Design Patterns

Various Chapters of Head First Design Patterns

Chapter 4

Baking with OO Goodness

When you see "new" think concrete

```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

Pizza Shop application

```
Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese") {
        pizza = new CheesePizza();
    } else if (type.equals("greek") {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

part that varies

part that stays the same

We'd like to remove and encapsulate this

Simple Pizza Factory

```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese") {
            pizza = new CheesePizza();
        } else if (type.equals("greek") {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni") {
            pizza = new PepperoniPizza();
        }
        return pizza;
    }
}
```

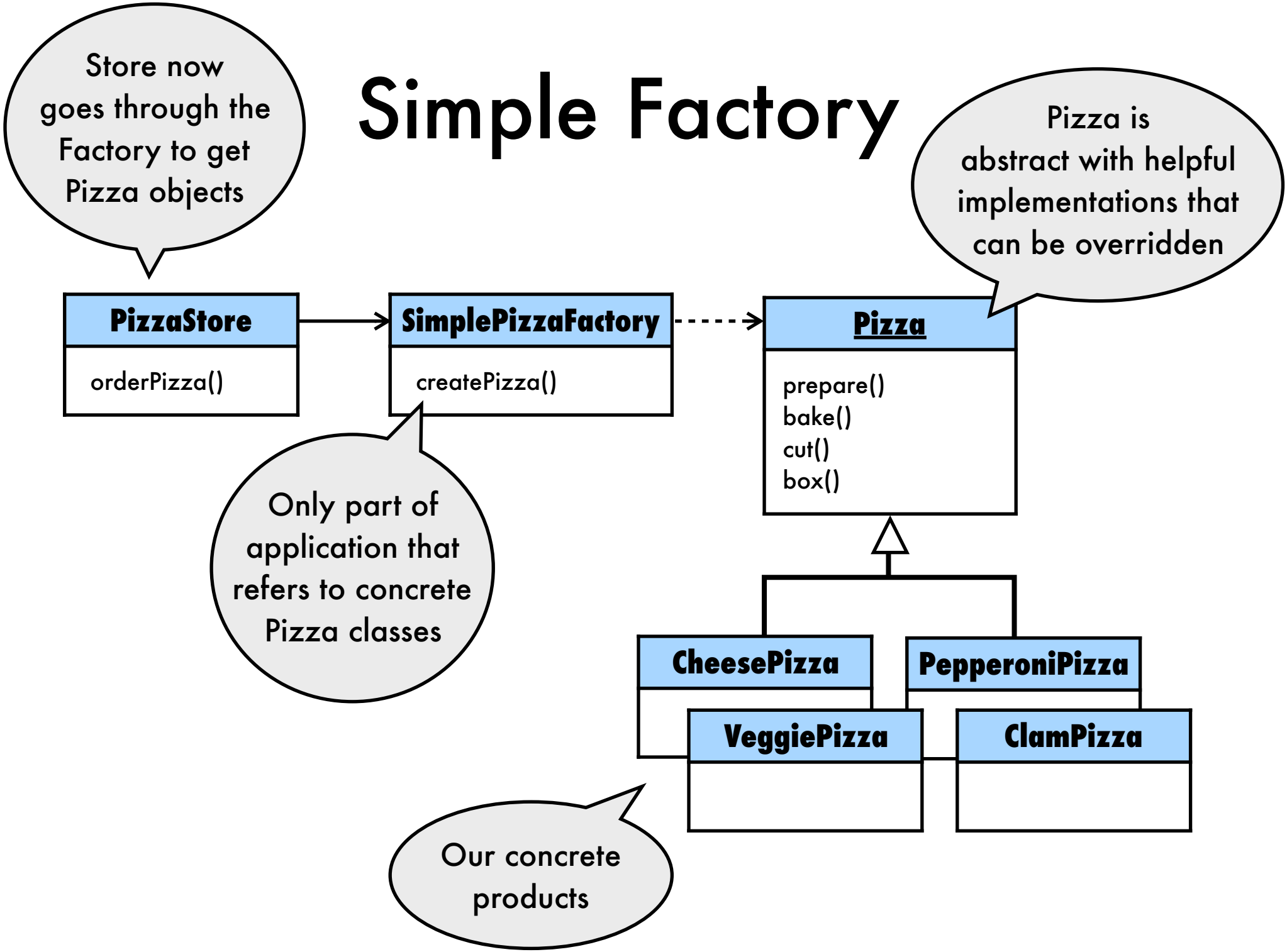
```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(
            SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Simple Factory



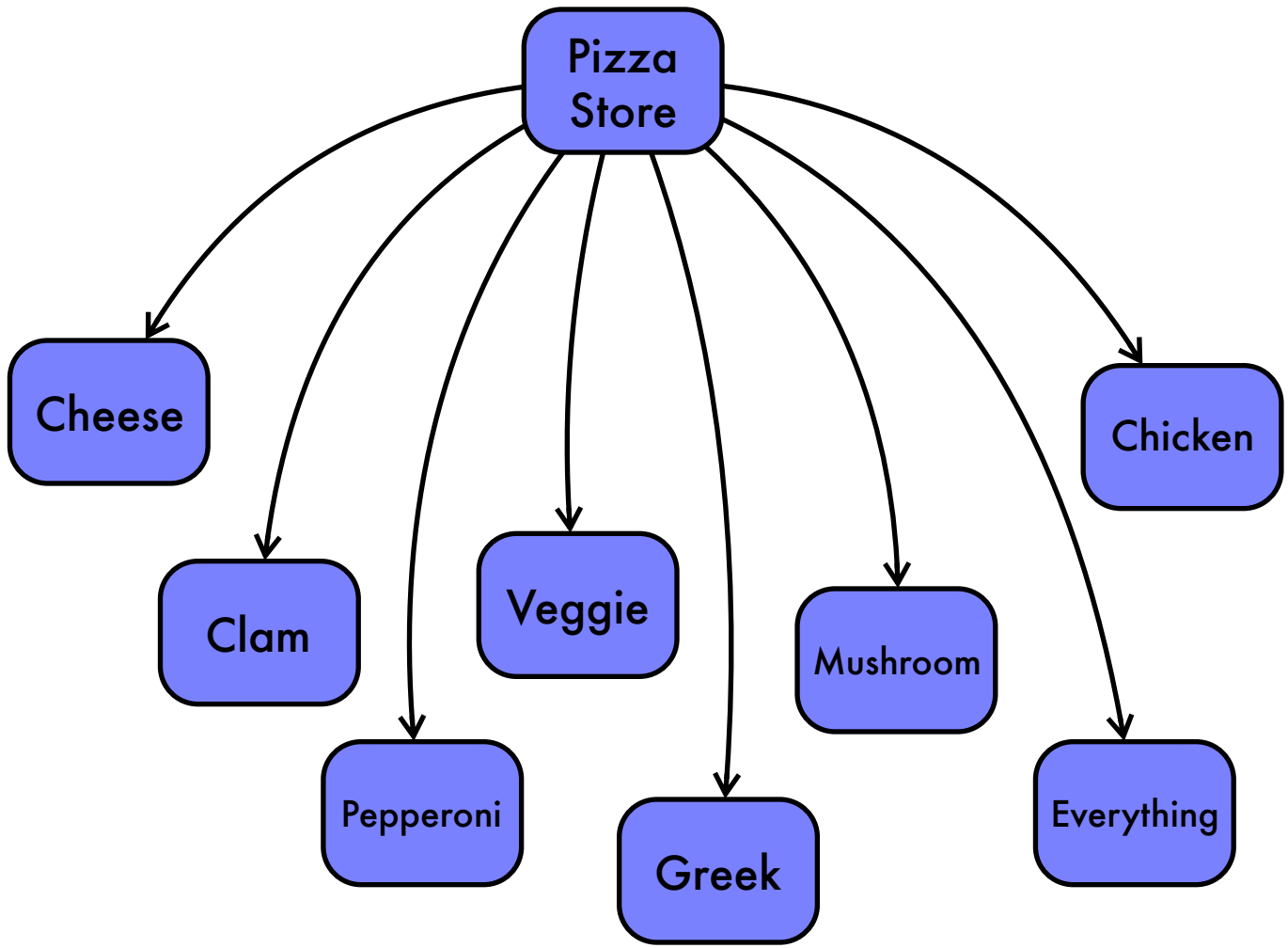
Simple Factory

The Simple Factory is a programming idiom rather than a design pattern. An idiom is an expression of a simple task that may not be built in to the programming language being used.

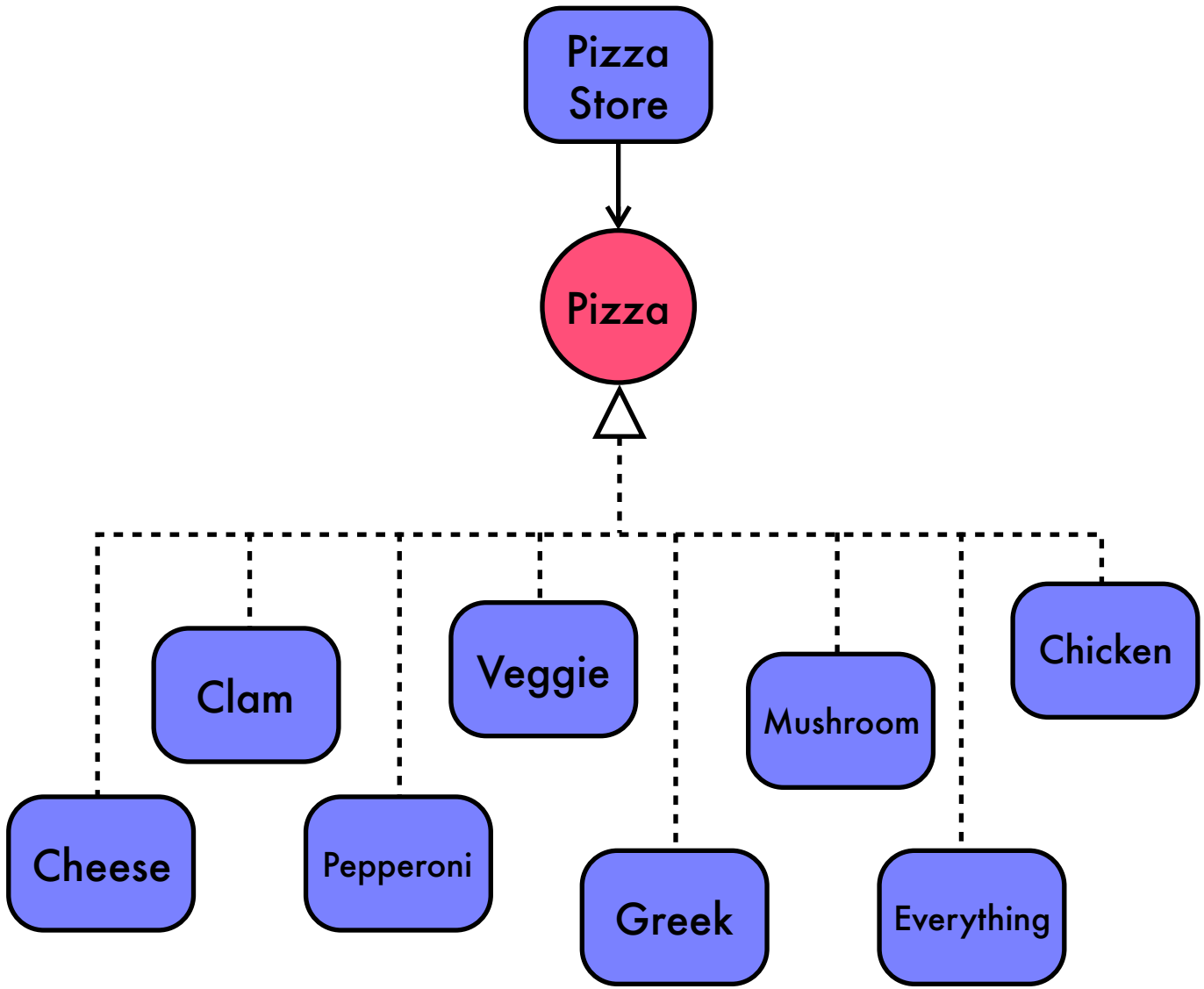
Design Principle

Depend upon abstractions. Do not depend upon concrete classes.

