Patterns of Patterns

Working together

One of the best ways to use patterns is to get them out of the house so they can interact with other patterns. The more you use patterns the more you’re going to see them showing up together in your designs. We have a special name for a set of patterns that work together in a design that can be applied over many problems: a compound pattern. That’s right, we are now talking about patterns made of patterns!

- Patterns are often used together and combined within the same design solution.
- A compound pattern combines two or more patterns into a solution that solves a recurring or general problem.

Duck reunion

First, we'll create a Quackable interface.
Like we said, we're starting from scratch. This time around, the Ducks are going to implement a Quackable interface. That way we'll know what things in the simulator can quack() – like Mallard Ducks, Redhead Ducks, Duck Calls, and we might even see the Rubber Duck sneak back in.

```
public interface Quackable {
    public void quack();
}
```

Quackables only need to do one thing well: Quack!

Continue

Now, some Ducks that implement Quackable

What good is an interface without some classes to implement it? Time to create some concrete ducks.

```
public class MallardDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}
```

Your standard Mallard duck.

```
public class RedheadDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}
```

We've got to have some variation of species if we want this to be an interesting simulator.

Add other ducks

```
public class DuckCall implements Quackable {
    public void quack() {
        System.out.println("Kwak");
    }
}
```

A DuckCall that quacks but doesn't sound quite like the real thing.

```
public class RubberDuck implements Quackable {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

A RubberDuck that makes a squeak when it quacks.

Simulator

Here's our main method to get everything going.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();

        System.out.println("\nDuck Simulator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

We create a simulator and then call its simulate() method.

We need some ducks, so here we create one of each Quackable...

... then we simulate each one.

Here we overload the simulate method to simulate just one duck.

Here we let polymorphism do its magic: no matter what kind of Quackable gets passed in, the simulate() method asks it to quack.

a Goose class that has been hanging around the simulator

```
public class Goose {
    public void honk() {
        System.out.println("Honk");
    }
}
```

A Goose is a honker, not a quacker.

Let's say we wanted to be able to use a Goose anywhere we'd want to use a Duck. After all, geese make noise; geese fly; geese swim. Why can't we have Geese in the simulator?
What pattern would allow Geese to easily intermingle with Ducks?

We need a goose adapter

Our simulator expects to see Quackable interfaces. Since geese aren't quackers (they're honkers), we can use an adapter to adapt a goose to a duck.

```
public class GooseAdapter implements Quackable {
    Goose goose;

    public GooseAdapter(Goose goose) {
        this.goose = goose;
    }

    public void quack() {
        goose.honk();
    }
}
```

Remember, an Adapter implements the target interface, which in this case is Quackable.

The constructor takes the goose we are going to adapt.

When quack is called, the call is delegated to the goose's honk() method.

Simulator with goose adapter

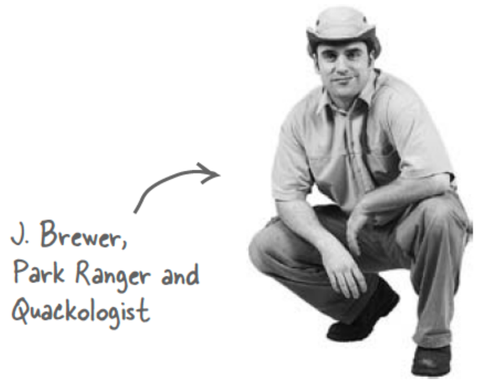
```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Goose Adapter");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

We make a Goose that acts like a Duck by wrapping the Goose in the GooseAdapter.

Once the Goose is wrapped, we can treat it just like other duck Quackables.

Quackology

- Quackologists are fascinated by all aspects of Quackable behavior. One thing Quackologists have always wanted to study is the total number of quacks made by a flock of ducks.
- How can we add the ability to count duck quacks without having to change the duck classes?
- Can you think of a pattern that would help?



Quack counts

Let's create a decorator that gives the ducks some new behavior (the behavior of counting) by wrapping them with a decorator object.

QuackCounter is a decorator

Like with Adapter, we need to implement the target interface.

We've got an instance variable to hold on to the quacker we're decorating.

And we're counting ALL quacks, so we'll use a static variable to keep track.

We get the reference to the Quackable we're decorating in the constructor.

