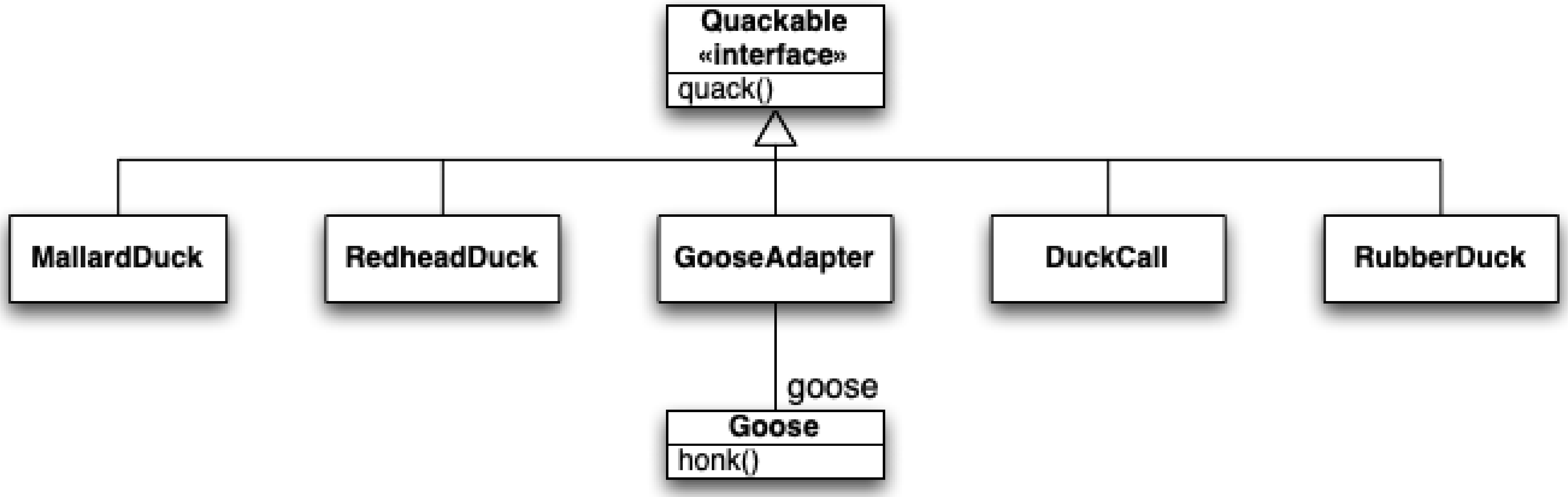


- Brand new version of the duck simulator
- All simulator participants will implement this interface