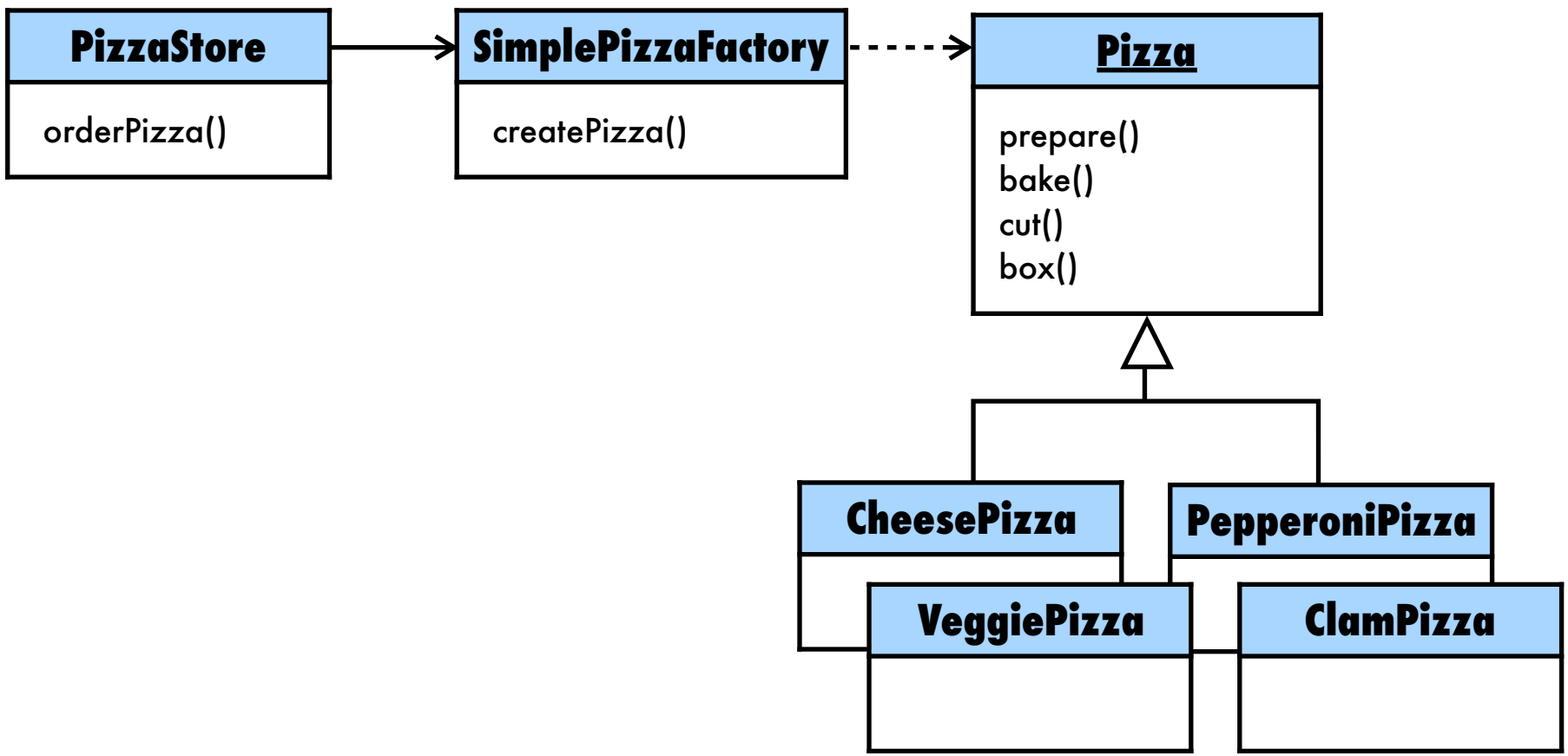
Object dependencies



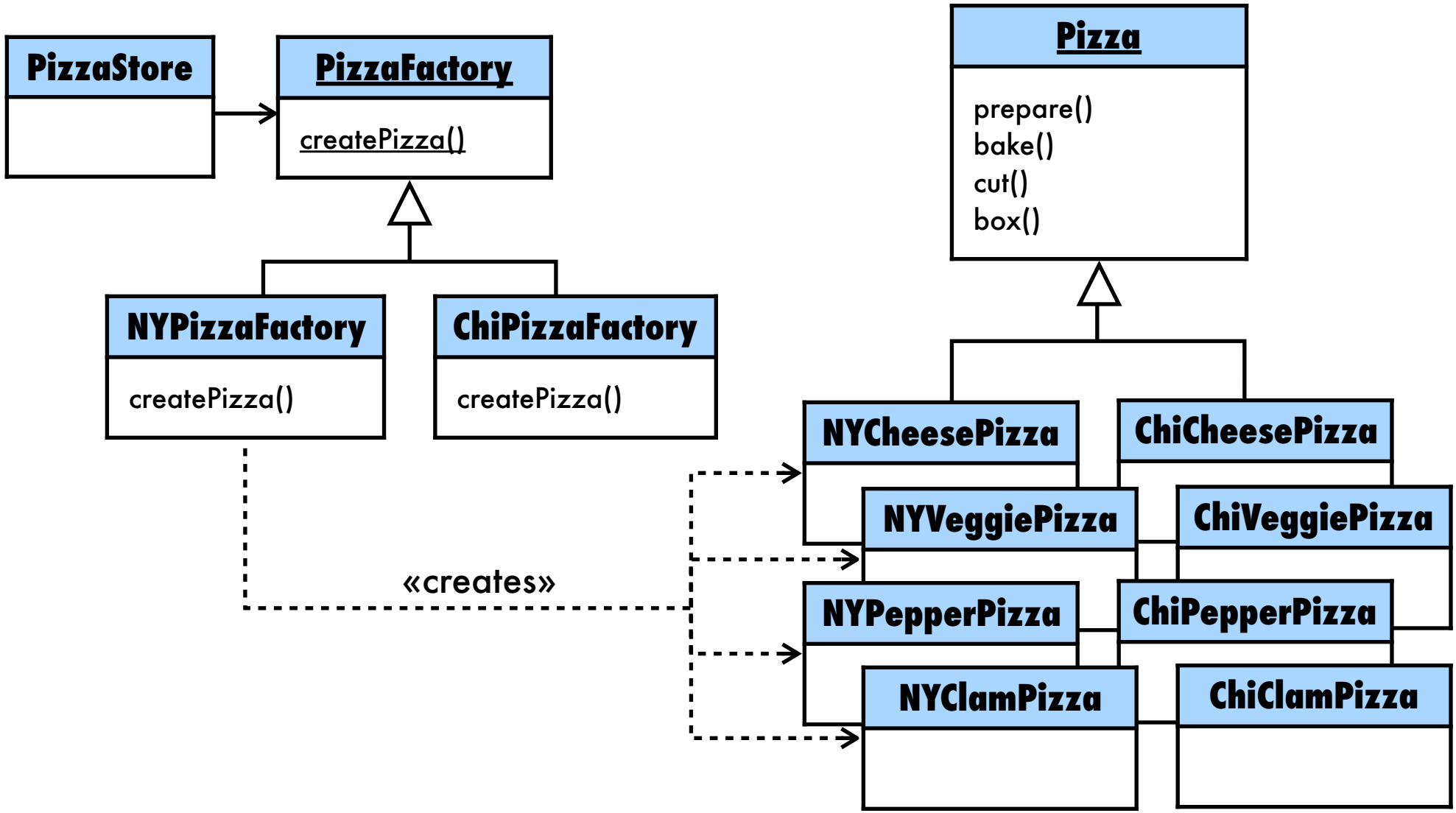
Object dependencies



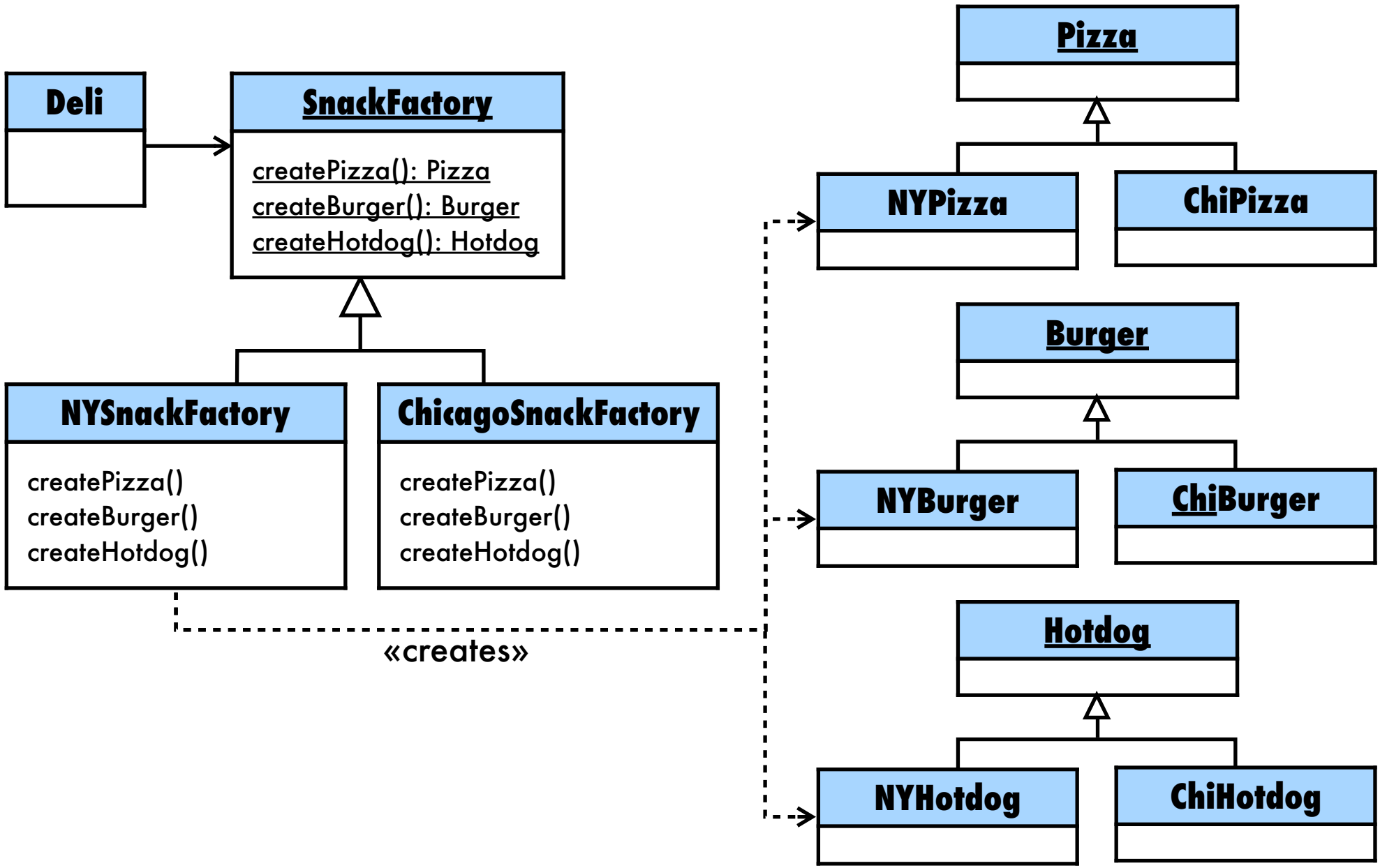
Simple Factory



Abstract factory pattern



Abstract factory pattern



Abstract factory pattern

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Chapter 5

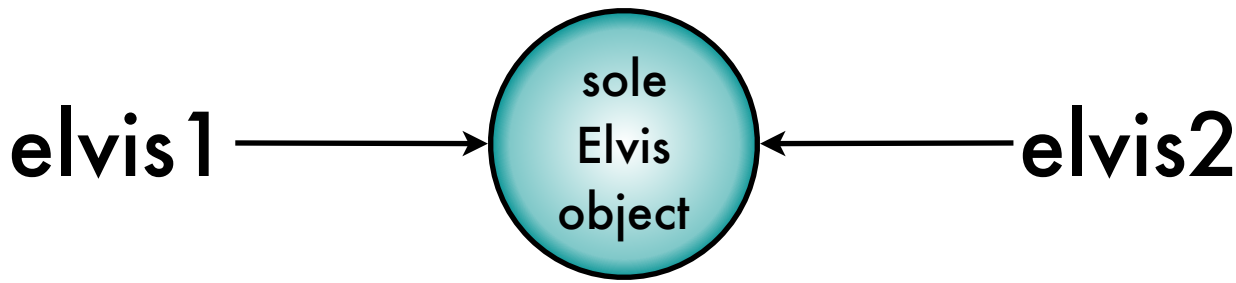
One of a kind objects

Singleton Pattern

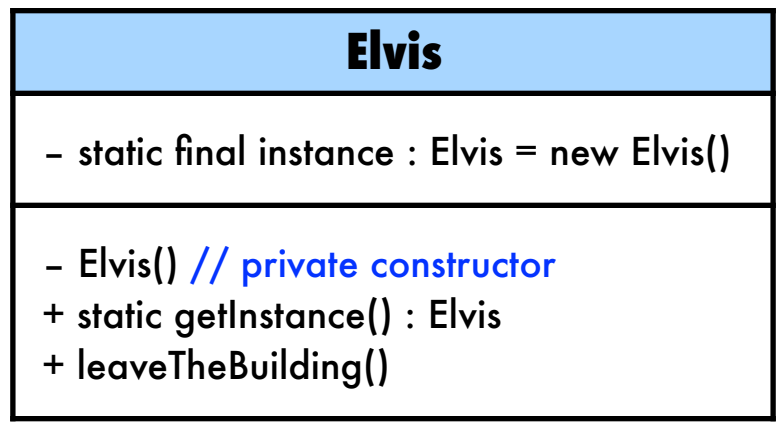
Many designers would call this an idiom

The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

Only one Elvis object

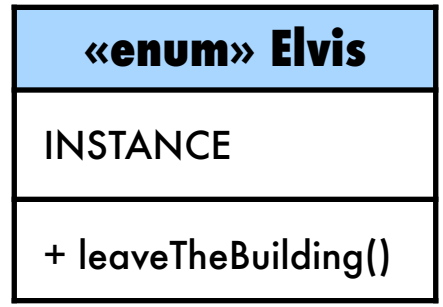


Class diagram for singleton




```
elvis = Elvis.getInstance();
elvis.leaveTheBuilding();
```

Class diagram for singleton




```
Elvis.INSTANCE.leaveTheBuilding();
```

Singleton in Scala

normal class

```
public class ElvisImpersonator {
    // fields
    // operations
}
```

singleton

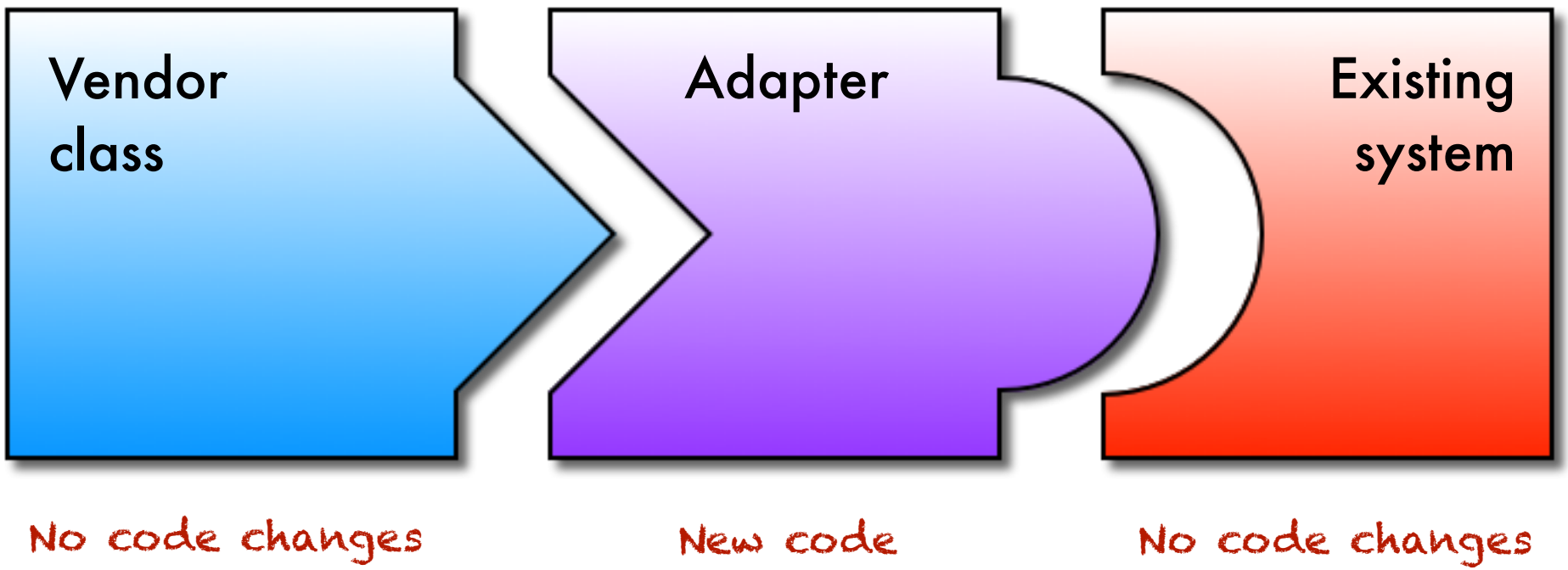
```
public object Elvis {
    // fields
    // operations
}
```

Chapter 7

Being adaptive

Adapters

OO Adapters


Vendor class
Adapter
Existing system
No code changes
New code
No code changes

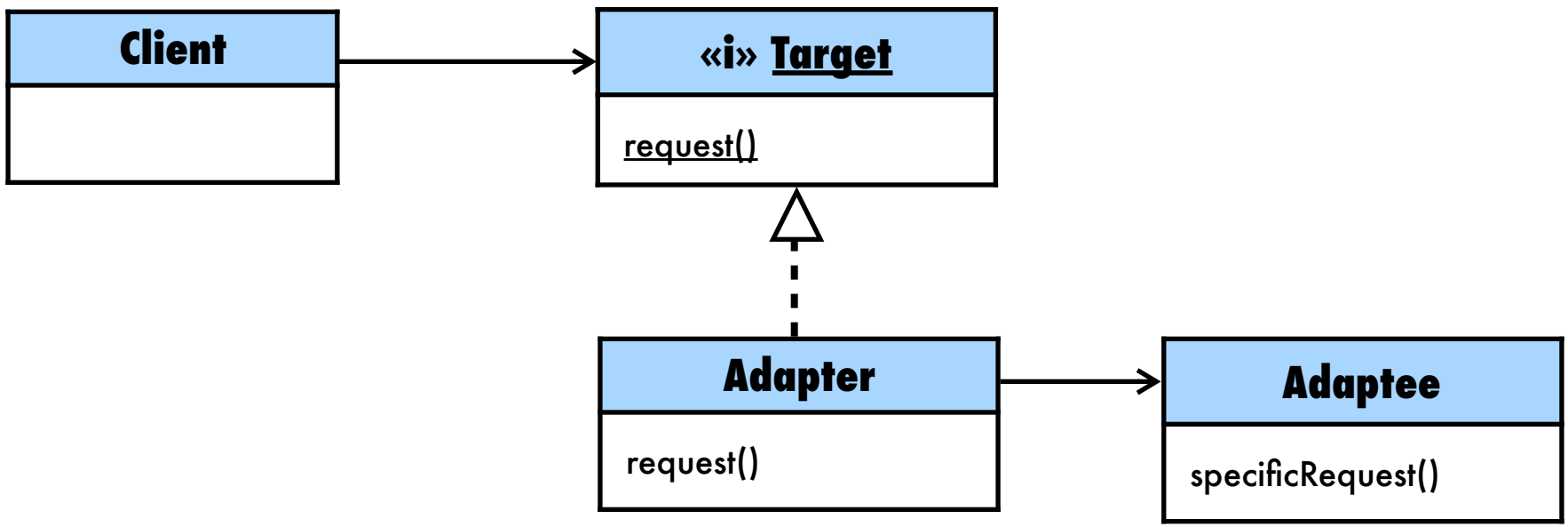
If it walks like a duck and quacks like a duck, then it must be a duck...

...unless it's a turkey wrapped with a duck adapter.