When quack() is called, we delegate the call to the Quackable we're decorating...

... then we increase the number of quacks.

```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter (Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }
}
```

You have to decorate objects to get decorated behavior

- This quack counting is great. We're learning things we never knew about the little quackers. But we're finding that too many quacks aren't being counted.

- That's the problem with wrapping objects: you have to make sure they get wrapped or they don't get the decorated behavior.
- Why don't we take the creation of ducks and localize it in one place; in other words, let's take the duck creation and decorating and encapsulate it.
- What pattern does that sound like?

Factory

Okay, we need some quality control to make sure our ducks get wrapped. We're going to build an entire factory just to produce them. The factory should produce a family of products that consists of different types of ducks, so we're going to use the Abstract Factory Pattern.

We're defining an abstract factory that subclasses will implement to create different families.

```
public abstract class AbstractDuckFactory {

    public abstract Quackable createMallardDuck();
    public abstract Quackable createRedheadDuck();
    public abstract Quackable createDuckCall();
    public abstract Quackable createRubberDuck();
}
```

Each method creates one kind of duck.

Creating a factory that creates ducks without decorators

```
public class DuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new MallardDuck();
    }

    public Quackable createRedheadDuck() {
        return new RedheadDuck();
    }

    public Quackable createDuckCall() {
        return new DuckCall();
    }

    public Quackable createRubberDuck() {
        return new RubberDuck();
    }
}
```

DuckFactory extends the abstract factory.

Each method creates a product: a particular kind of Quackable. The actual product is unknown to the simulator – it just knows it's getting a Quackable.

CountingDuckFactory

CountingDuckFactory also extends the abstract factory.

```
public class CountingDuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new QuackCounter(new MallardDuck());
    }

    public Quackable createRedheadDuck() {
        return new QuackCounter(new RedheadDuck());
    }

    public Quackable createDuckCall() {
        return new QuackCounter(new DuckCall());
    }

    public Quackable createRubberDuck() {
        return new QuackCounter(new RubberDuck());
    }
}
```

Each method wraps the Quackable with the quack counting decorator. The simulator will never know the difference; it just gets back a Quackable. But now our rangers can be sure that all quacks ar being counted.

How Abstract Factory works? We create a polymorphic method that takes a factory and uses it to create objects. By passing in different factories, we get to use different product families in the method.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable mallardDuck = duckFactory.createMallardDuck();
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Abstract Factory");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

First we create the factory that we're going to pass into the simulate() method.

The simulate() method takes an AbstractDuckFactory and uses it to create ducks rather than instantiating them directly.

Nothing changes here! Same ol' code.

Manage a flock of ducks

- It’s getting a little difficult to manage all these different ducks separately. Is there any way you can help us manage ducks as a whole, and perhaps even allow us to manage a few duck “families” that we’d like to keep track of?

This isn’t very manageable!

```
Quackable mallardDuck = duckFactory.createMallardDuck();
Quackable redheadDuck = duckFactory.createRedheadDuck();
Quackable duckCall = duckFactory.createDuckCall();
Quackable rubberDuck = duckFactory.createRubberDuck();
Quackable gooseDuck = new GooseAdapter(new Goose());

simulate(mallardDuck);
simulate(redheadDuck);
simulate(duckCall);
simulate(rubberDuck);
simulate(gooseDuck);
```