Step 2: Need Participants

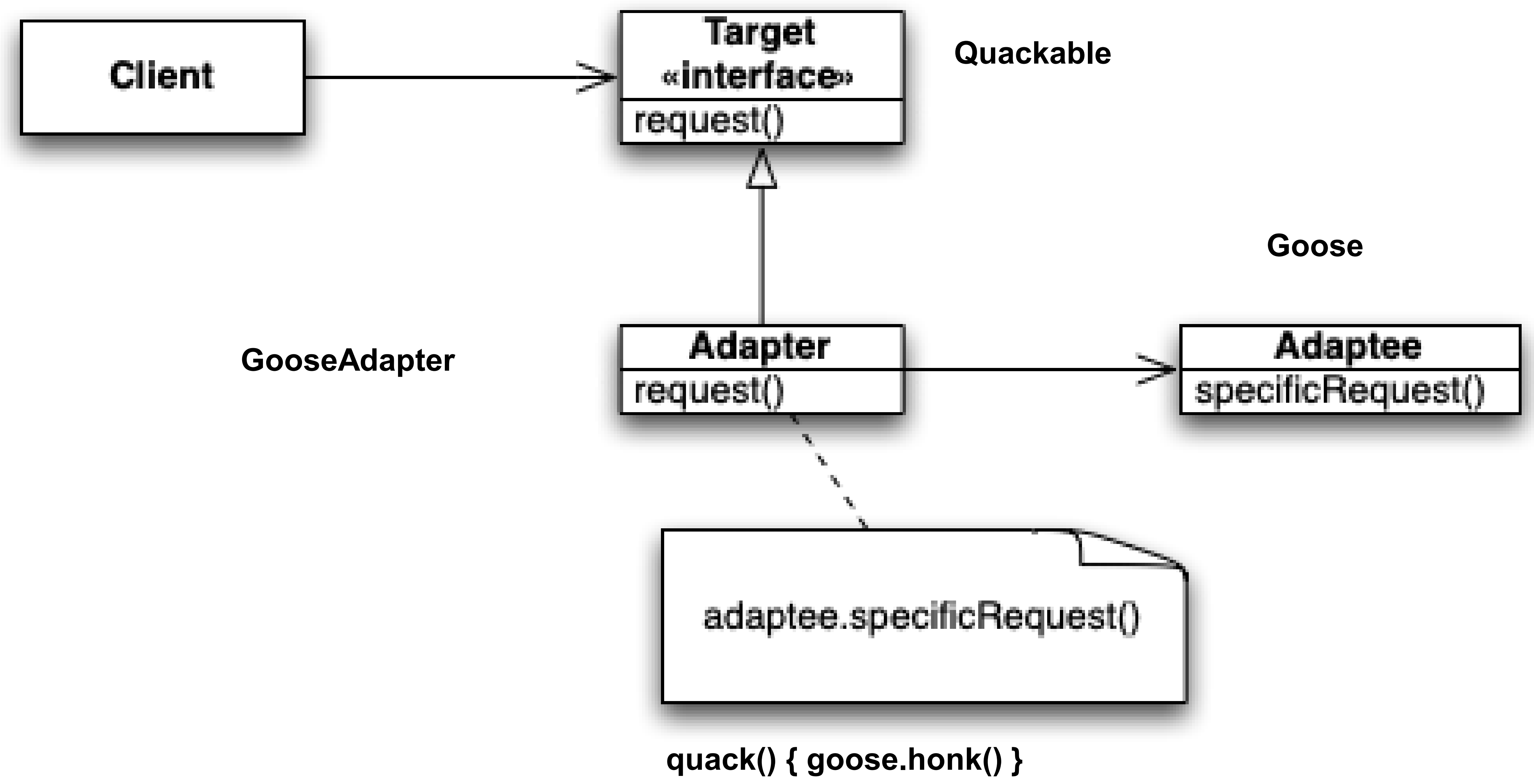


Step 3: Need Adapter

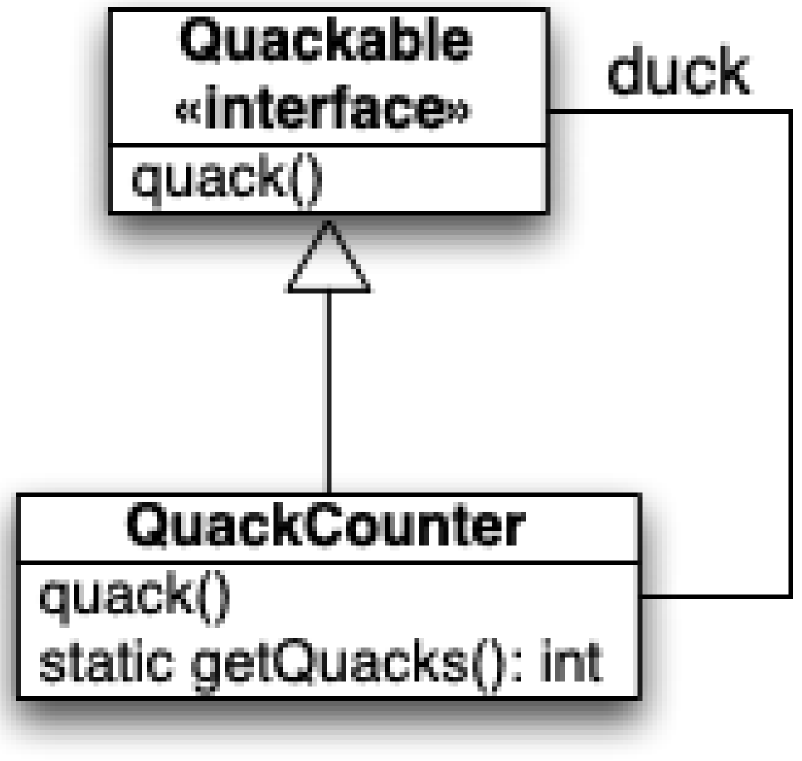


- All participants are now Quackables, allowing us to treat them uniformly

Review: (Object) Adapter Structure



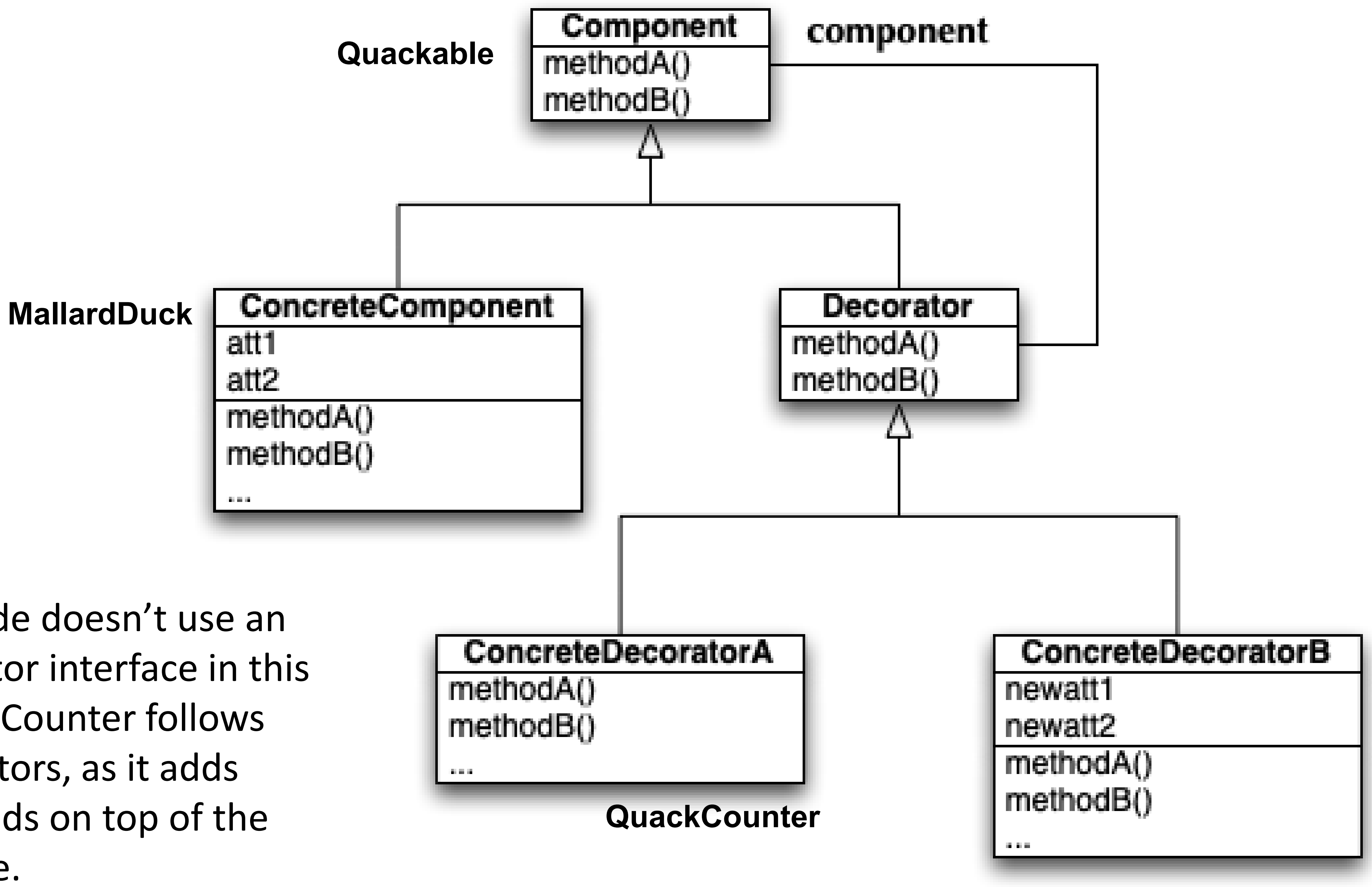
Step 4: Use Decorator to Add Quack Counting



- Note: two relationships between QuackCounter and Quackable
- Inheritance of Quackable
- Reference in QuackCounter constructor to duck

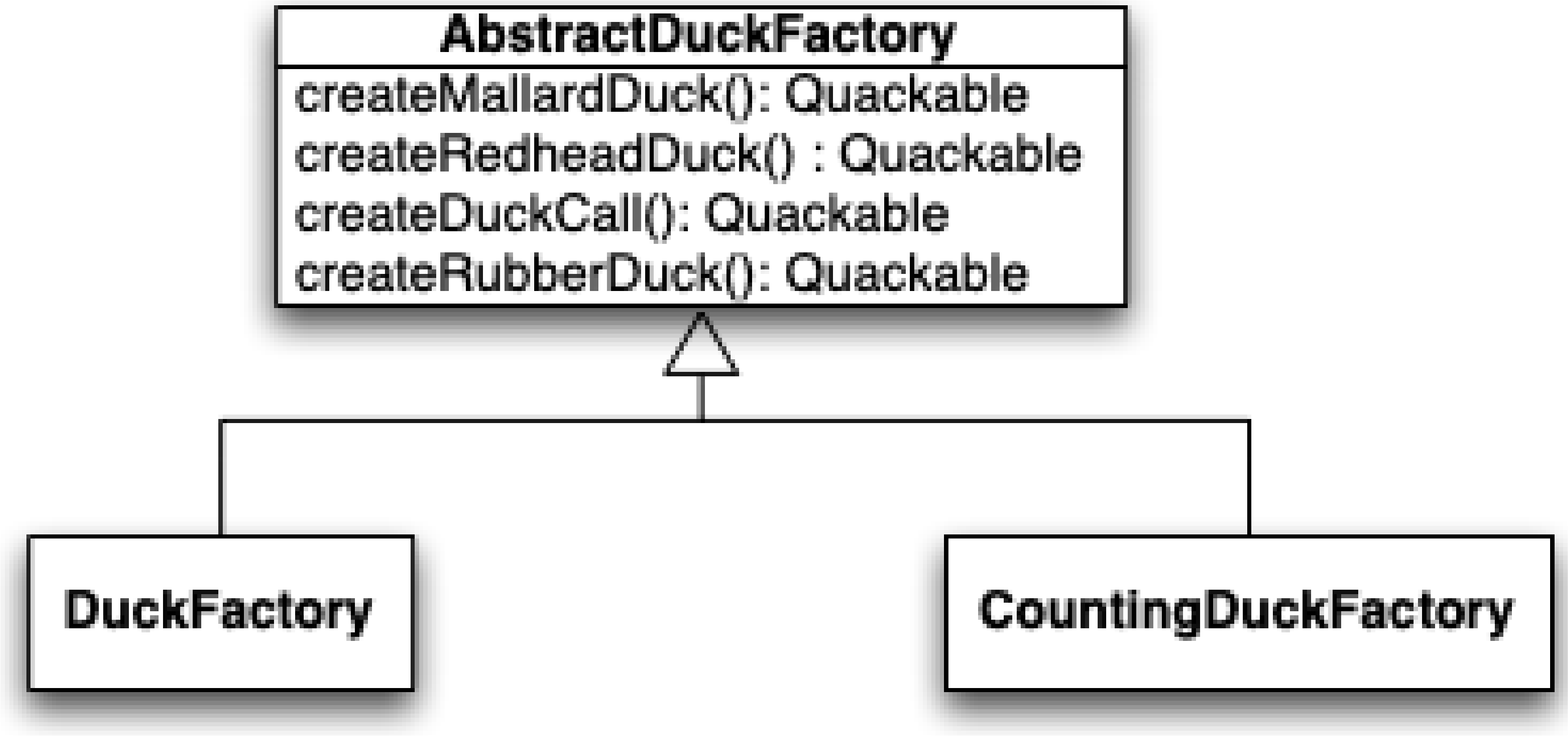
Previous classes/relationships are all still there… just edited for clarity

Review: Decorator Structure



The example code doesn't use an abstract Decorator interface in this situation; QuackCounter follows ConcreteDecorators, as it adds state and methods on top of the original interface.