Adapter Pattern

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapter pattern class diagram



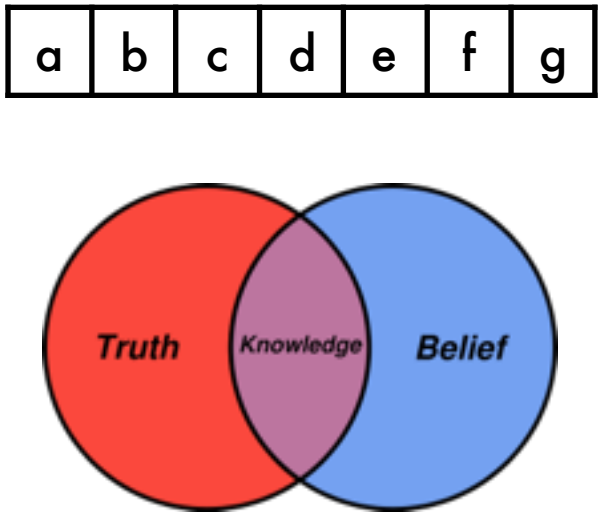
Chapter 9

Well-managed collections

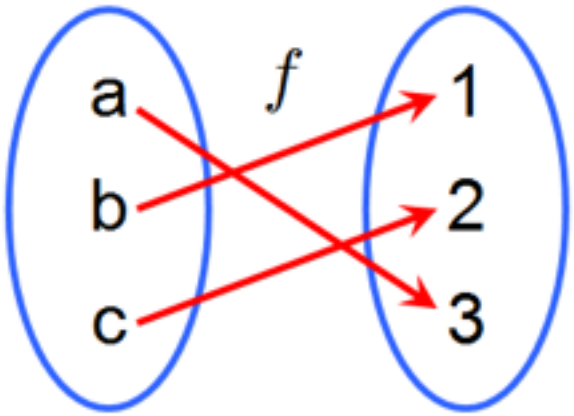
Iterator Pattern

Some designers might call this an idiom also

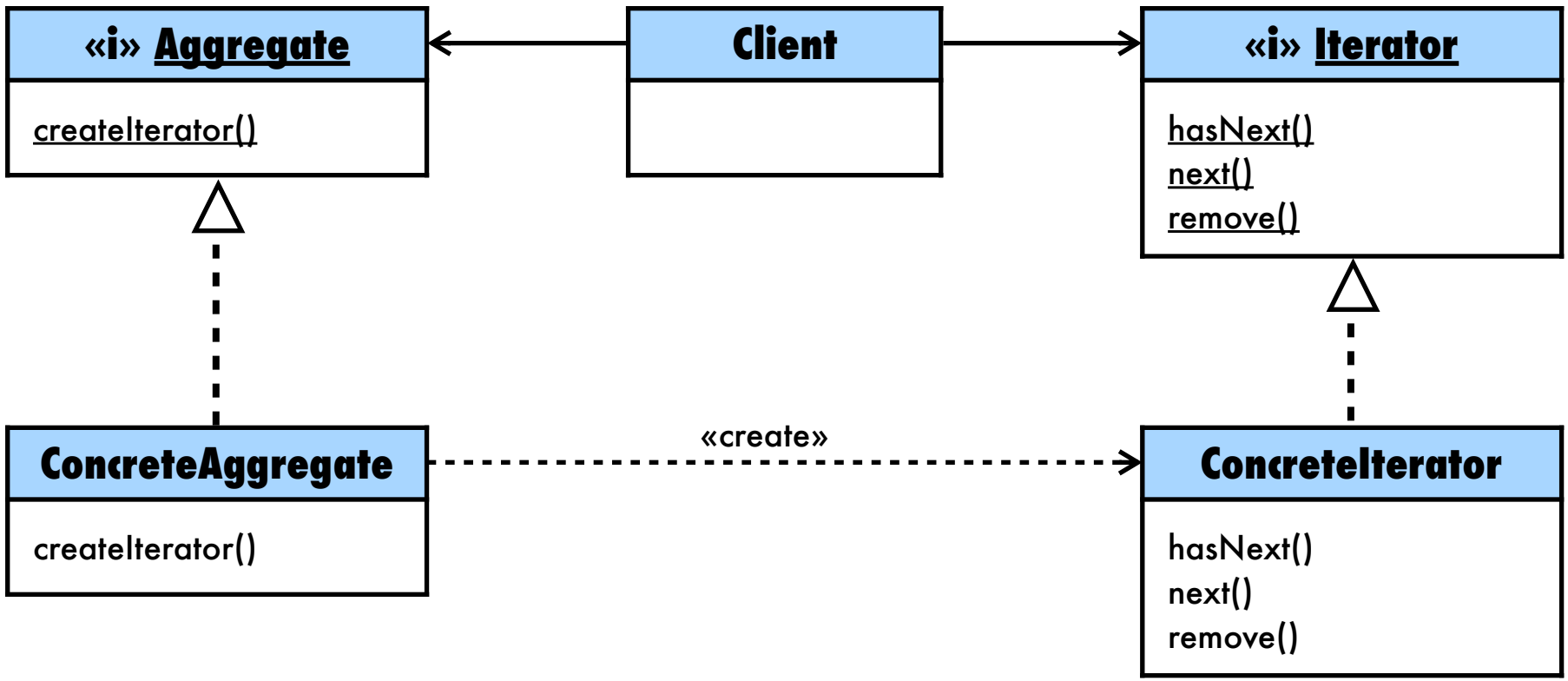
The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



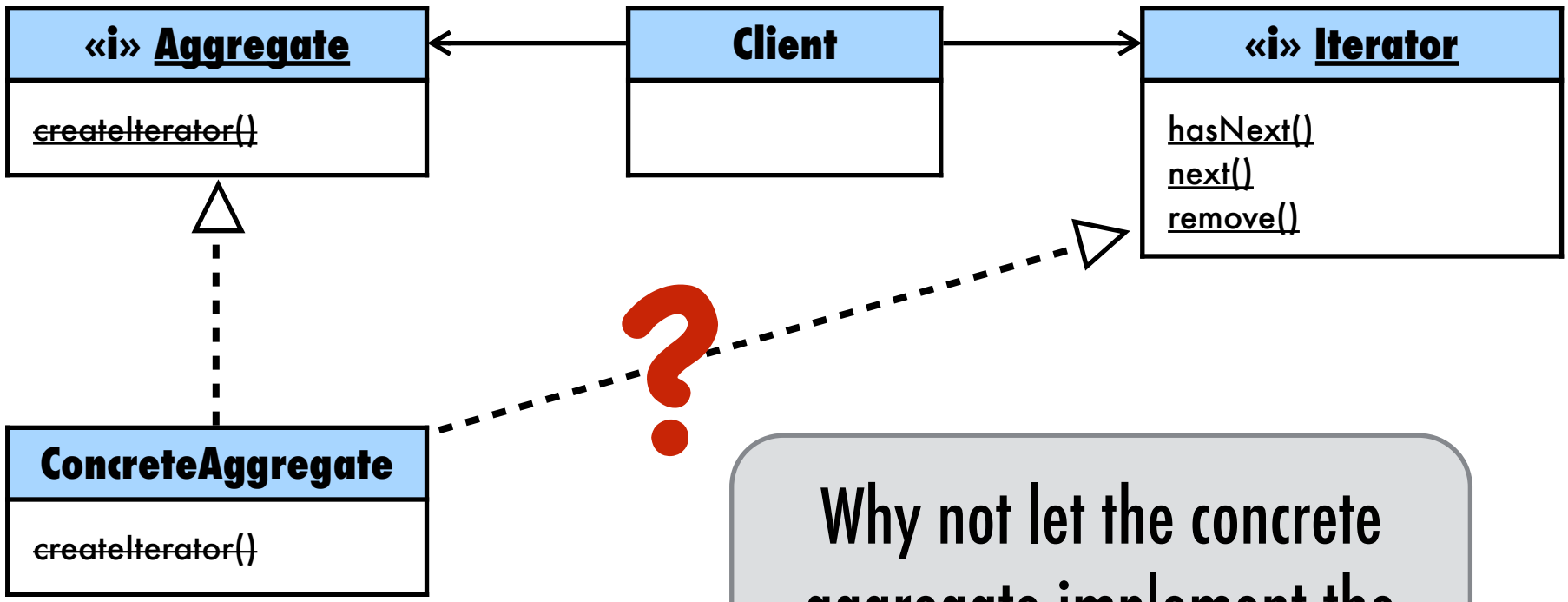
Aggregate Objects



Class diagram for iterator

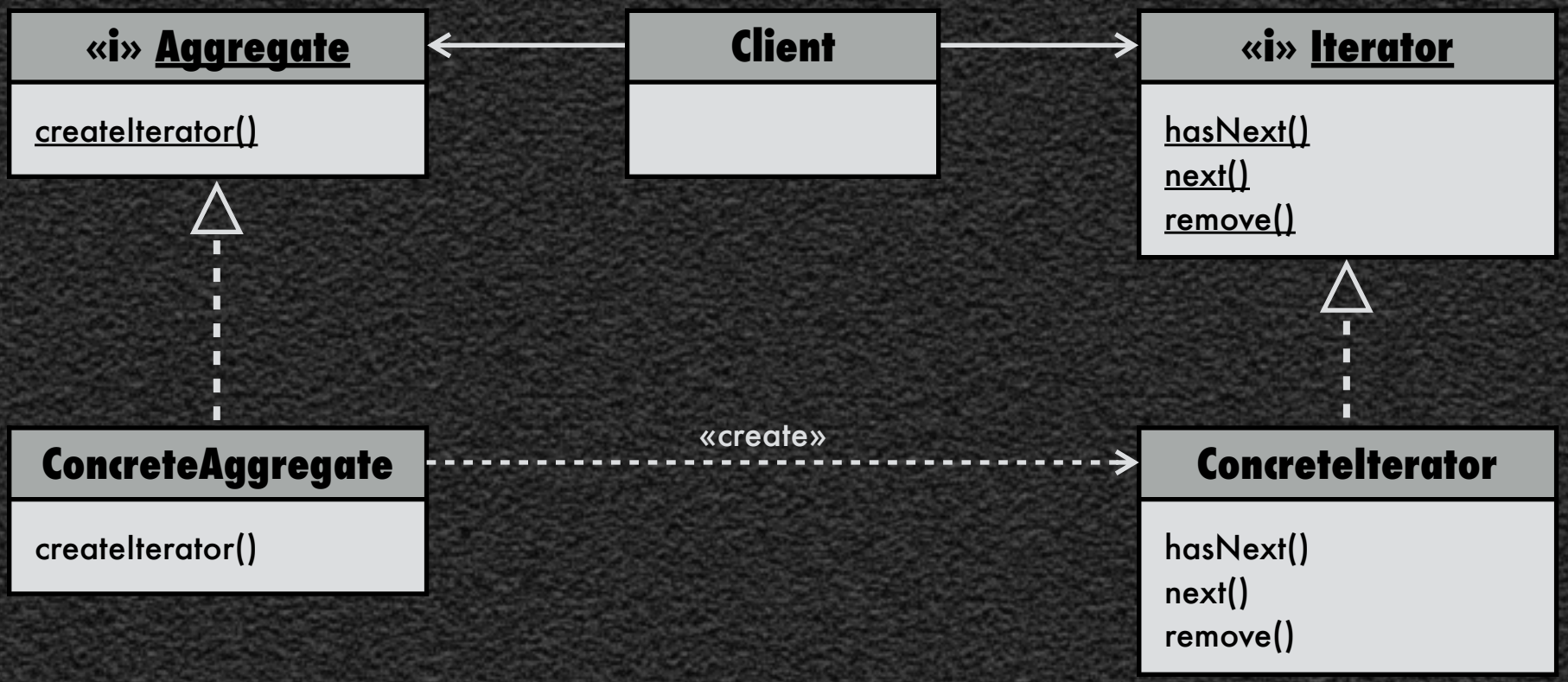


Class diagram for iterator



Pause and Think

Do we really need ConcreteIterator in this design?



Design Principle

A class should have only one reason to change

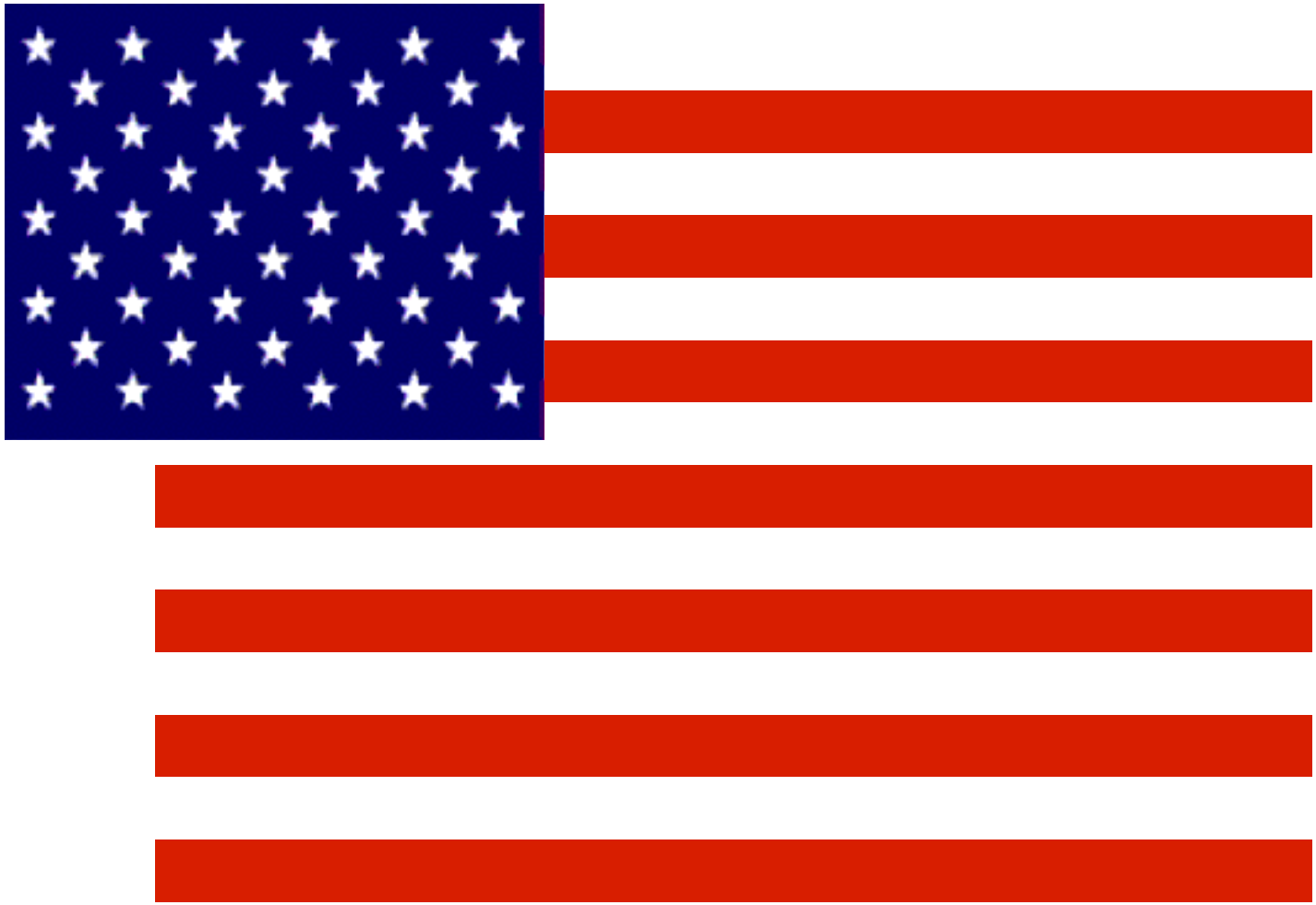
Cohesion

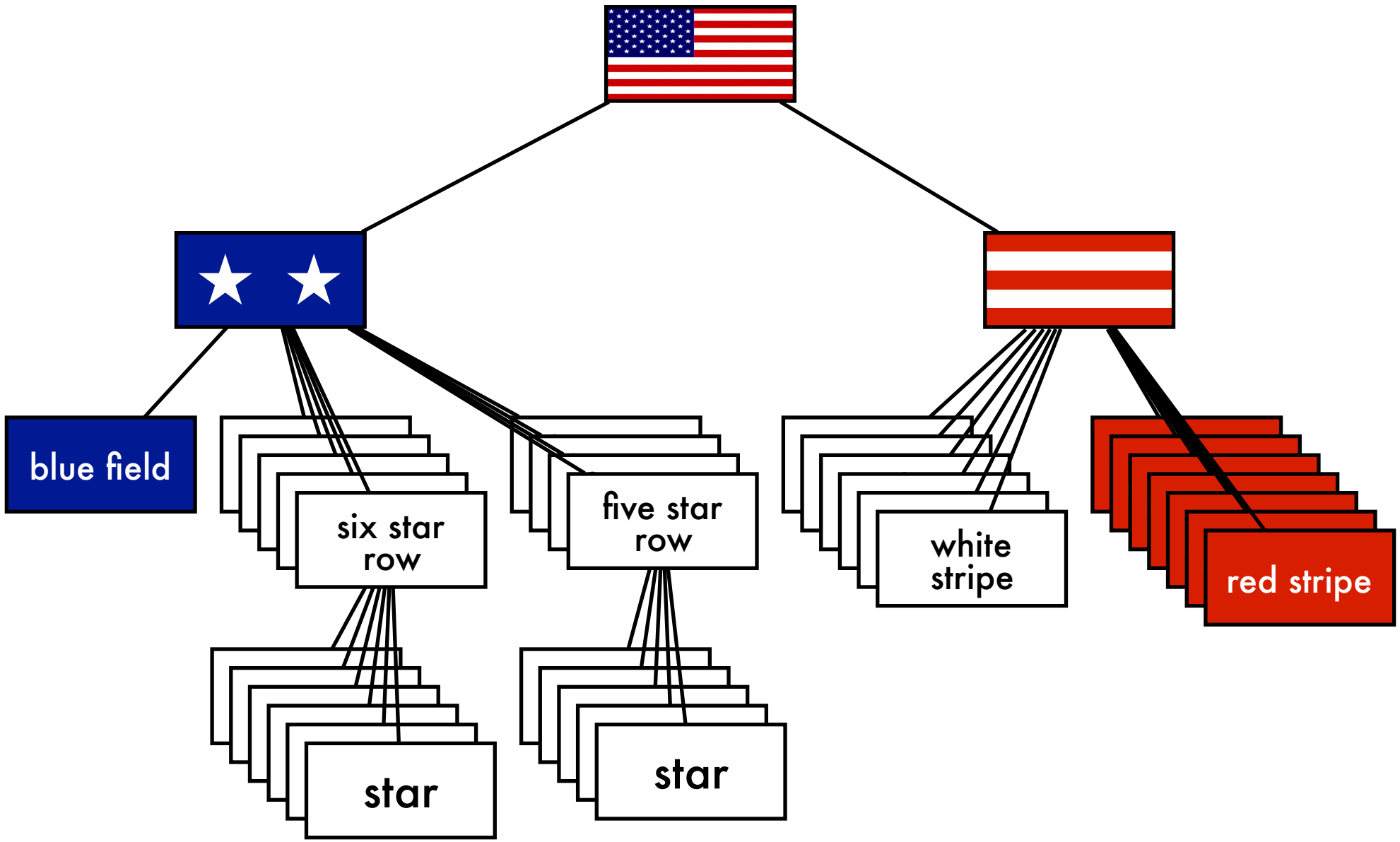
- A class has high cohesion when it is designed around a set of related functions.
- Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.

Composite Pattern

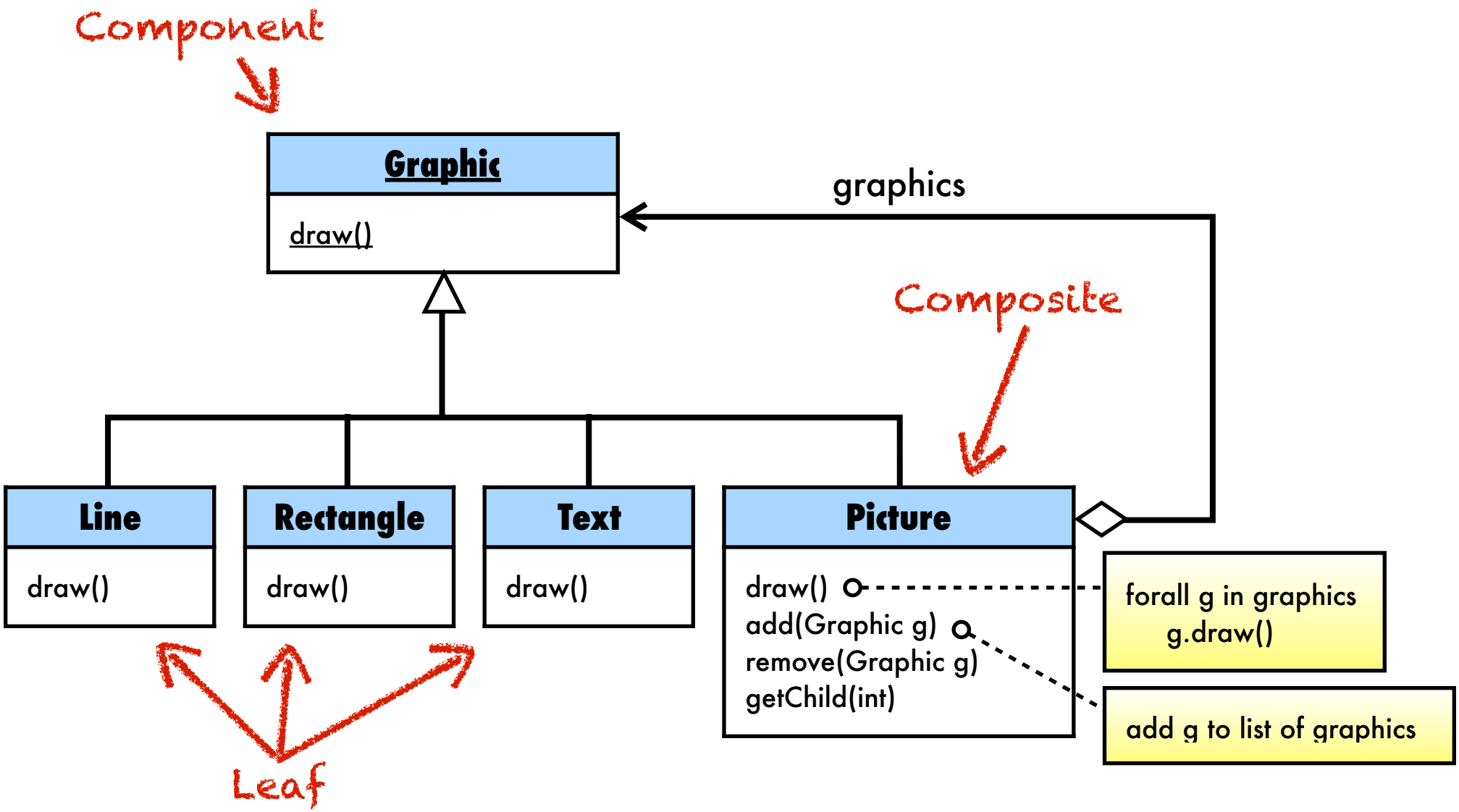
The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Drawing a flag in PowerPoint




blue field
six star row
five star row
white stripe
red stripe
star
star

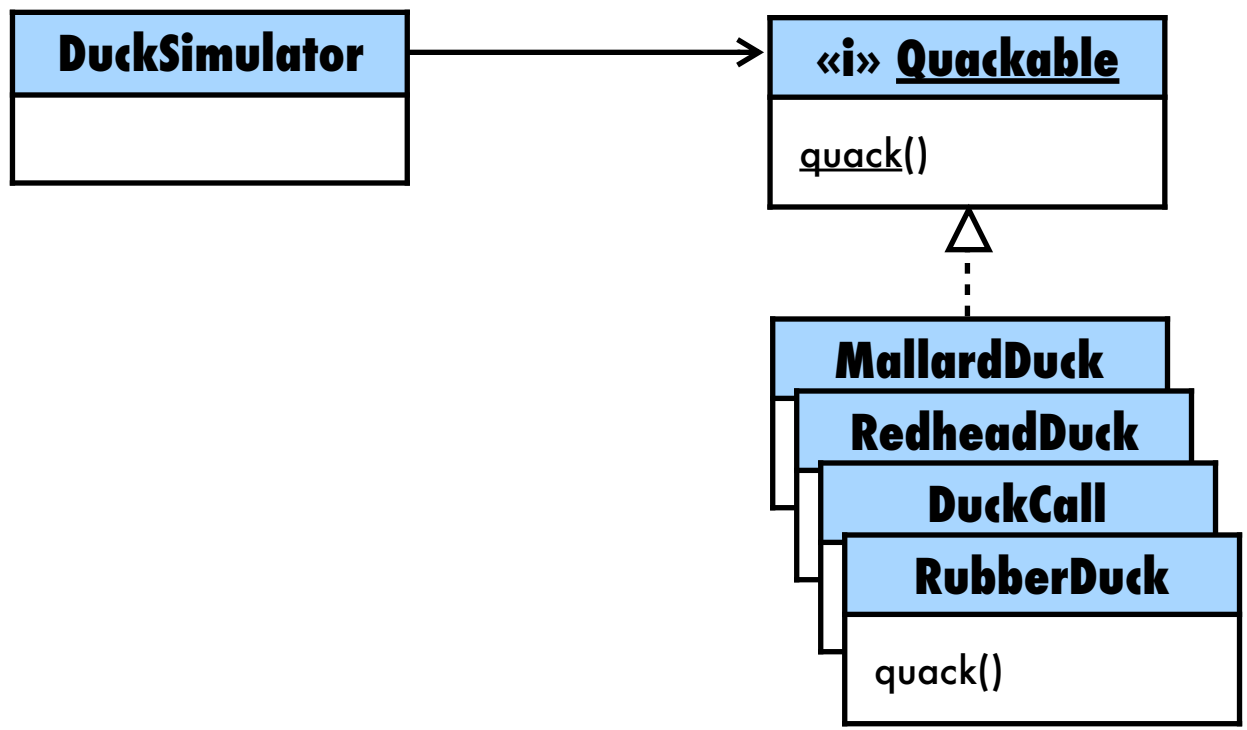
Class diagram for composite



Chapter 12

Patterns of patterns (duck sim example)

Simple duck simulator



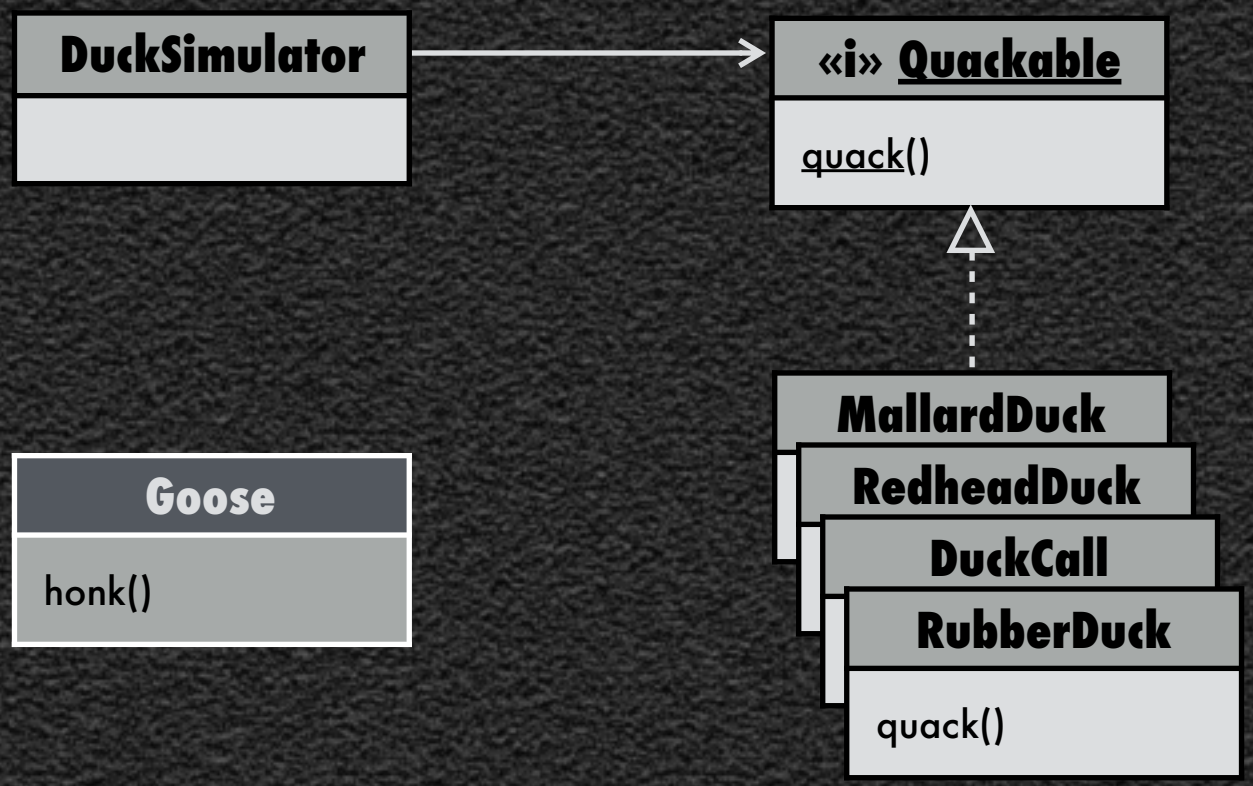
There's just one thing...

We would like to add geese to the simulation and treat them just like ducks.

Professional quackologist

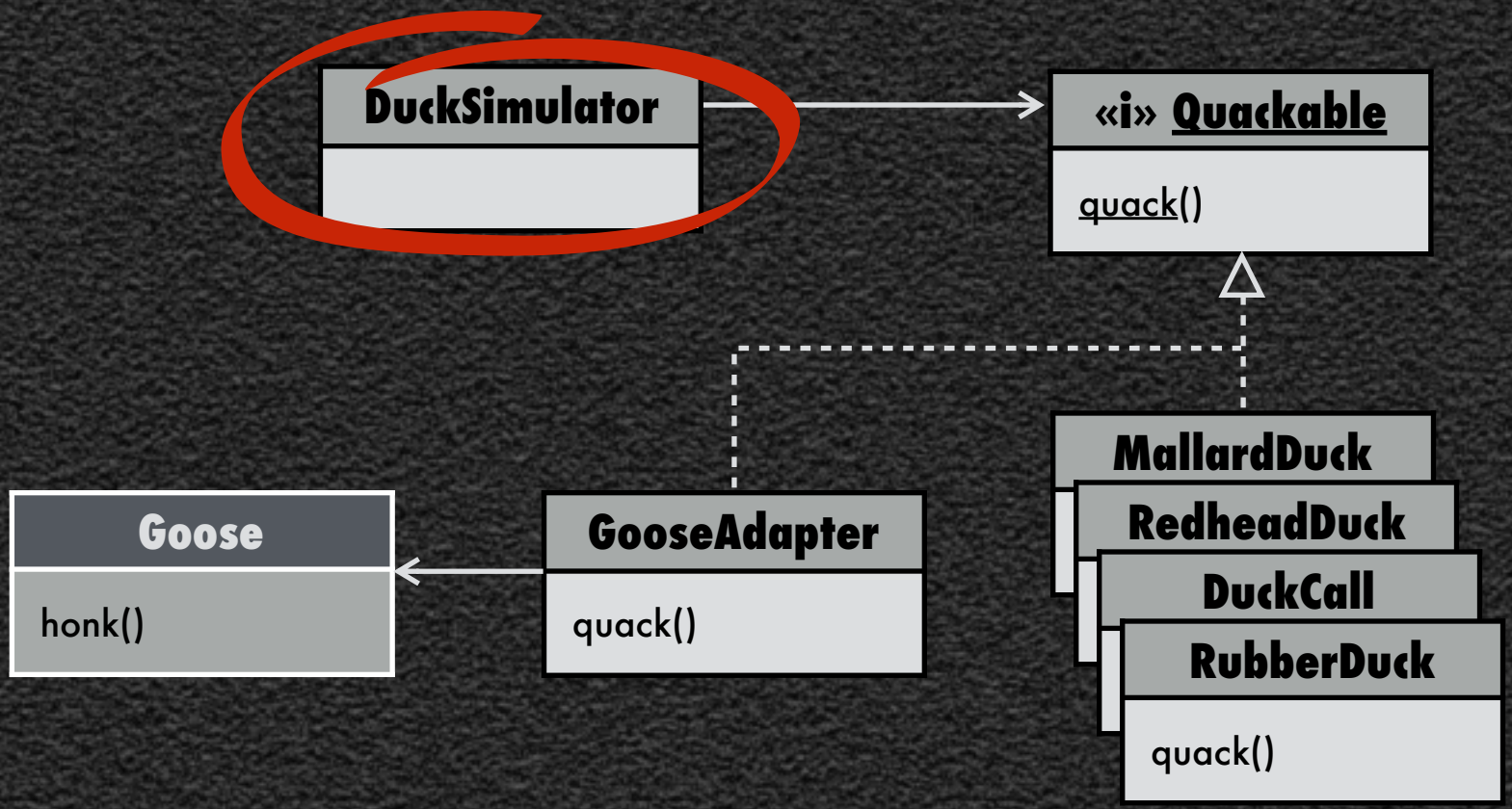
Pause and Think

How can we treat geese like ducks in this design?