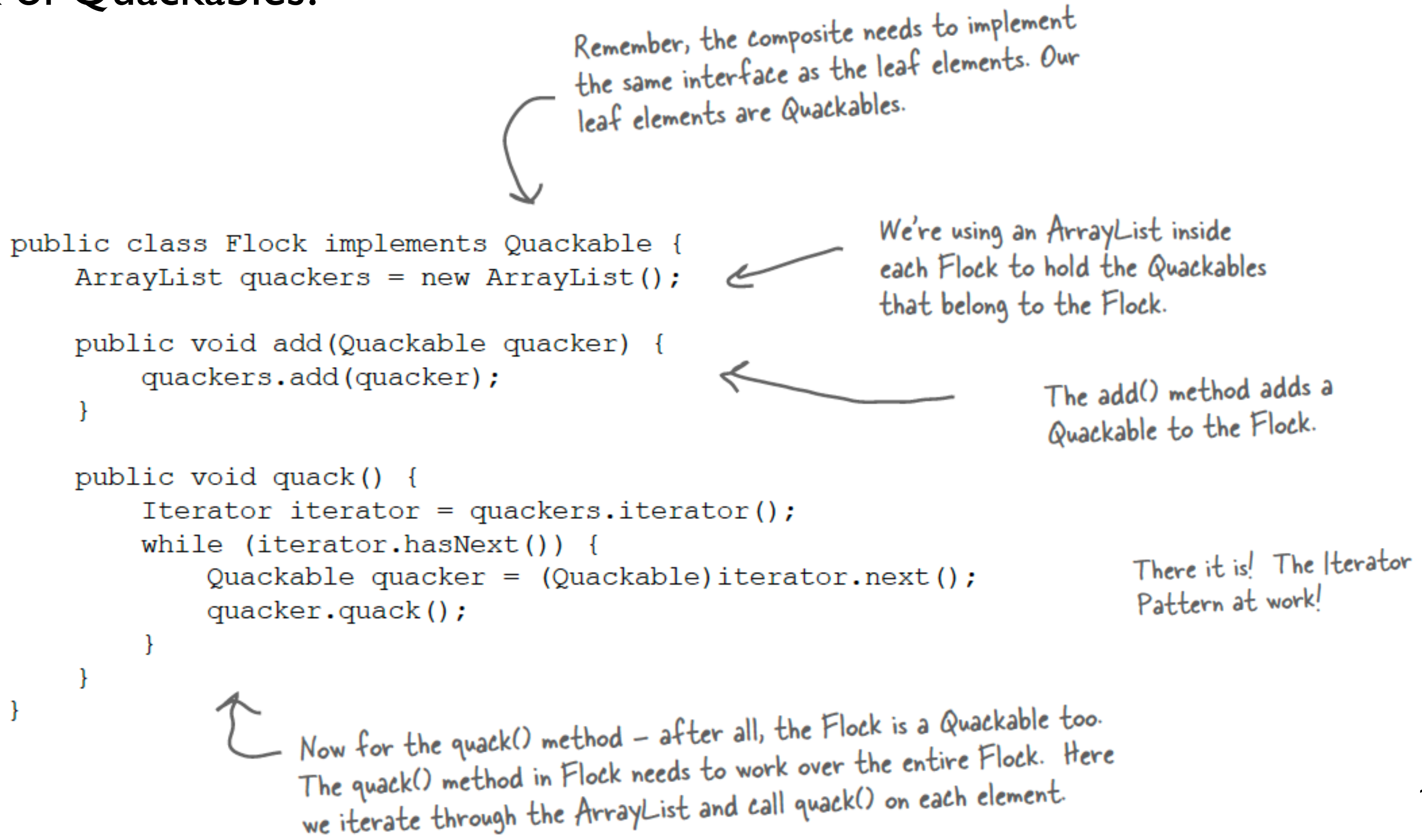
Manage a flock of ducks

- What we need is a way to talk about collections of ducks and even sub-collections of ducks (to deal with the family request from Ranger Brewer). It would also be nice if we could apply operations across the whole set of ducks.

- What pattern can help us?

a flock of ducks (a flock of Quackables)

Remember the Composite Pattern that allows us to treat a collection of objects in the same way as individual objects? What better composite than a flock of Quackables!



Remember, the composite needs to implement the same interface as the leaf elements. Our leaf elements are Quackables.

```
public class Flock implements Quackable {
    ArrayList quackers = new ArrayList();

    public void add(Quackable quacker) {
        quackers.add(quacker);
    }

    public void quack() {
        Iterator iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable quacker = (Quackable)iterator.next();
            quacker.quack();
        }
    }
}
```

We're using an ArrayList inside each Flock to hold the Quackables that belong to the Flock.

The add() method adds a Quackable to the Flock.

There it is! The Iterator Pattern at work!

Now for the quack() method – after all, the Flock is a Quackable too. The quack() method in Flock needs to work over the entire Flock. Here we iterate through the ArrayList and call quack() on each element.

Composite is ready

```
public class DuckSimulator {
    // main method here

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());
        System.out.println("\nDuck Simulator: With Composite - Flocks");

        Flock flockOfDucks = new Flock();

        flockOfDucks.add(redheadDuck);
        flockOfDucks.add(duckCall);
        flockOfDucks.add(rubberDuck);
        flockOfDucks.add(gooseDuck);

        Flock flockOfMallards = new Flock();
```

Create all the Quackables, just like before.