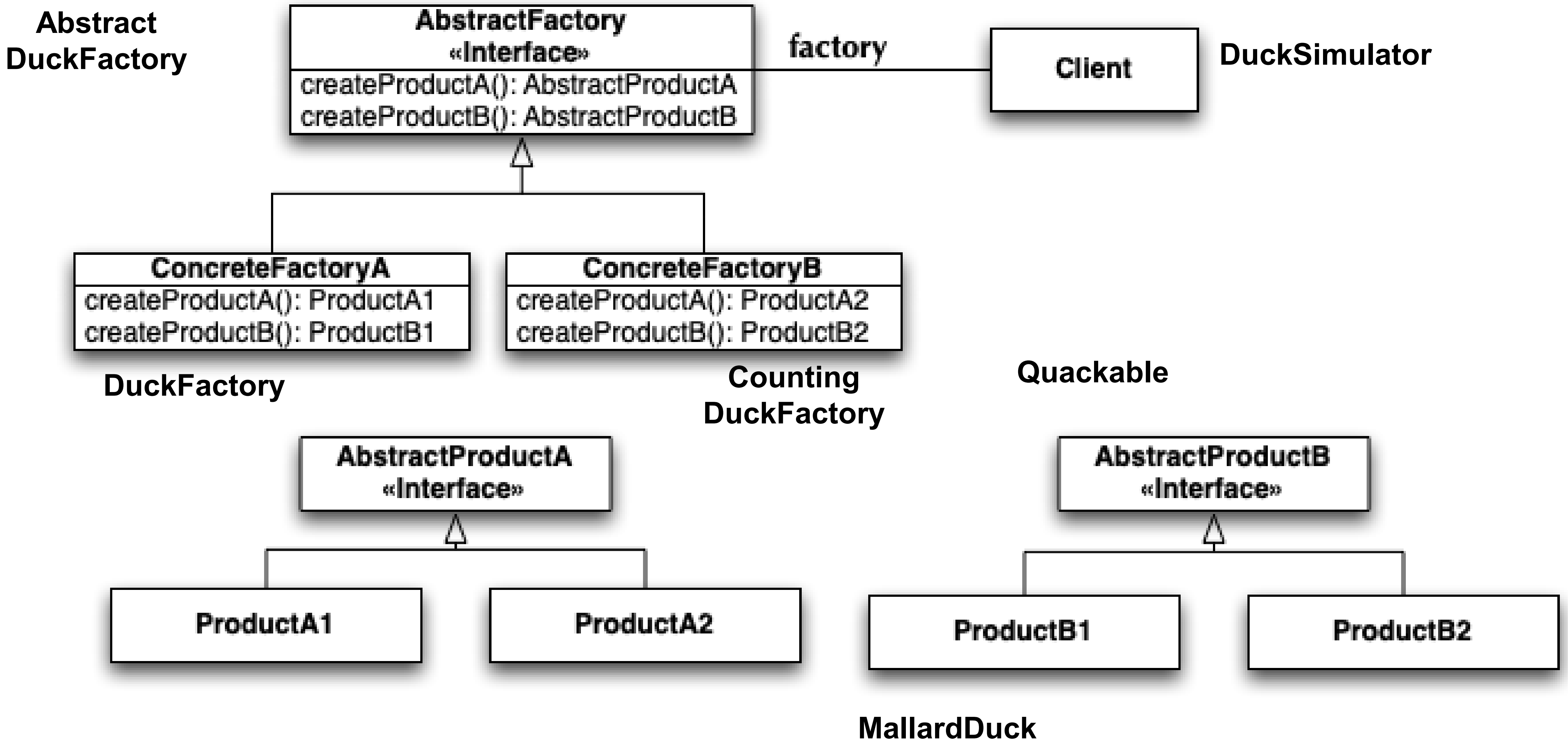
Step 5: Add Factory to Control Duck Creation


AbstractDuckFactory
createMallardDuck(): Quackable
createRedheadDuck() : Quackable
createDuckCall(): Quackable
createRubberDuck(): Quackable
DuckFactory
CountingDuckFactory

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

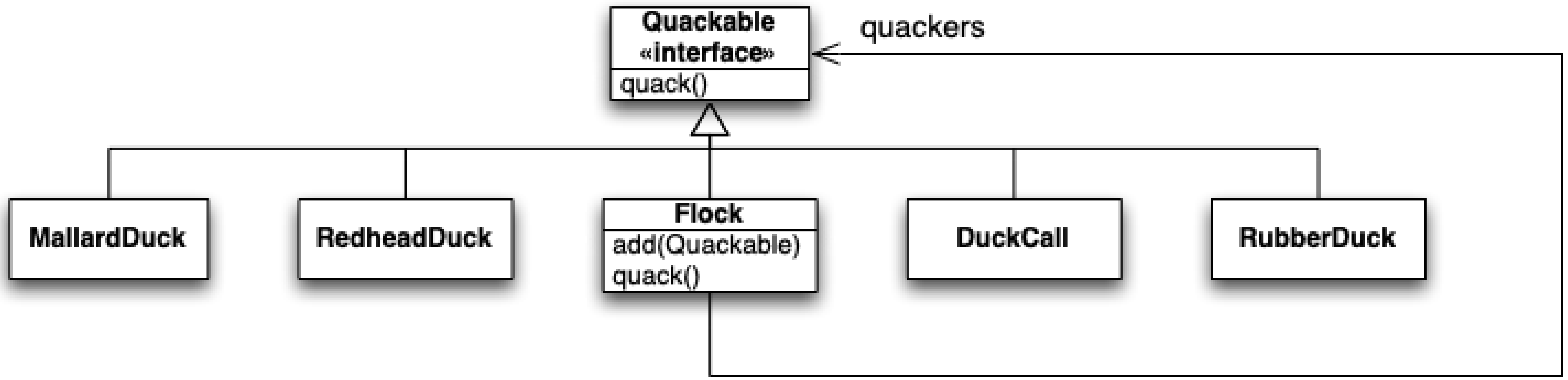
This code is used by a method in DuckSimulator that accepts an instance of AbstractDuckFactory as a parameter.

Review: Abstract Factory Structure



Abstract DuckFactory
AbstractFactory «Interface»
createProductA(): AbstractProductA
createProductB(): AbstractProductB
factory
Client
DuckSimulator
ConcreteFactoryA
createProductA(): ProductA1
createProductB(): ProductB1
ConcreteFactoryB
createProductA(): ProductA2
createProductB(): ProductB2
DuckFactory
Counting DuckFactory
Quackable
AbstractProductA «Interface»
AbstractProductB «Interface»
ProductA1
ProductA2
ProductB1
ProductB2
MallardDuck

Step 6: Add support for Flocks with Composite



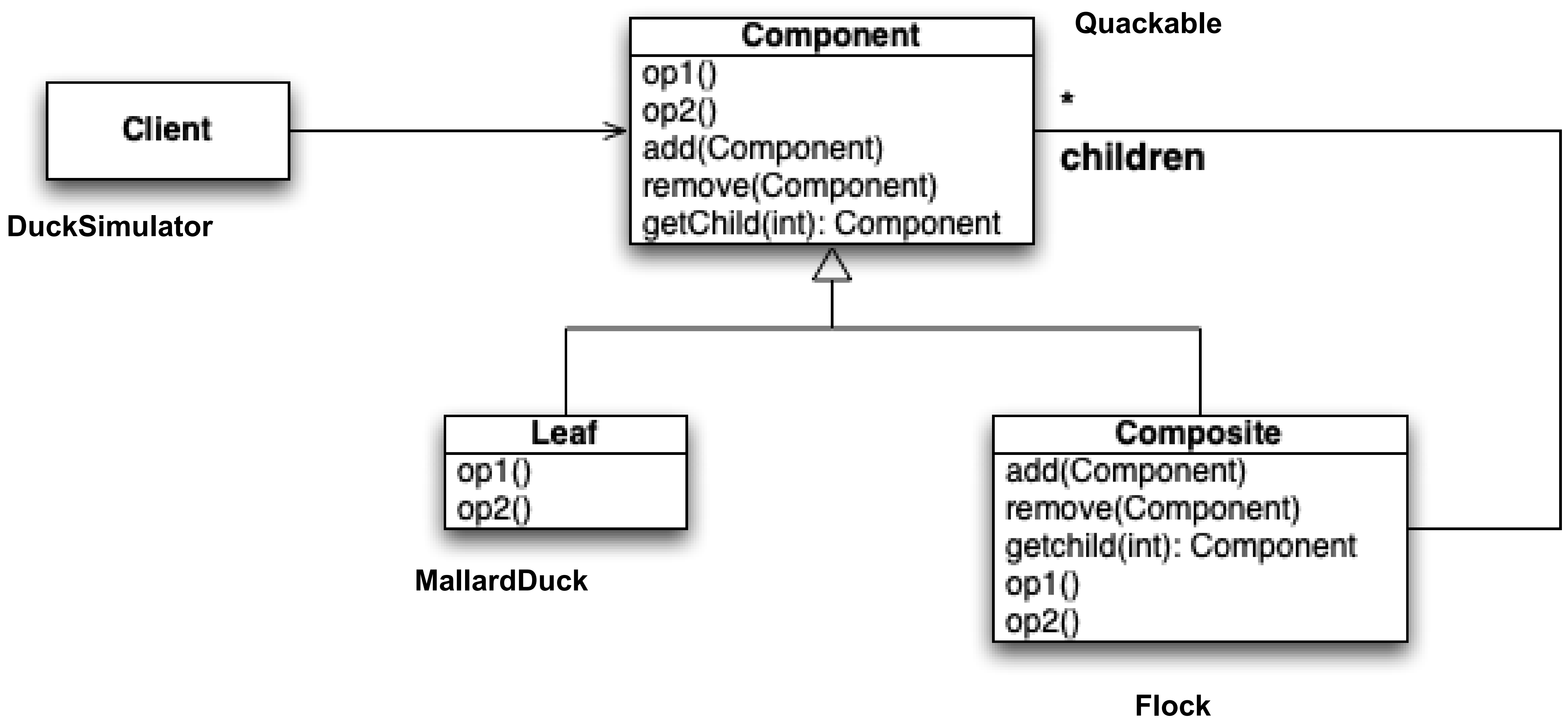
Note: In the book example, there's an Iterator pattern hiding inside of Flock.quack();