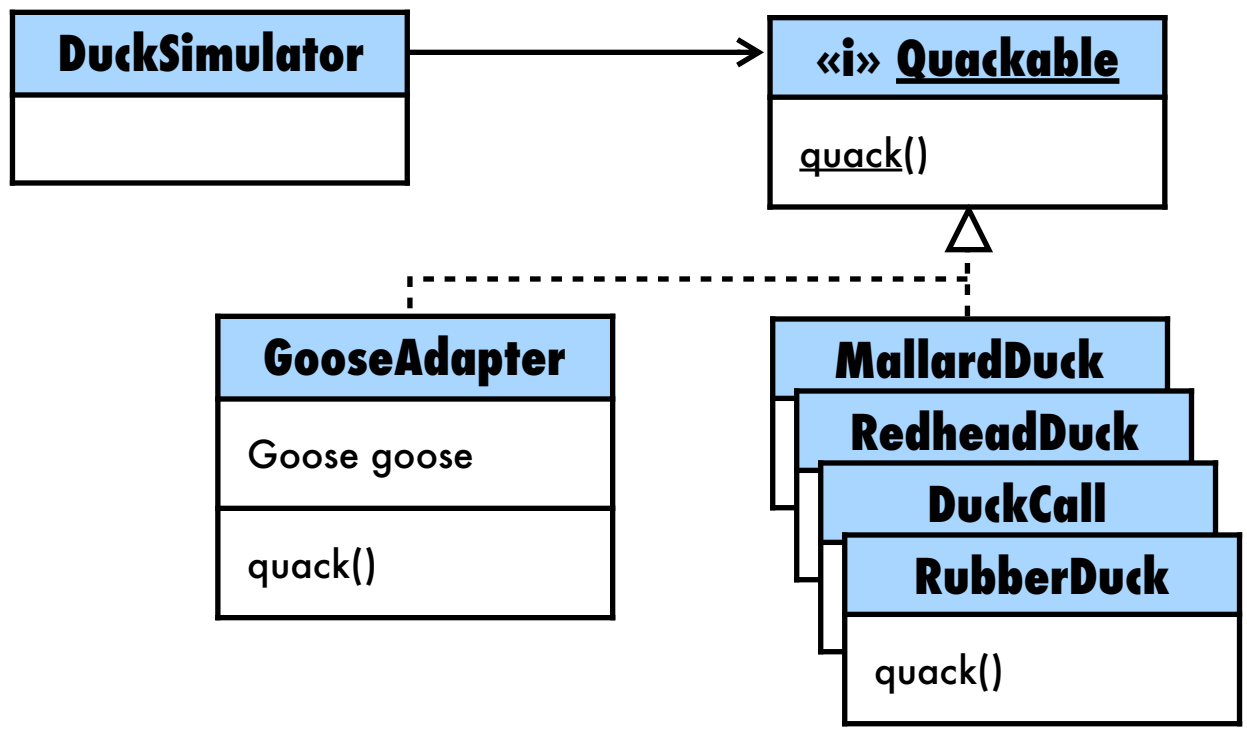


Pause and Think

How can we treat geese like ducks in this design?



Duck simulator with goose adapter



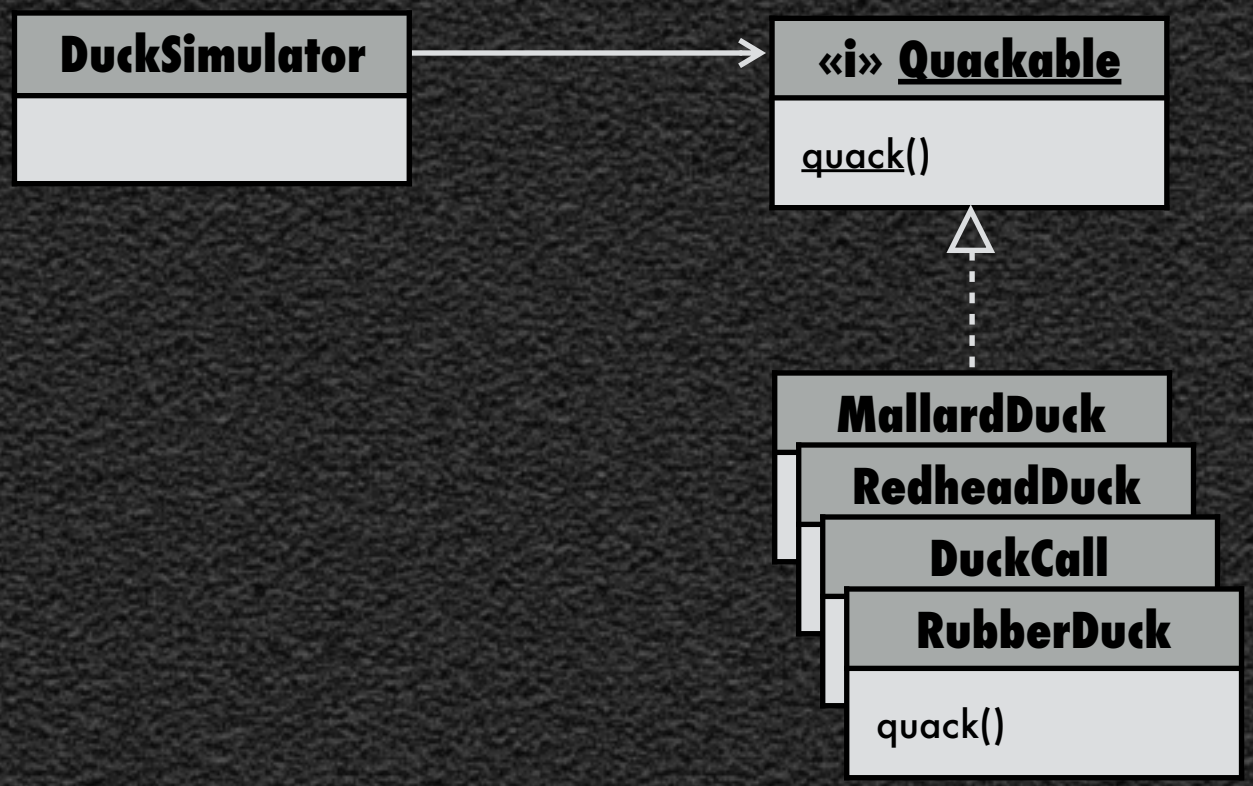
Well, two things…

What we really need to be able to do is count the number of quacks in the simulation from the individual ducks.

Professional quackologist

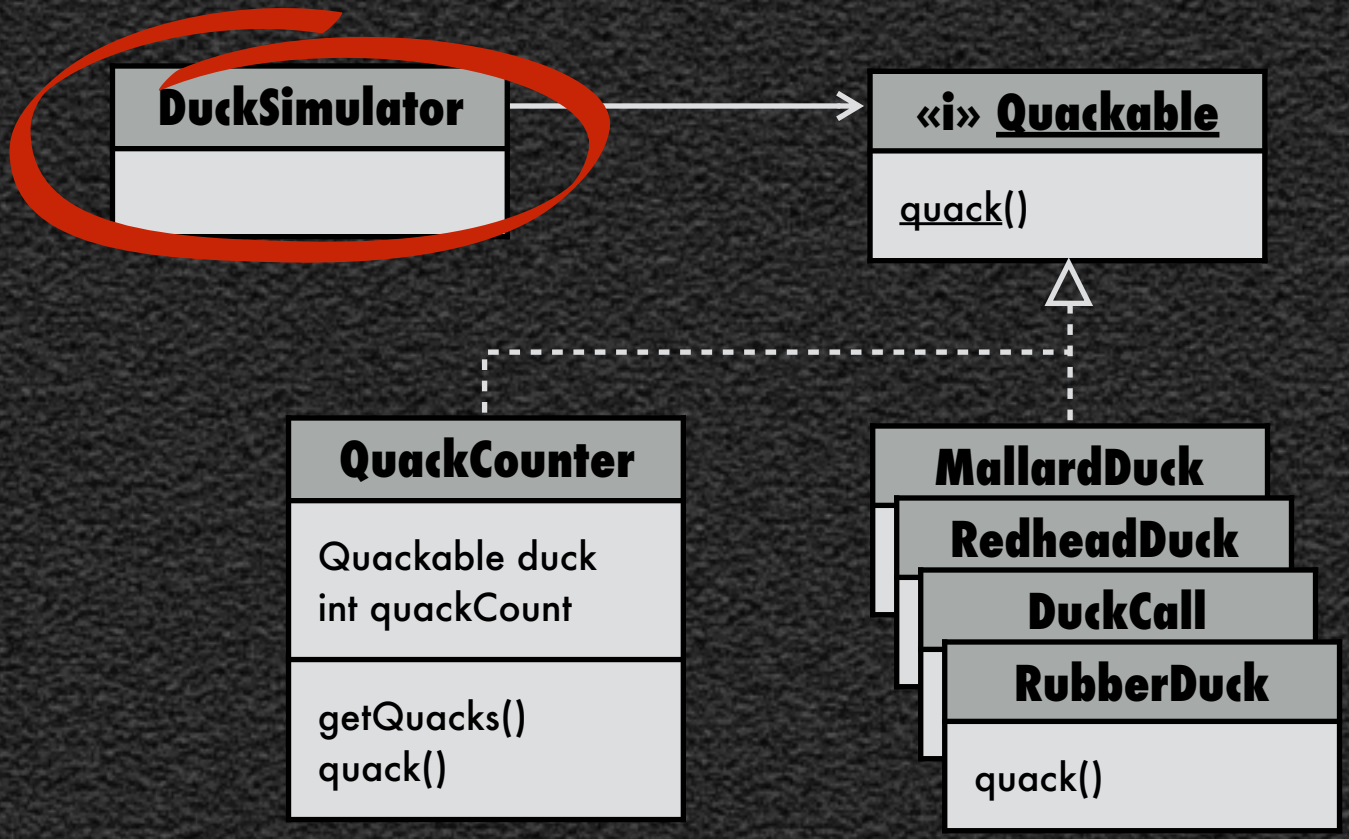
Pause and Think

How can we modify this to count the number of quacks?

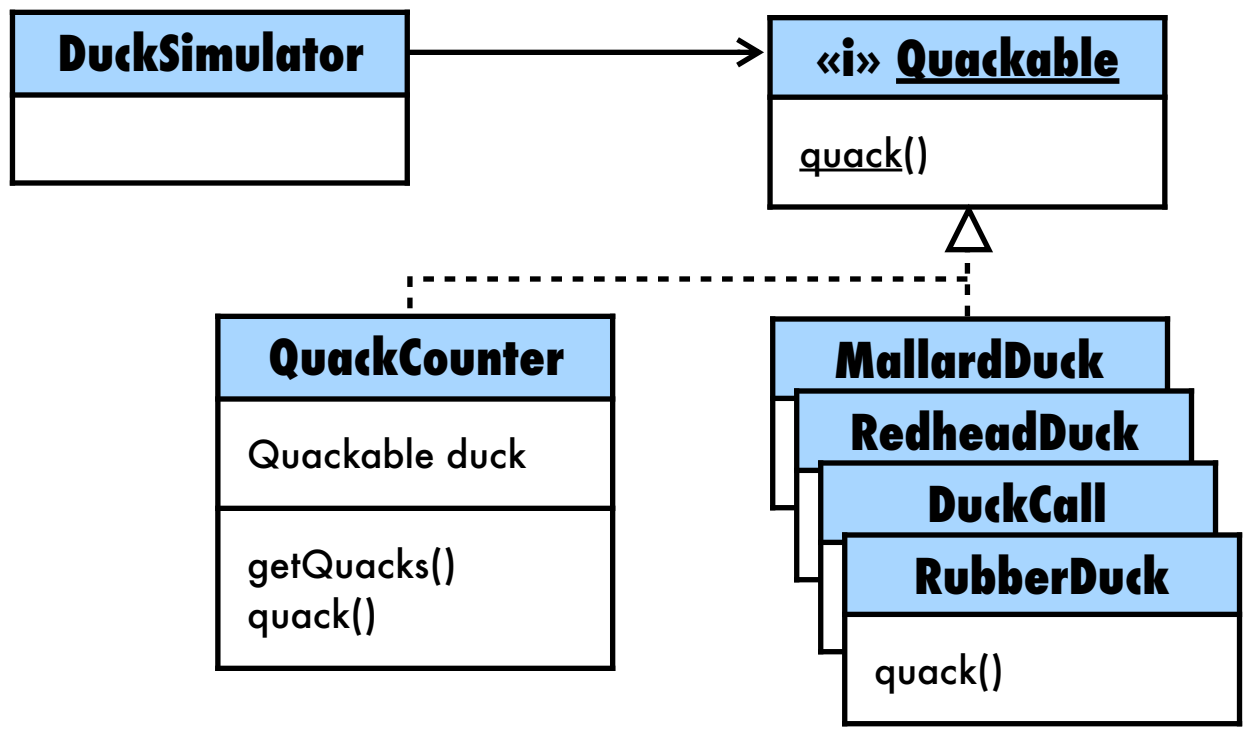


Pause and Think

How can we modify this to count the number of quacks?



Duck simulator with quack counting decorator



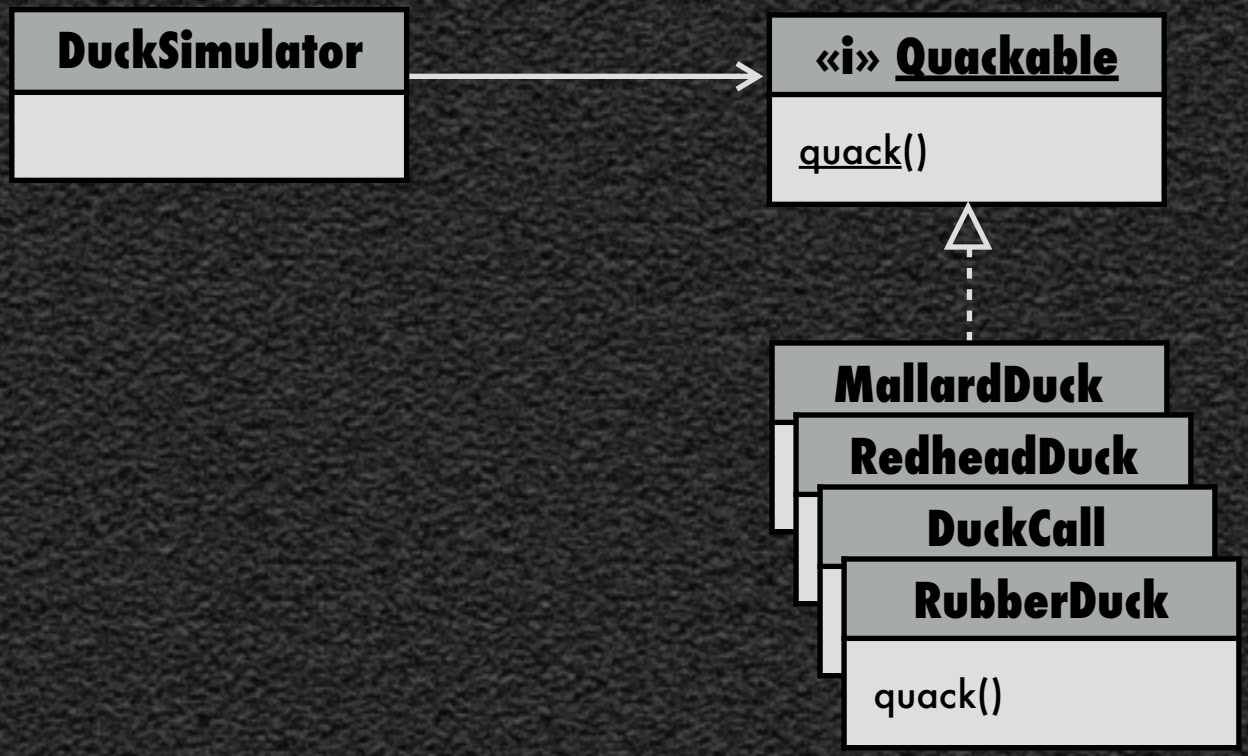
And while you're at it...

We want to make sure that each time a new duck is created, it has the quack-counting ability.