First we create a Flock, and load it up with Quackables.

Then we create a new Flock of Mallards.

Safety versus Transparency

- You might remember that in the Composite Pattern chapter the composites (the Menus) and the leaf nodes (the MenuItems) had the *same* exact set of methods, including the add() method. Because they had the same set of methods, we could call methods on MenuItems that didn't really make sense (like trying to add something to a MenuItem by calling add()). The benefit of this was that the distinction between leaves and composites was *transparent*: the client didn't have to know whether it was dealing with a leaf or a composite; it just called the same methods on both.
- Here, we've decided to keep the composite's child maintenance methods separate from the leaf nodes: that is, only Flocks have the add() method. We know it doesn't make sense to try to add something to a Duck, and in this implementation, you can't. You can only add() to a Flock. So this design is *safer* – you can't call methods that don't make sense on components – but it's less transparent. Now the client has to know that a Quackable is a Flock in order to add Quackables to it.
- As always, there are trade-offs when you do OO design and you need to consider them as you create your own composites.

Also track individual ducks

- The Composite is working great! Thanks! Now we have the opposite request: we also need to track individual ducks. Can you give us a way to keep track of individual duck quacking in real time?

Observer

First we need an Observable interface.

Remember that an Observable is the object being observed. An Observable needs methods for registering and notifying observers. We could also have a method for removing observers, but we'll keep the implementation simple here and leave that out.

QuackObservable is the interface that Quackables should implement if they want to be observed.

```
public interface QuackObservable {
    public void registerObserver(Observer observer);
    public void notifyObservers();
}
```

It has a method for registering Observers. Any object implementing the Observer interface can listen to quacks. We'll define the Observer interface in a sec.

It also has a method for notifying the observers.

Now we need to make sure all Quackables implement this interface...

```
public interface Quackable extends QuackObservable {
    public void quack();
}
```

So, we extend the Quackable interface with QuackObserver.

QuackObservable

Now, we need to make sure all the concrete classes that implement Quackable can handle being a QuackObservable.
We could approach this by implementing registration and notification in each and every class. But we're going to do it a little differently this time: we're going to encapsulate the registration and notification code in another class, call it Observable, and compose it with a QuackObservable. That way we only write the real code once and the QuackObservable just needs enough code to delegate to the helper class Observable.

Observable implements all the functionality a Quackable needs to be an observable. We just need to plug it into a class and have that class delegate to Observable.

Observable must implement QuackObservable because these are the same method calls that are going to be delegated to it.

In the constructor we get passed the QuackObservable that is using this object to manage its observable behavior. Check out the notify() method below; you'll see that when a notify occurs, Observable passes this object along so that the observer knows which object is quacking.

Here's the code for registering an observer.

And the code for doing the notifications.

```
public class Observable implements QuackObservable {
    ArrayList observers = new ArrayList();
    QuackObservable duck;

    public Observable(QuackObservable duck) {
        this.duck = duck;
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        Iterator iterator = observers.iterator();
        while (iterator.hasNext()) {
            Observer observer = (Observer)iterator.next();
            observer.update(duck);
        }
    }
}
```

Integrate the helper Observable with the Quackable classes

All we need to do is make sure the Quackable classes
are composed with an Observable and that they know how to delegate to it.

```
public class MallardDuck implements Quackable {
    Observable observable;

    public MallardDuck() {
        observable = new Observable(this);
    }

    public void quack() {
        System.out.println("Quack");
        notifyObservers();
    }

    public void registerObserver(Observer observer) {
        observable.registerObserver(observer);
    }

    public void notifyObservers() {
        observable.notifyObservers();
    }
}
```

Each Quackable has an Observable instance variable.

In the constructor, we create an Observable and pass it a reference to the MallardDuck object.

When we quack, we need to let the observers know about it.

Here's our two QuackObservable methods. Notice that we just delegate to the helper.