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

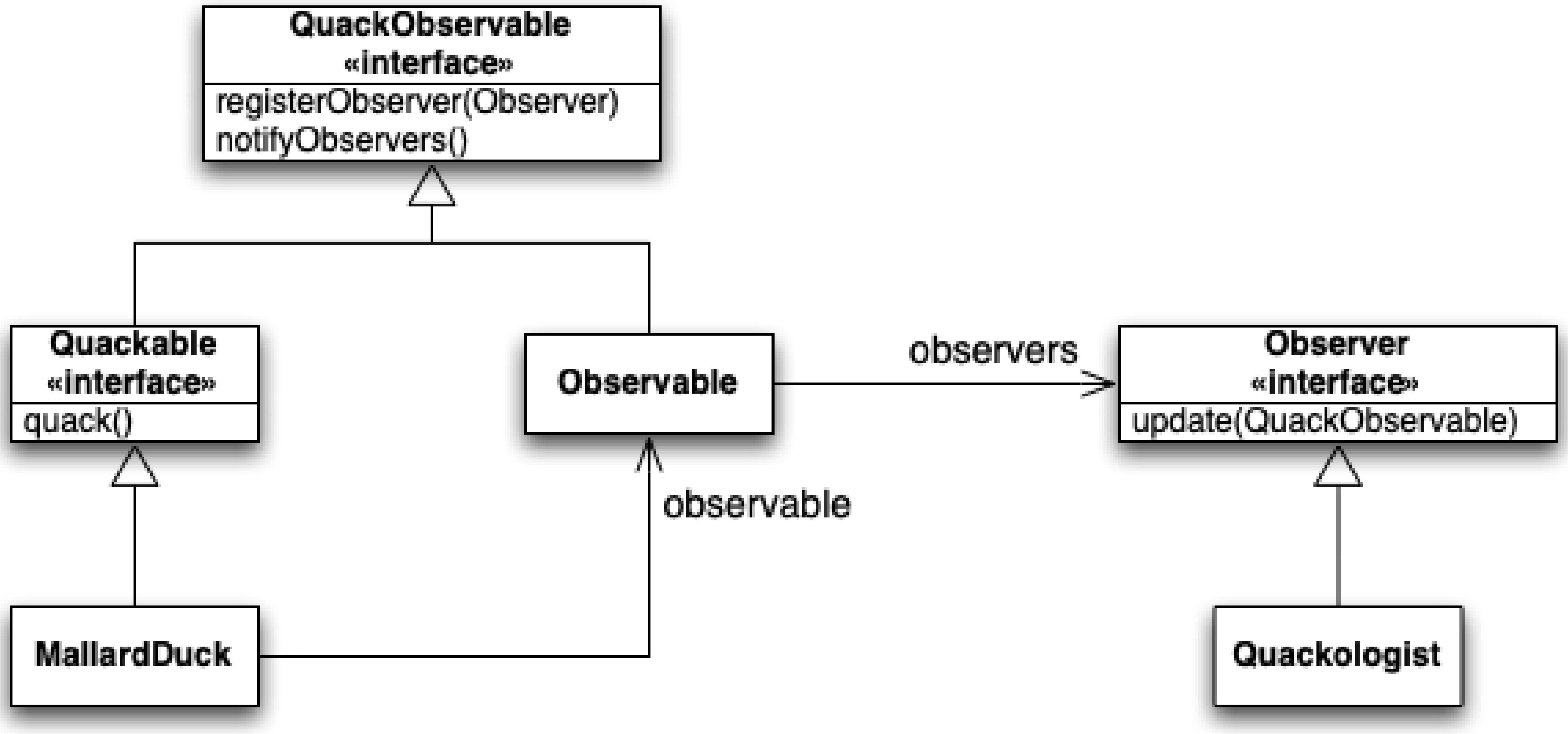
Only Flock has the "add(Quackable)" method.

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

Review: Composite Structure

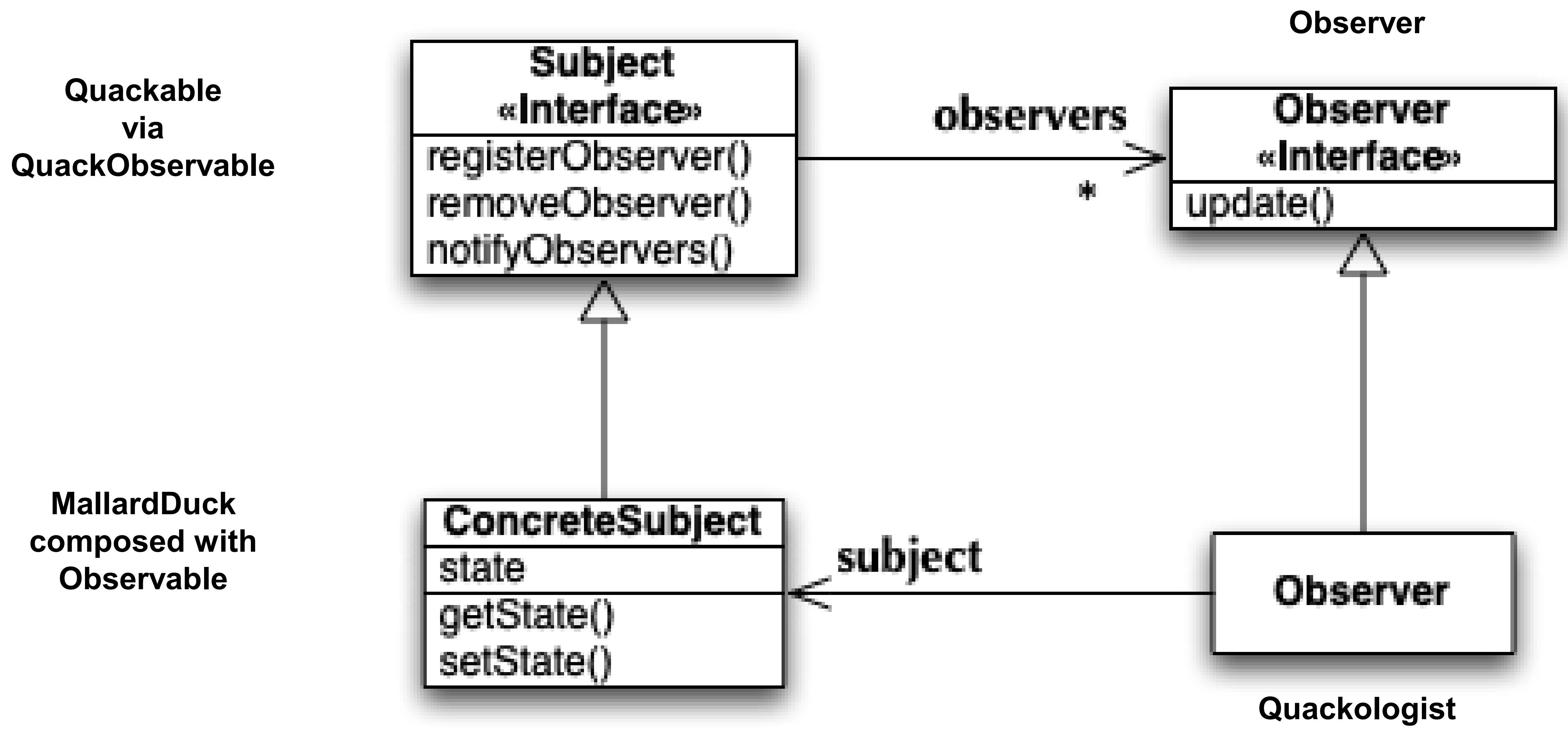


Step 7: Add Quack Notification via Observer



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.