Professional quackologist

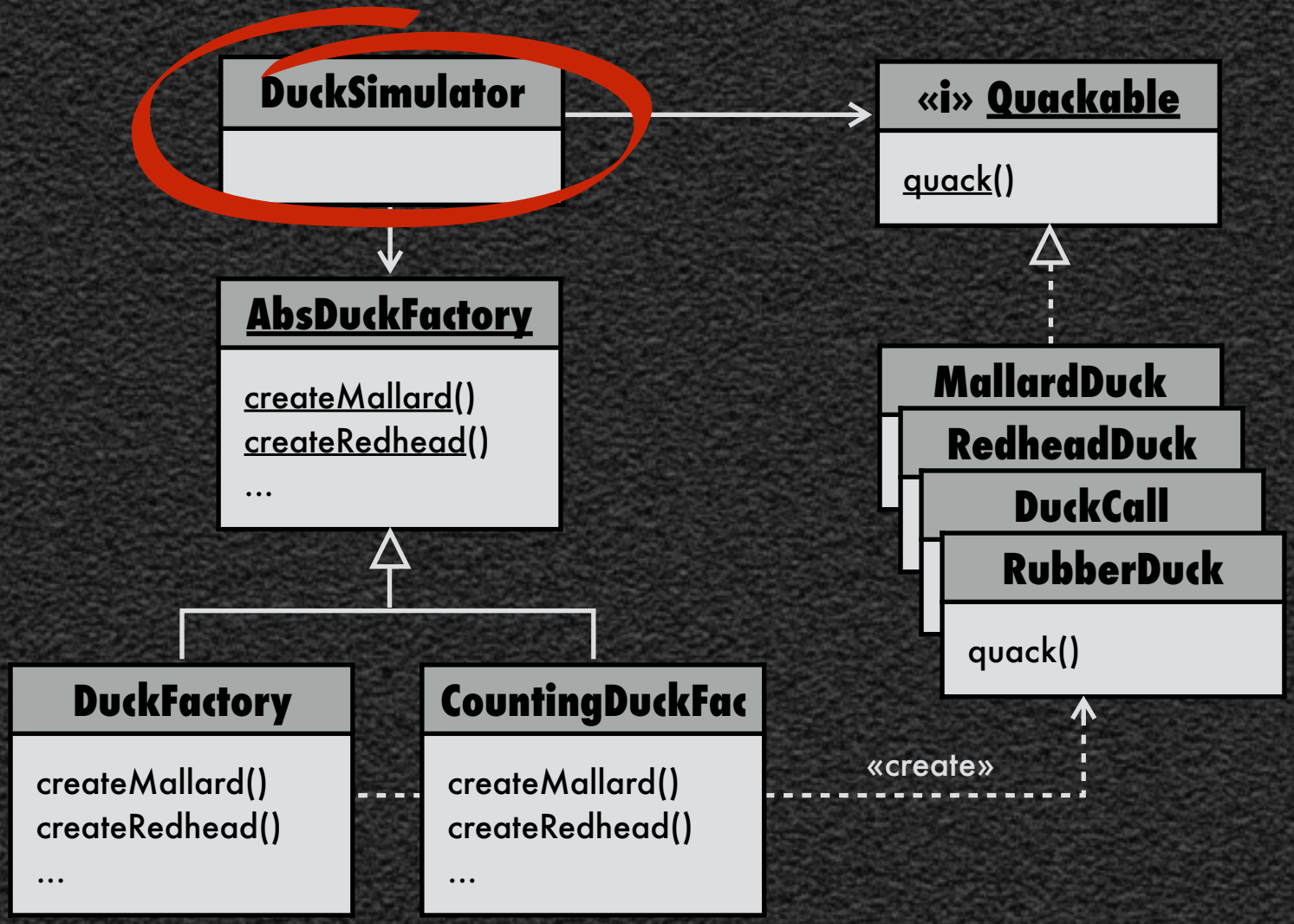
Pause and Think

How do we give all ducks the quack counting ability?



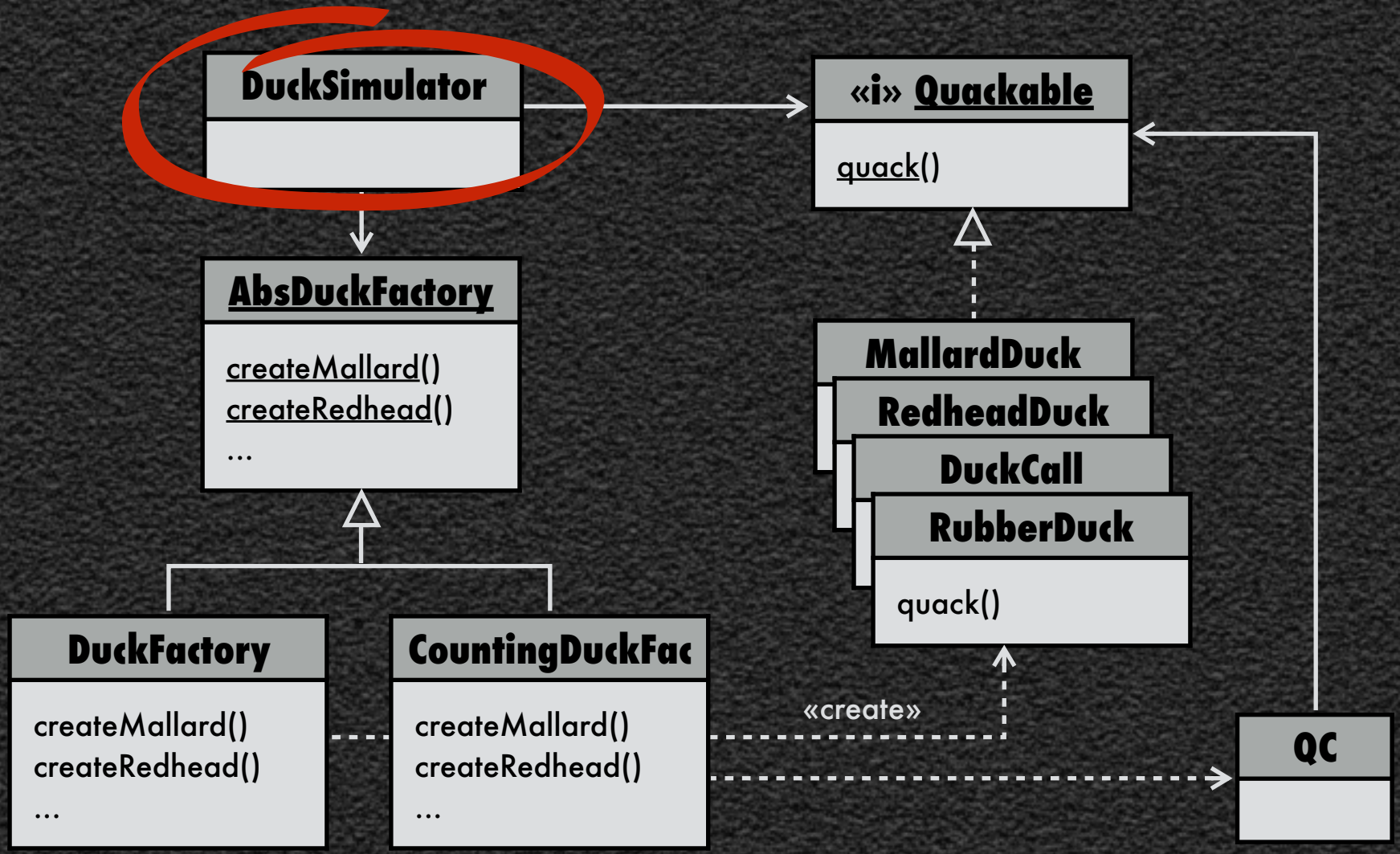
Pause and Think

How do we give all ducks the quack counting ability?

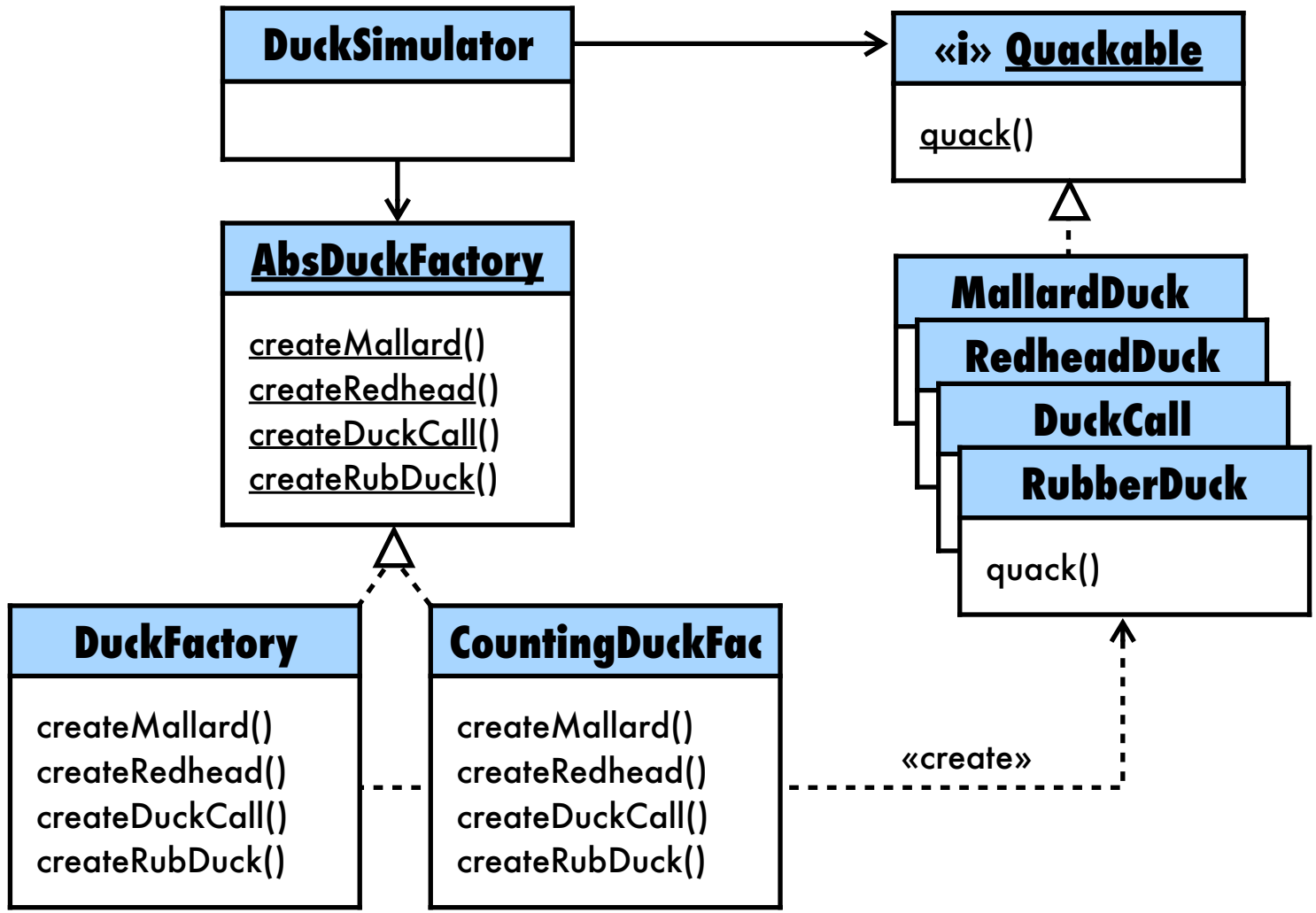


Pause and Think

How do we give all ducks the quack counting ability?



Duck simulator with duck and counting factories



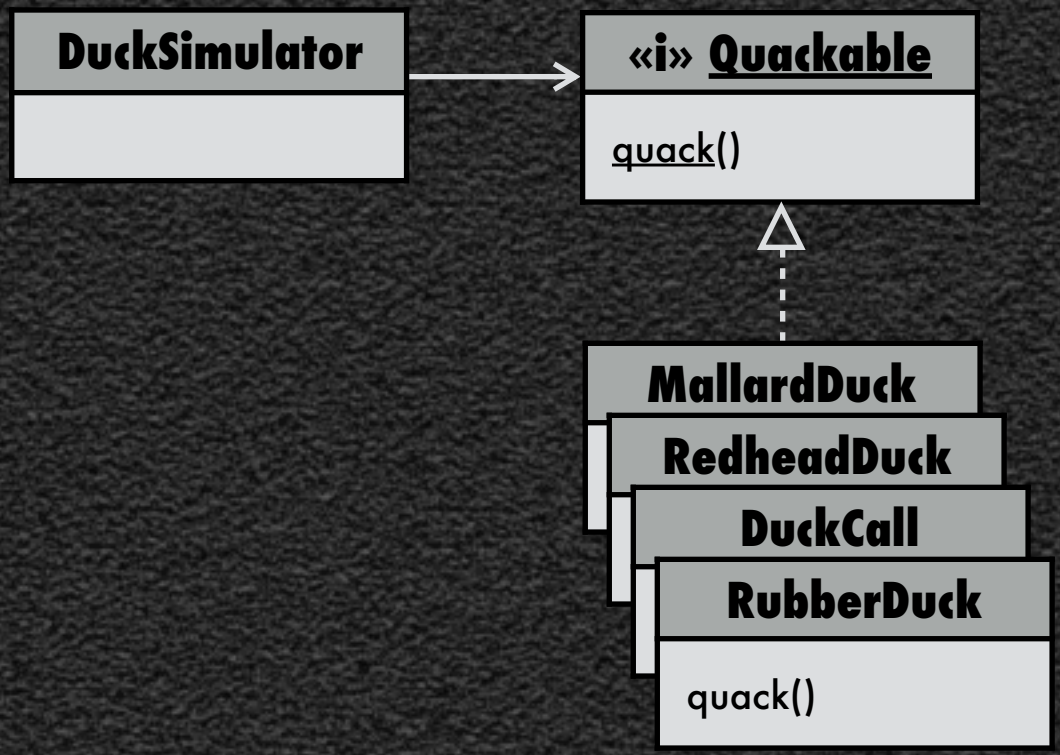
Now that I think of it…

We would also like to be able to look at quacking behavior for flocks and families of ducks.

Professional quackologist

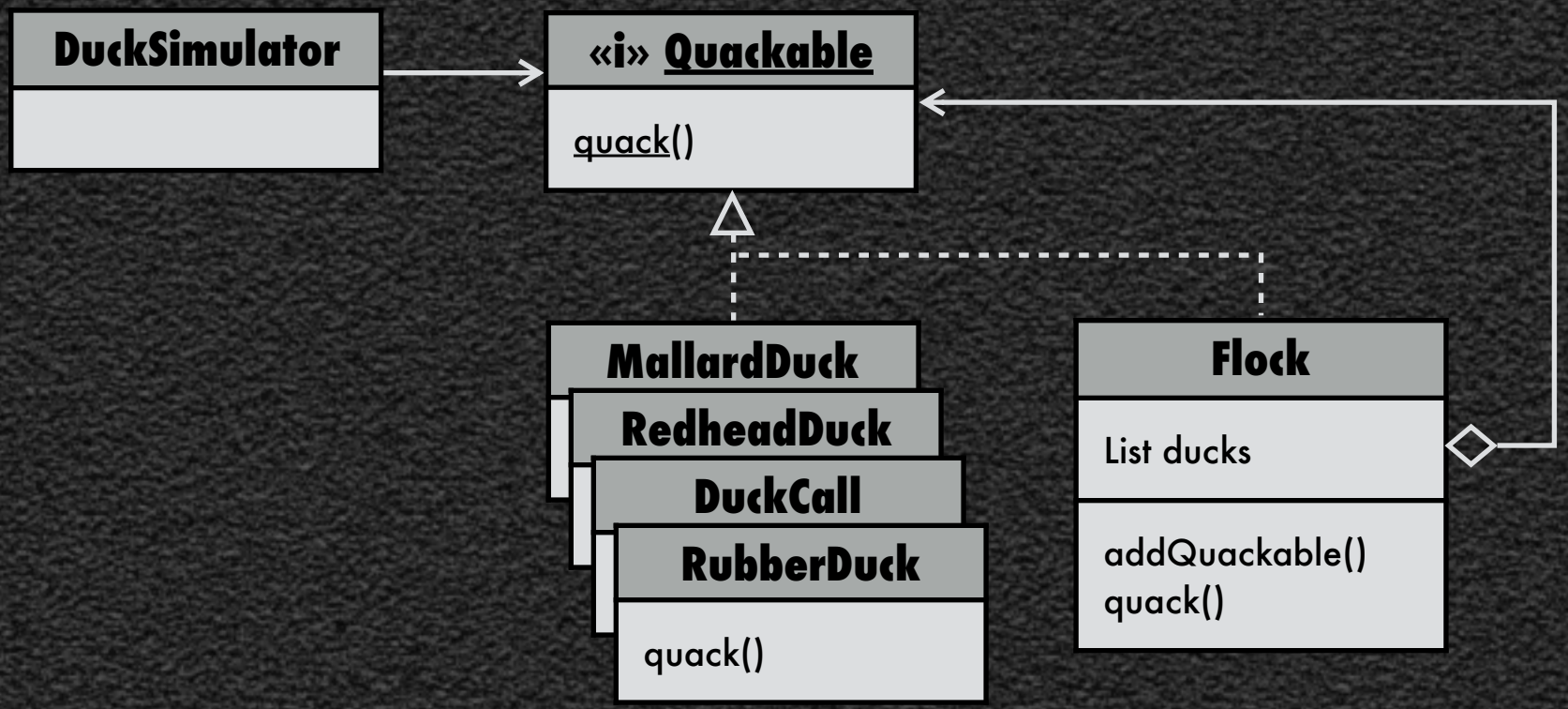
Pause and Think

How can we look at the quacks of flocks of ducks?