Review: Observer Structure

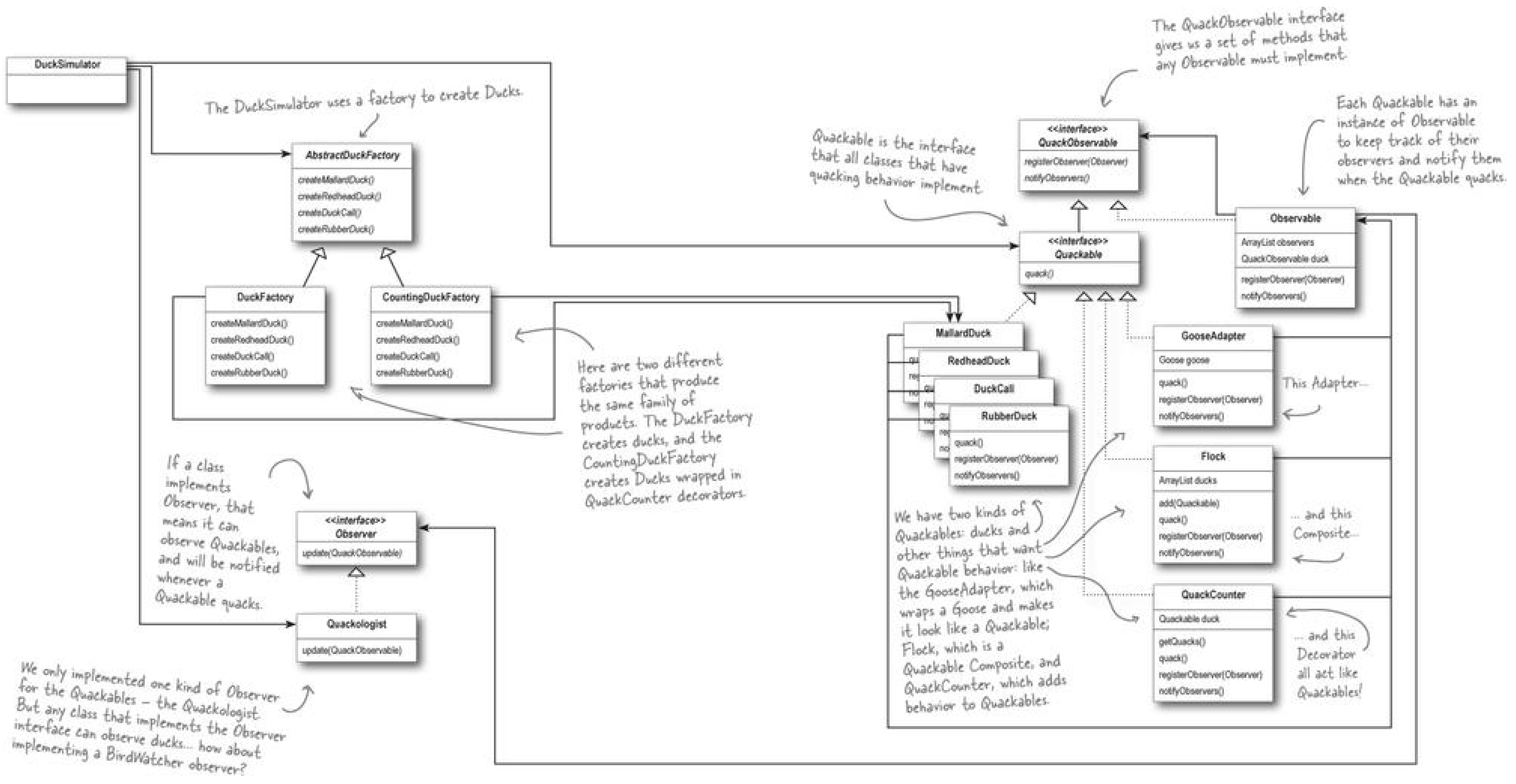


Counting Roles

- 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.

The whole Duck Simulator…

- …is too big to show here. When you get a chance, look at it in Chapter 12 of the textbook.



The DuckSimulator uses a factory to create Ducks.

Here are two different factories that produce the same family of products. The DuckFactory creates ducks, and the CountingDuckFactory creates Ducks wrapped in QuackCounter decorators.

If a class implements Observer, that means it can observe Quackables, and will be notified whenever a Quackable quacks.

We only implemented one kind of Observer for the Quackables – the Quackologist. But any class that implements the Observer interface can observe ducks... how about implementing a BirdWatcher observer?

The QuackObservable interface gives us a set of methods that any Observable must implement.

Quackable is the interface that all classes that have quacking behavior implement.

Each Quackable has an instance of Observable to keep track of their observers and notify them when the Quackable quacks.

We have two kinds of Quackables: ducks and other things that want Quackable behavior: like the GooseAdapter, which wraps a Goose and makes it look like a Quackable; Flock, which is a Quackable Composite, and QuackCounter, which adds behavior to Quackables.

This Adapter...

... and this Composite...

... and this Decorator all act like Quackables!

Model-View-Controller: A Pattern of Patterns

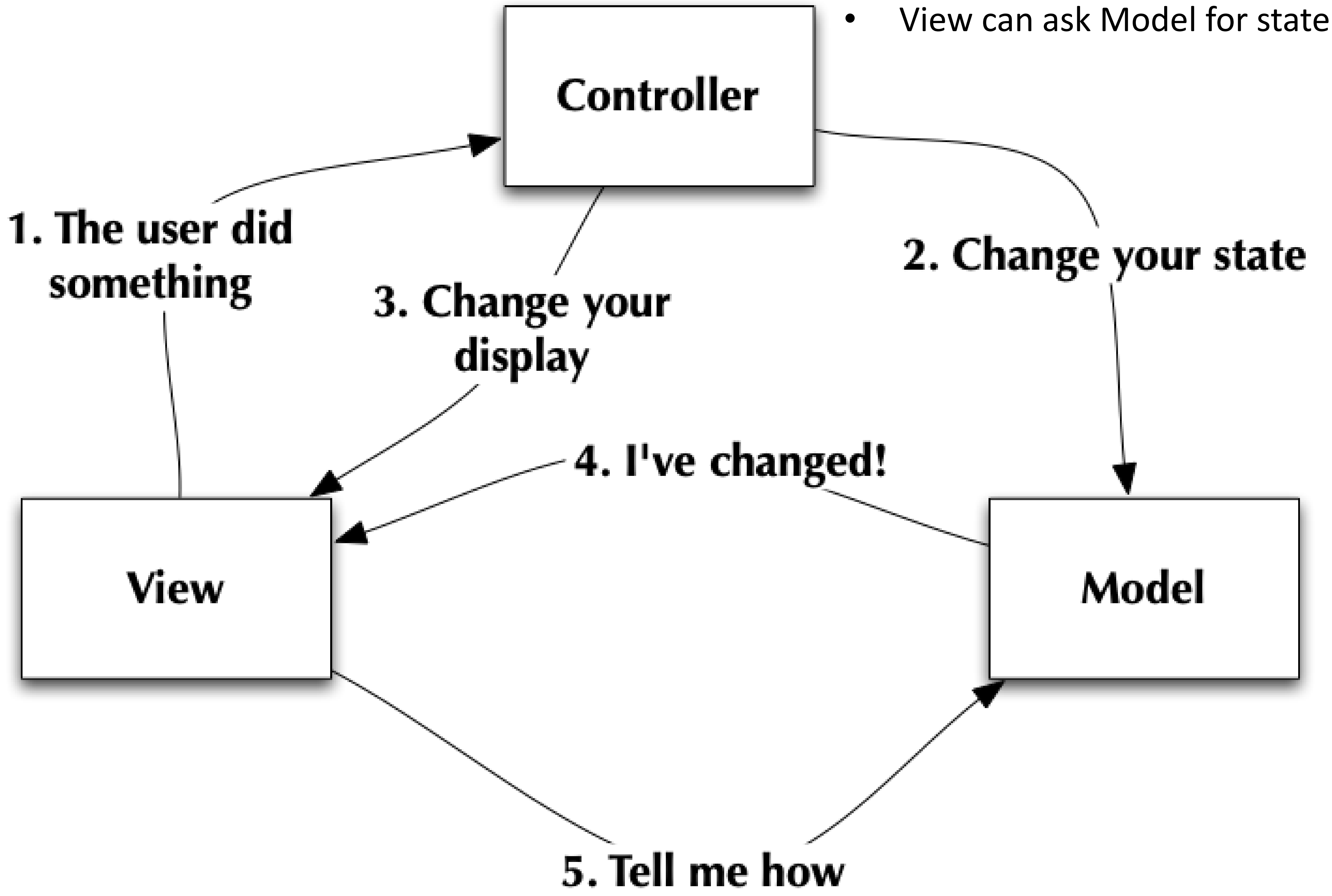
- Model-View-Controller (MVC) is a ubiquitous pattern that allows information (stored in models) to be viewed in a number of different ways (views), with each view aided by code that handles user input or notifies the view of updates to its associated models (controllers)
- Started as a framework developed for Smalltalk in the late 1970s
- MVC is NOT one of the original Gang of Four OO Patterns (although it gets mentioned as a Smalltalk framework)
- 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 usually written by hand, but can be generated
        - 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

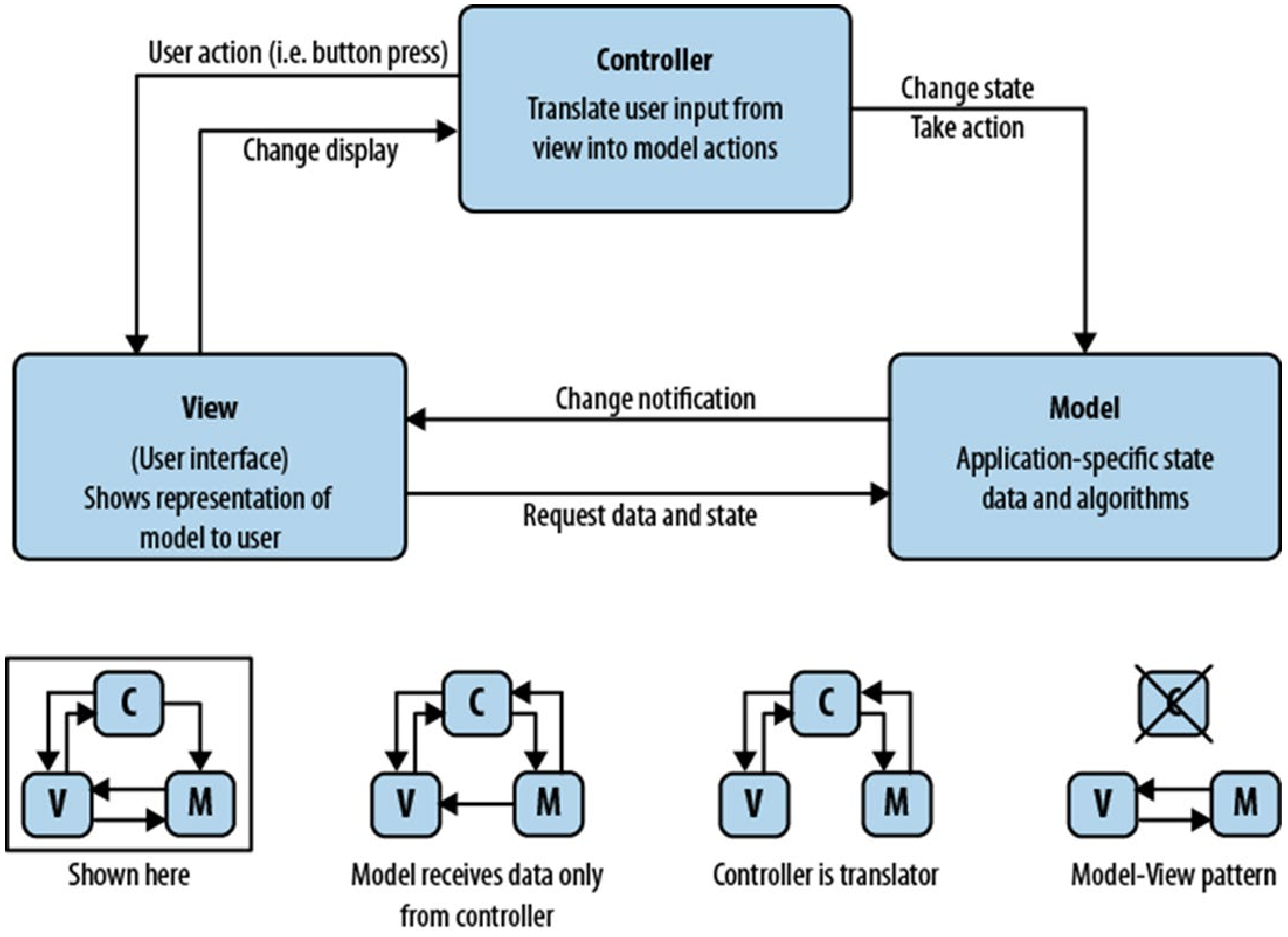
MVC: Structure

- User interacts with View
- Controller changes the Model state
- Controller renders the View
- Model updates the View
- View can ask Model for state



More MVC observations

- No one true approach to this pattern, many variations
- You can find many variations and examples of the rules of engagement between the View, the Controller, and the Model
- Key is defining what the abstraction and interface is for each element
- Enables switching out alternate MVC elements
- Also supports independent development of MVC elements



User action (i.e. button press)
Change display
Controller
Translate user input from view into model actions
Change state
Take action
View
(User interface)
Shows representation of model to user
Change notification
Request data and state
Model
Application-specific state data and algorithms
C
V
M
Shown here
Model receives data only from controller
Controller is translator
Model-View pattern