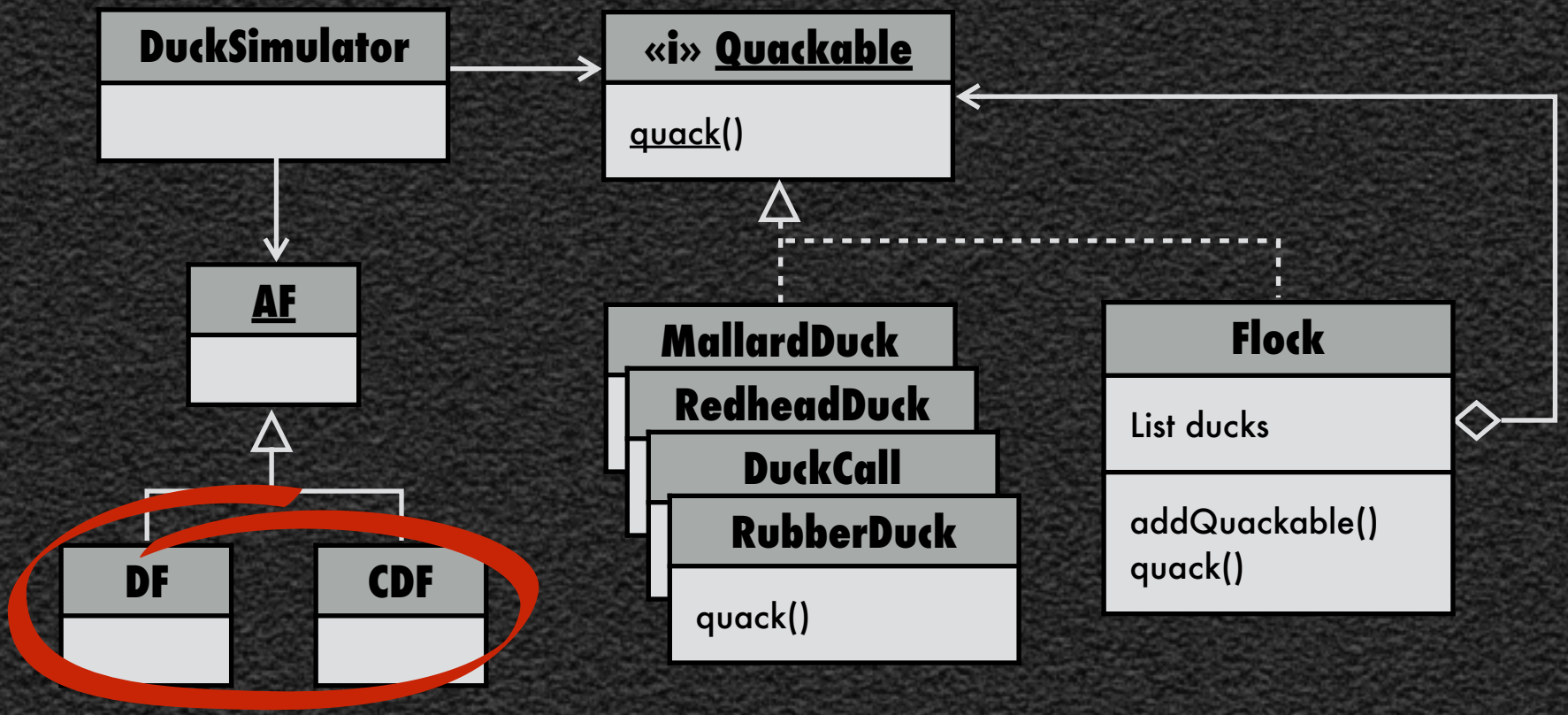
Pause and Think

How can we look at the quacks of flocks of ducks?

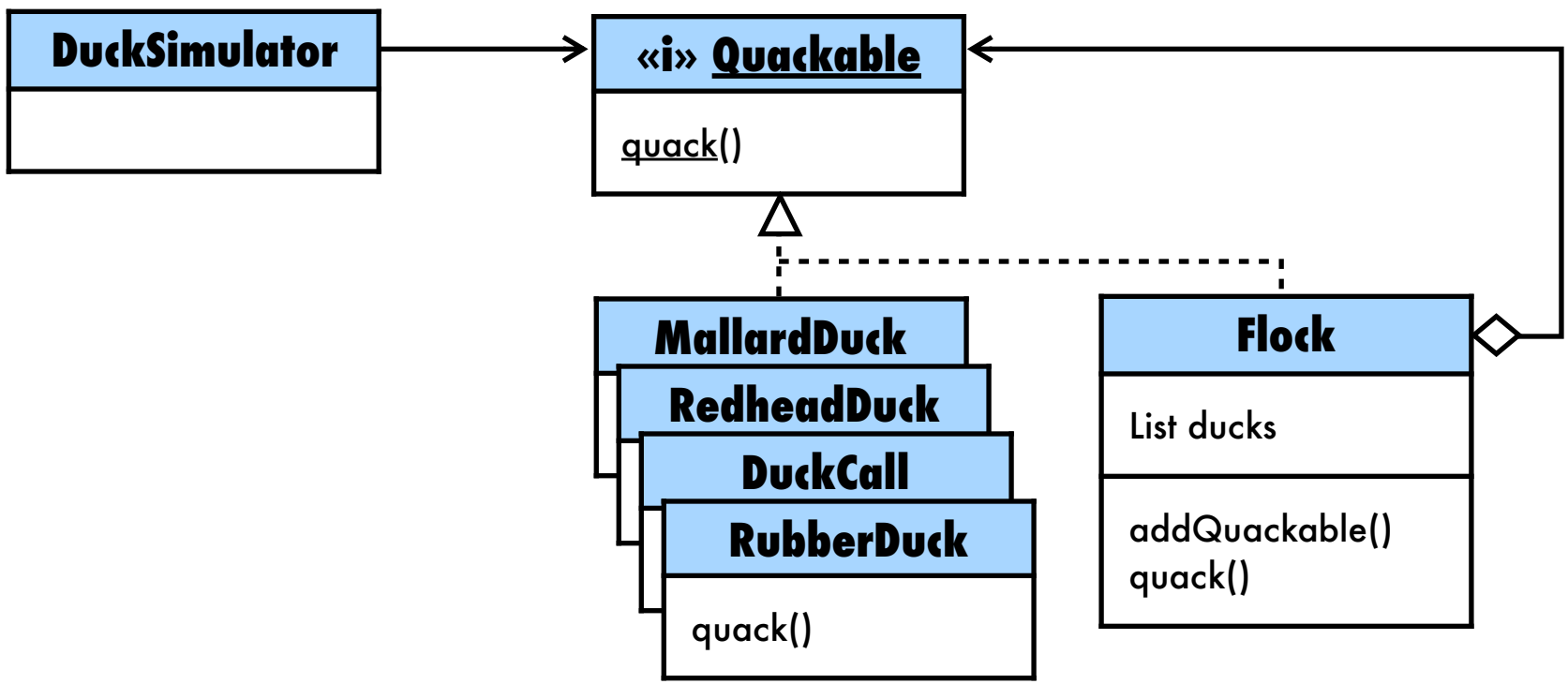


Pause and Think

How can we look at the quacks of flocks of ducks?



Duck simulator with flock composite



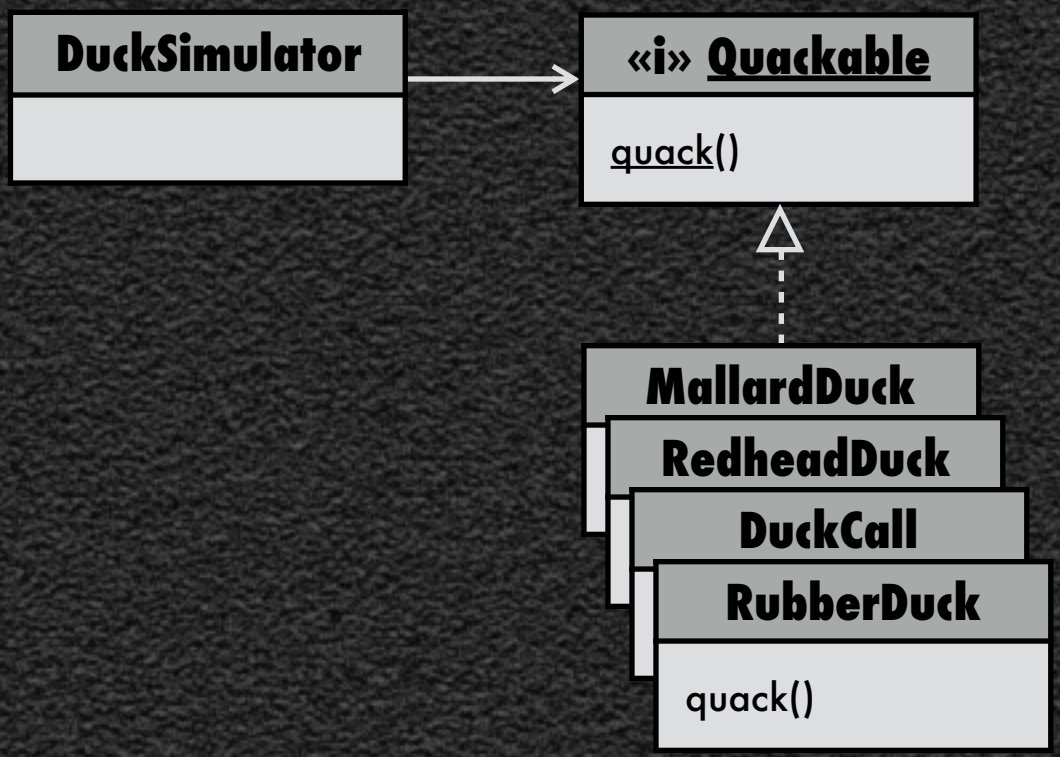
And I almost forgot…

Finally, we would like to be notified when any kind of duck, goose, or flock of ducks quack.

Professional quackologist

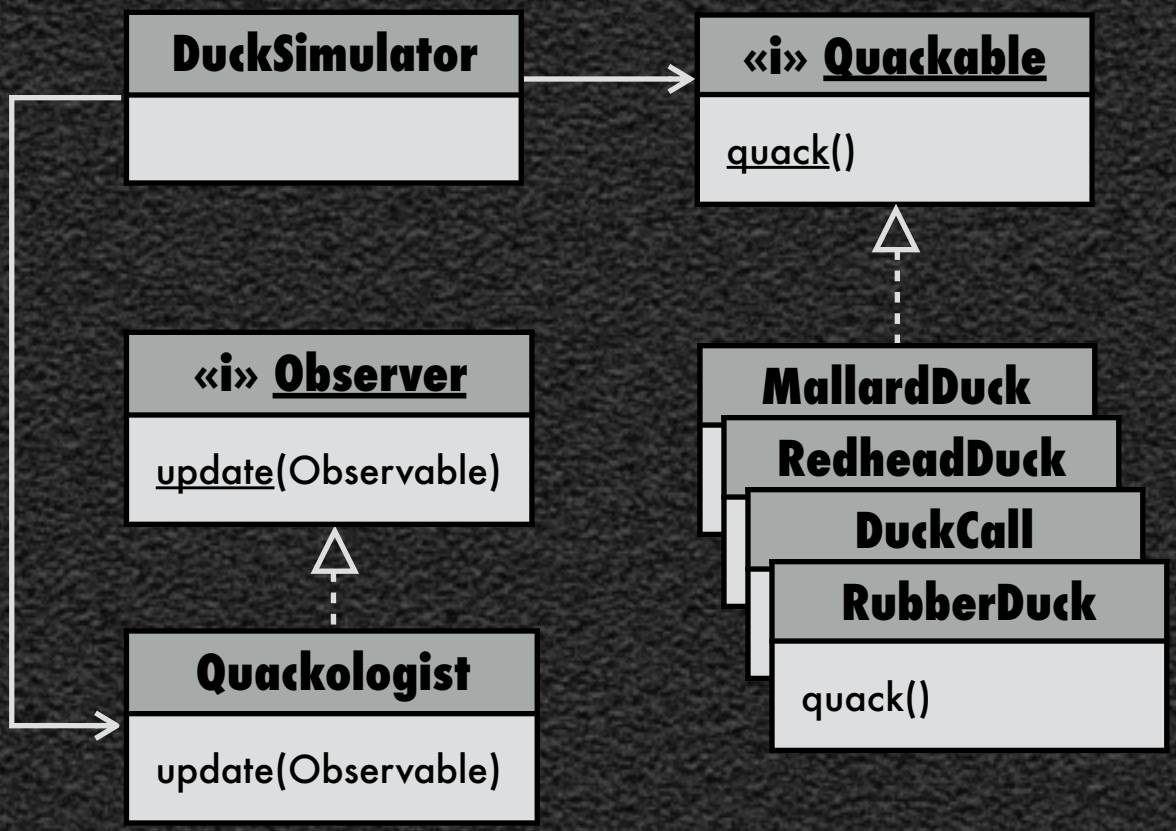
Pause and Think

How can we notify others when a duck quacks?



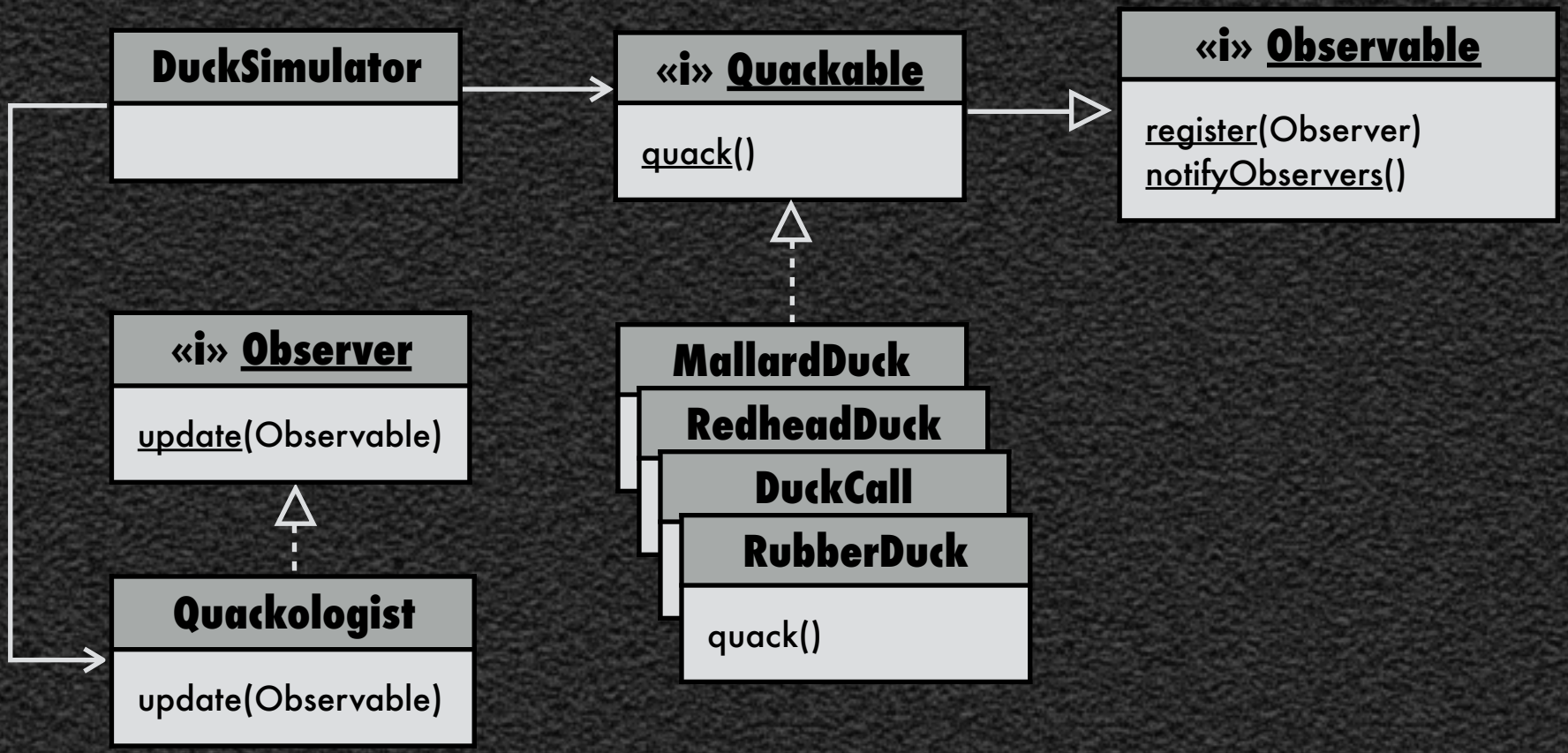
Pause and Think

How can we notify others when a duck quacks?