From White, Making Embedded Systems

MVC UML Diagram

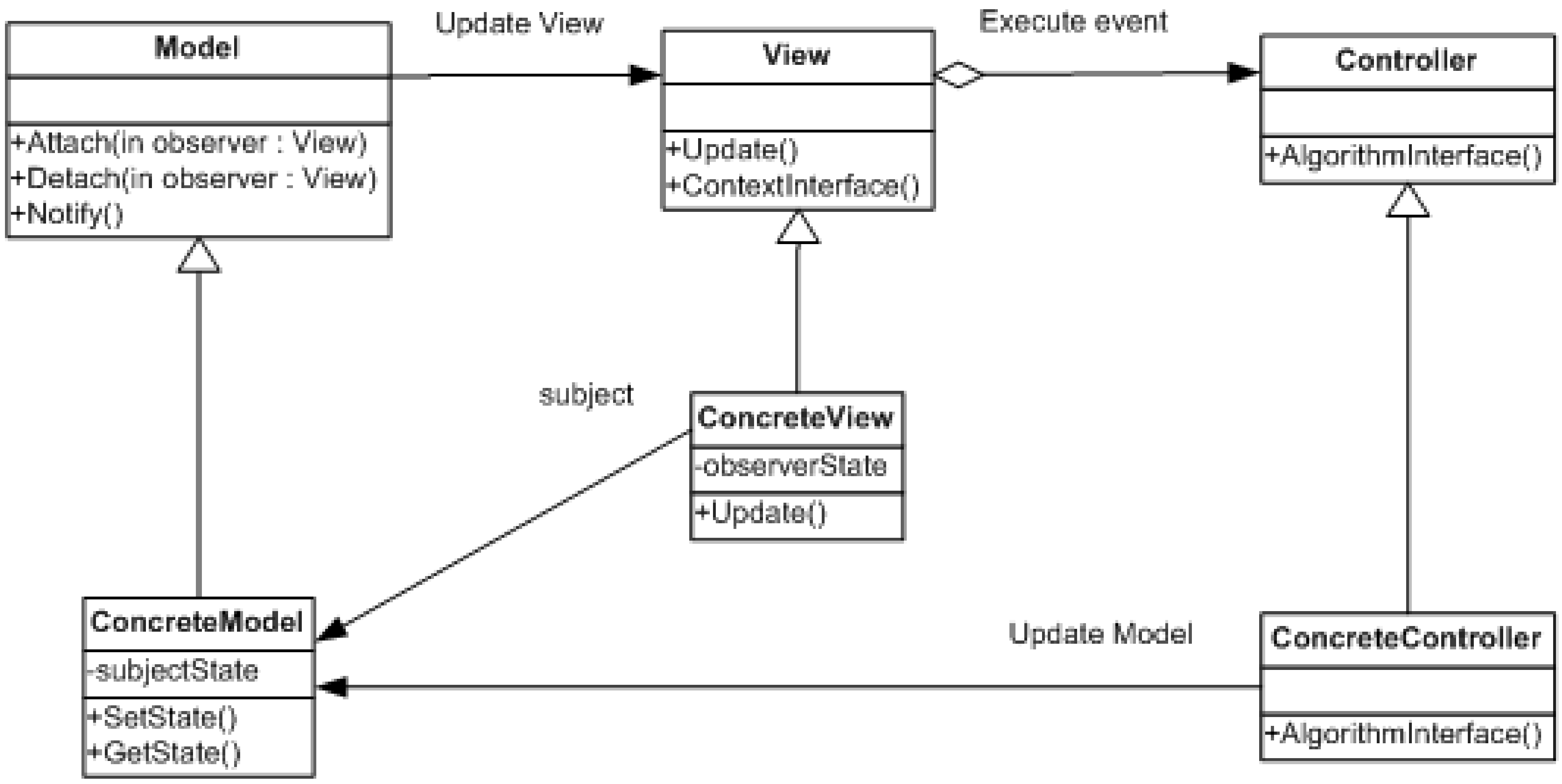


Figure 3: MVC

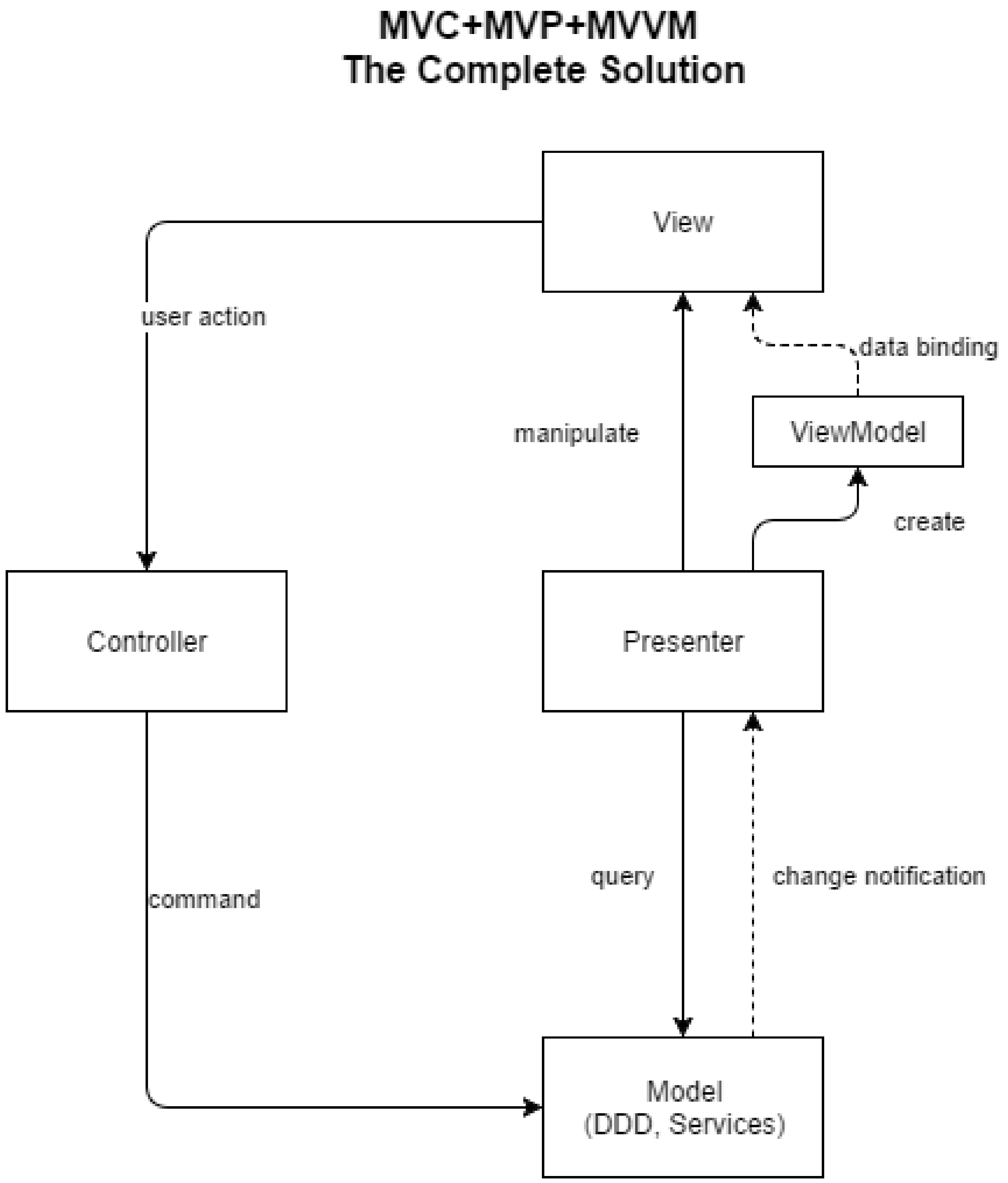
- Simple MVC Diagram
- https://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod

MVC: A Pattern of Patterns

- MVC is a compound pattern
- 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 a Command pattern, views can be dynamically augmented with the Decorator pattern, etc.

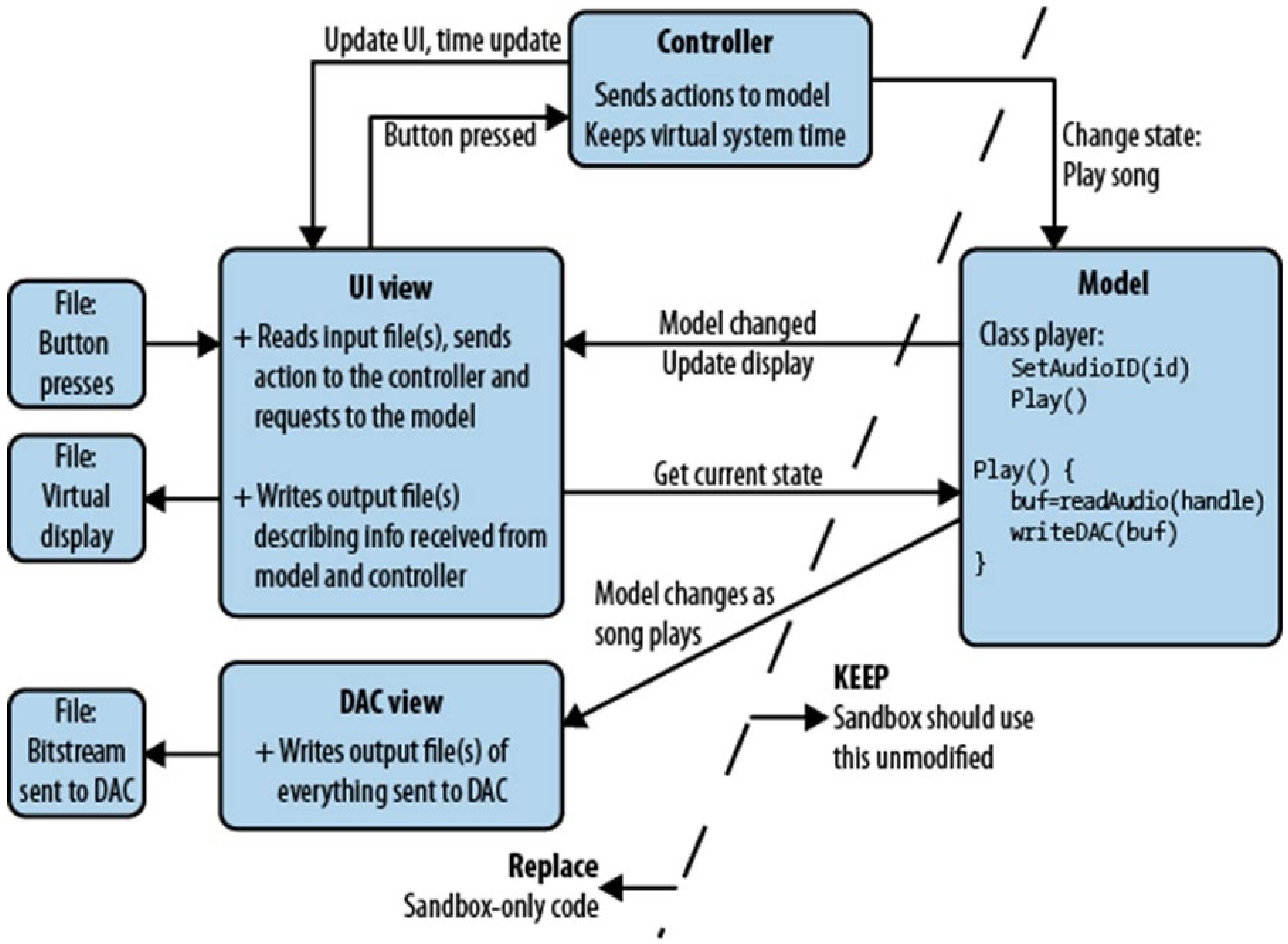
MVC: Variations

- MVP – Model View Presenter
  - Presenter handles all presentation logic
- MVVM – Model View ViewModel
  - Decouples the view from a model of the view
- Many more related patterns
- Nice comparison here: http://gexiaoguo.github.io/MVC,-MVP-and-MVVM/
- Java examples at https://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod



Using MVC variations for your application

- The pattern is easily adapted by defining the role and interface of each MVC element in your application
- Here's an example of using MVC to support development of an embedded device application with malleable "sandbox" code for views and controller talking to a more defined model element



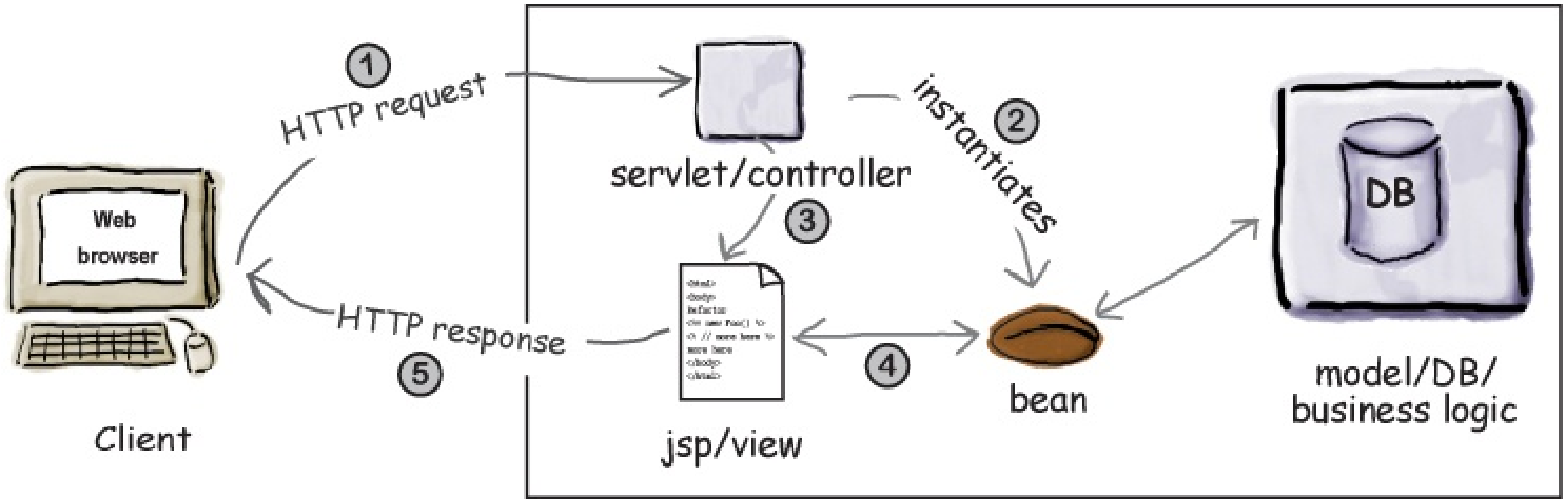
From White, Making Embedded Systems

Very Flexible Pattern

- Book provides an example of a DJ application
- One Model, One Controller, Multiple Views
  - Consider multiple open windows in MacOS X Finder
- One Model, Multiple Controllers, Multiple Views
  - Same example but with Finder windows in multiple modes (icon, list, column)
  - Another example: Spreadsheet, rows and columns view and chart view
- And in certain cases, almost all of the view, controller, and model can be automated
  - Example: An OS X application that makes use of Core Data, Cocoa Bindings, and a XIB file; created without writing a single line of code!

Model 2

- Model 2 is an adaptation of MVC to browser/server models



1. You make an HTTP request, received by a servlet
2. Servlet acts as controller, instantiates a JavaBean for DB requests
3. Controller gives control to View (A JSP or JavaServer Page)
4. View returns a page to Browser over HTTP

- Full Example with Code in Chapter 12

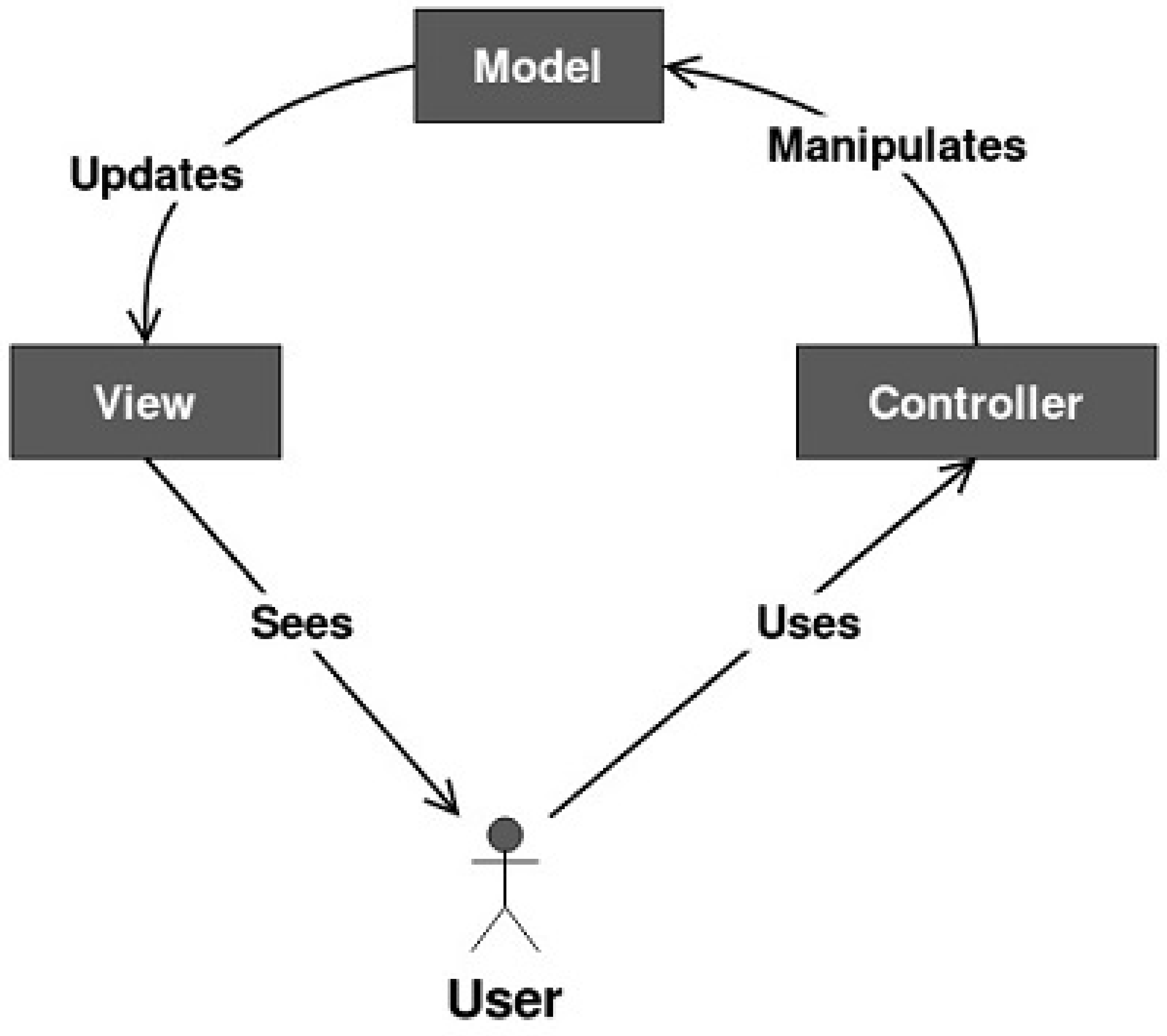
MVC Examples for Python

- Super simple example at

  https://www.tutorialspoint.com/python_design_patterns/python_design_patterns_model_view_controller.htm

- Another summary of MVC in Python – with Legos!

  https://realpython.com/the-model-view-controller-mvc-paradigm-summarized-with-legos/



Wrapping Up

- Looked at patterns for managing collections of objects…
- 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
- The use of multiple patterns within a system provides a significant amount of flexibility and extensibility in a software system
  - Classes will typically play multiple, well-defined roles in such systems
  - Will sometimes manifest as compound patterns, such as MVC

Final Points

- The Model View Controller Pattern (MVC) is a compound pattern consisting of the (at least) Observer, Strategy and Composite patterns
- The model makes use of the Observer Pattern so that it can keep observers updated yet stay decoupled from them
- The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior.
- The view uses the Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames and buttons
- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible
- You can vary the interfaces and rules of how MVC elements talk to fit your design
- The Adapter Pattern can be used to adapt a new model to an existing view and controller
- Model 2 is an adaptation of MVC for web applications
- In Model 2, the controller is implemented as a servlet and JSP & HTML implement the view