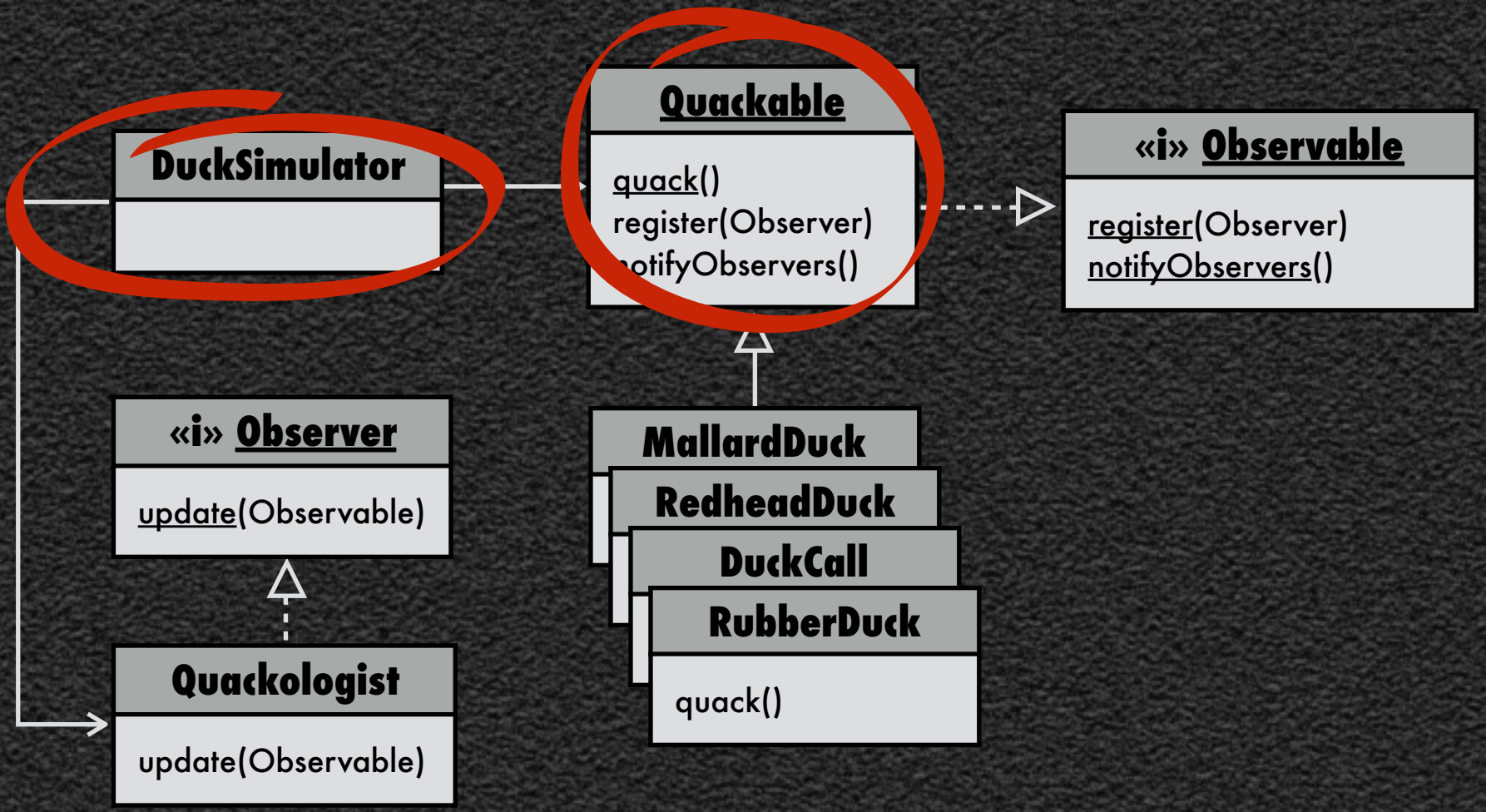
Pause and Think

How can we notify others when a duck quacks?

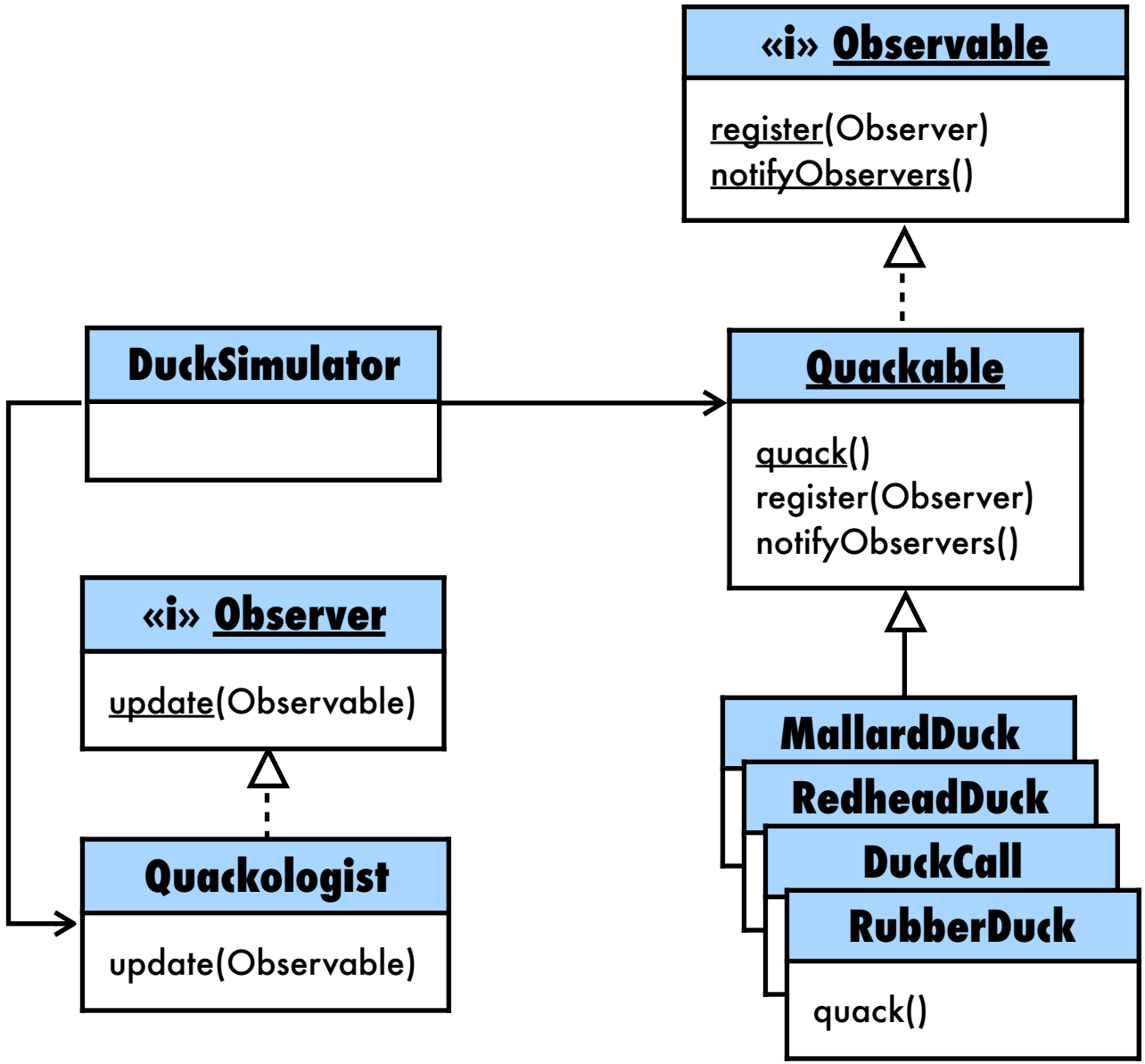


Pause and Think

How can we notify others when a duck quacks?

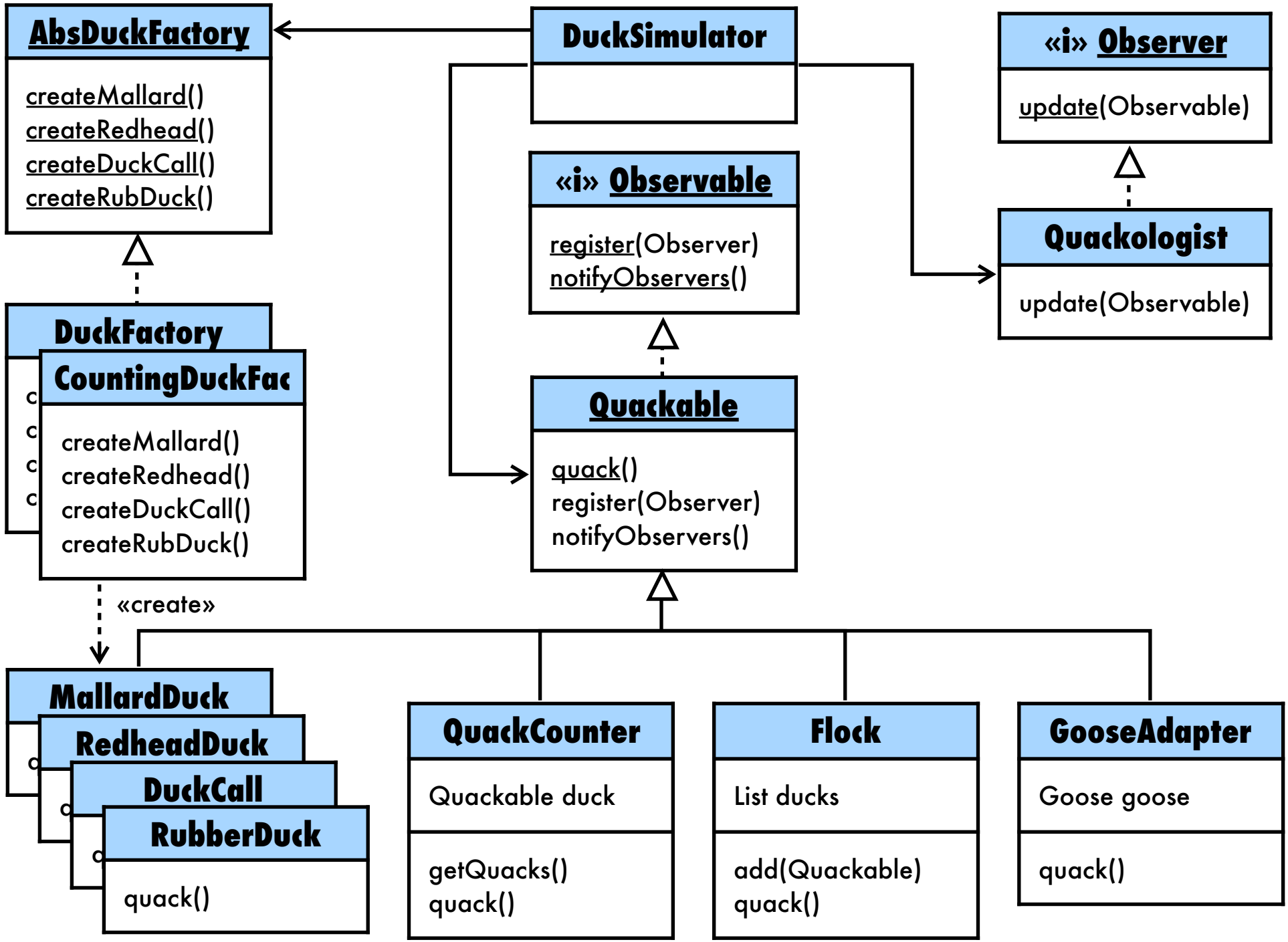


DuckSim with quack observer



Growing the duck simulator

- We want to treat geese as quackables.
- We want to count quacks.
- We want to add the quack counter to each new duck.
- We want to group quackables into flocks and families.
- We want to be notified when any quackable quacks.


AbsDuckFactory
createMallard()
createRedhead()
createDuckCall()
createRubDuck()
DuckSimulator
«i» Observer
update(Observable)
«i» Observable
register(Observer)
notifyObservers()
Quackologist
update(Observable)
DuckFactory
CountingDuckFac
createMallard()
createRedhead()
createDuckCall()
createRubDuck()
Quackable
quack()
register(Observer)
notifyObservers()
«create»
MallardDuck
RedheadDuck
DuckCall
RubberDuck
quack()
QuackCounter
Quackable duck
getQuacks()
quack()
Flock
List ducks
add(Quackable)
quack()
GooseAdapter
Goose goose
quack()

Chapter 12

Patterns of patterns (model-view-controller)

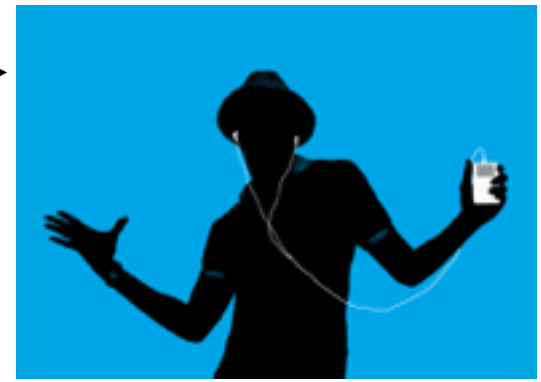
Model - View - Controller

The model holds all data, state and application logic. The model is oblivious to View and Controller. It just provides an interface to manipulate and retrieve its state, and it can send notifications of state changes to observers.

The view gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.

The controller takes user input and figures out what it means to the model.

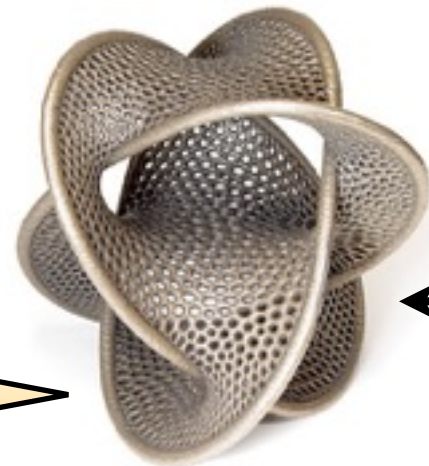
iPod listener



You use the interface; your actions go to the controller

"Play new song"

controller asks model to play song



Model contains all the state, data, and application logic needed to maintain and play mp3s.

model tells view the state has changed

song display updates and new song plays

A closer look...

1. You're the user – you interact with the view.
   The view is your window to the model. When you do something to the view (like click the Play button) then the view tells the controller what you did. It's the controller's job to handle that.

2. The controller asks the model to change its state.
   The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.

3. The controller may also ask the view to change.
   When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

4 The model notifies the view when its state has changed.
When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.

5 The view asks the model for state.
The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. the view might also ask the model for state as the result of the controller requesting some change in the view.

MVC through pattern-colored glasses

- Model uses observer pattern to keep views and controllers updated on the latest state changes.
- View and controller implement strategy pattern. Controller is the behavior view, and it can be easily exchanged with another controller if you want different behavior.
- View typically uses composite pattern internally to manage the windows, buttons and other components of the display.

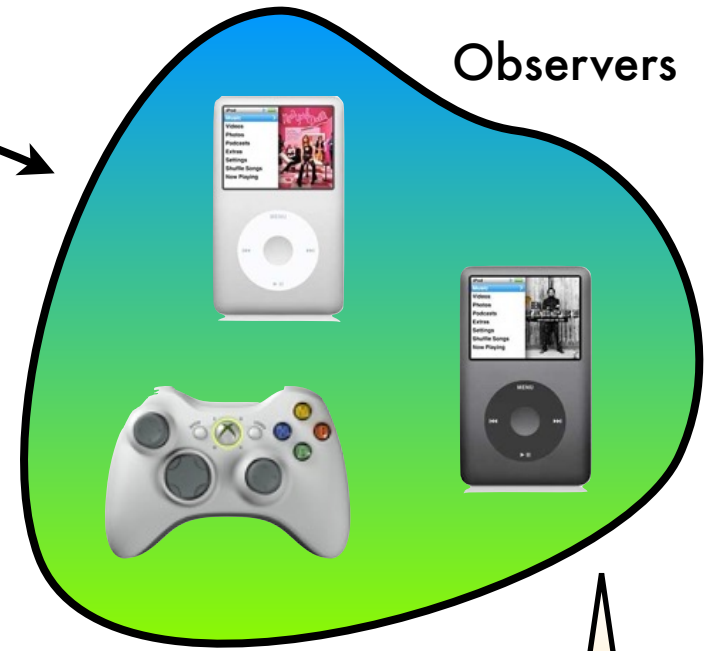
Observer

The model has no dependencies on viewers or controllers!

Observable

My state has changed!

Observers



I'd like to register as an observer

Any object that's interested in state changes in the model registers with the model as an observer.

All these observers will be notified whenever state changes in the model.

Strategy

The view only worries about presentation, the controller worries about translating user input to actions on the model.

The user did something

Controller is the strategy for View. It knows how to handle user actions.

View delegates to the controller to handle the user actions.

We can swap in another behavior for the view by changing the controller.

Composite

The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components and so on until you get to the leaf nodes.

paint()