Observer

We've implemented everything we need for the Observables; now we need some Observers. We'll start with the Observer interface:

```
public interface Observer {
    public void update(QuackObservable duck);
}
```

Observer

We need to implement the Observable interface or else we won't be able to register with a QuackObservable.

```
public class Quackologist implements Observer {

    public void update(QuackObservable duck) {
        System.out.println("Quackologist: " + duck + " just quacked.");
    }
}
```

The Quackologist is simple; it just has one method, update(), which prints out the Quackable that just quacked.

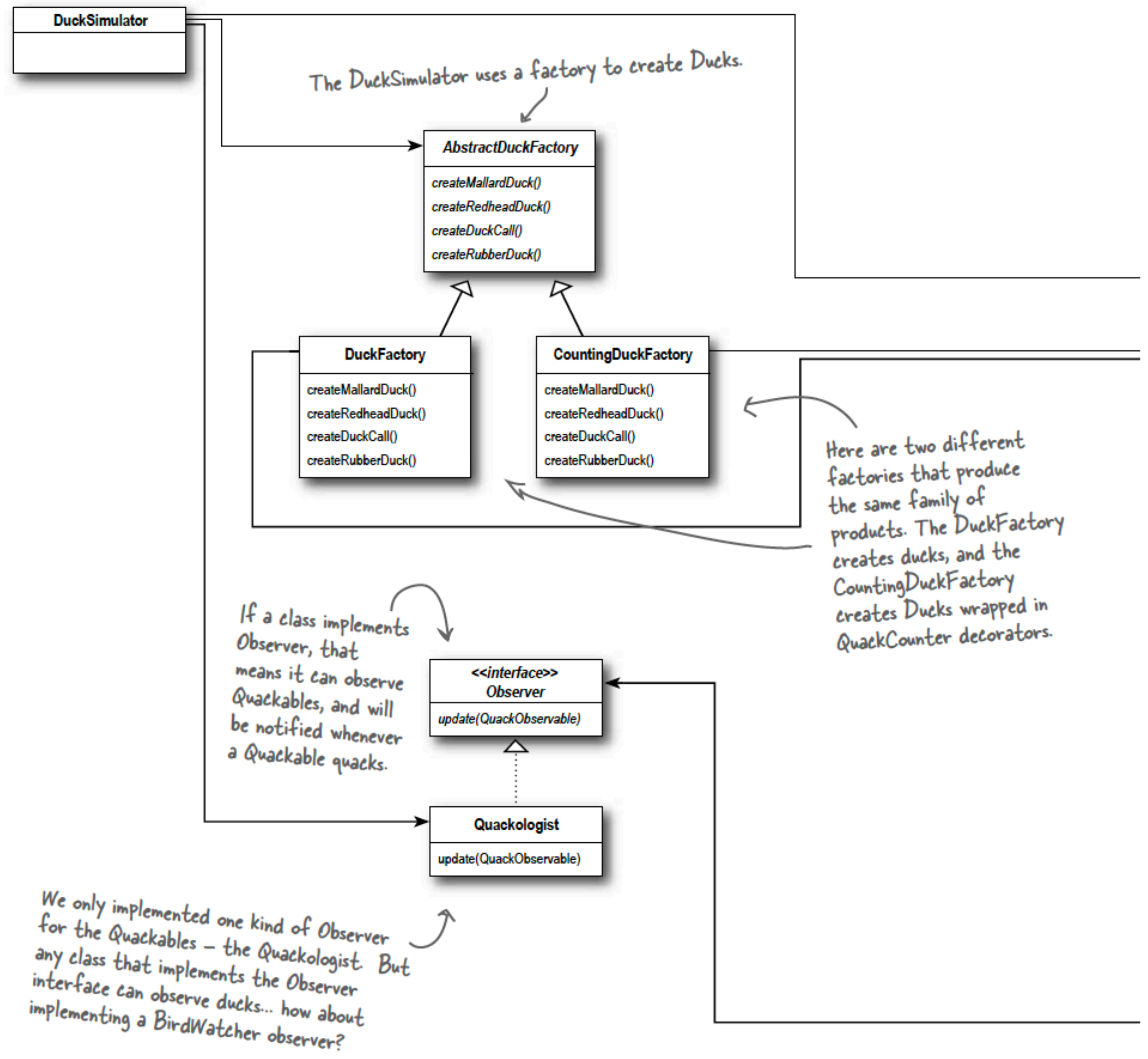
Simulator

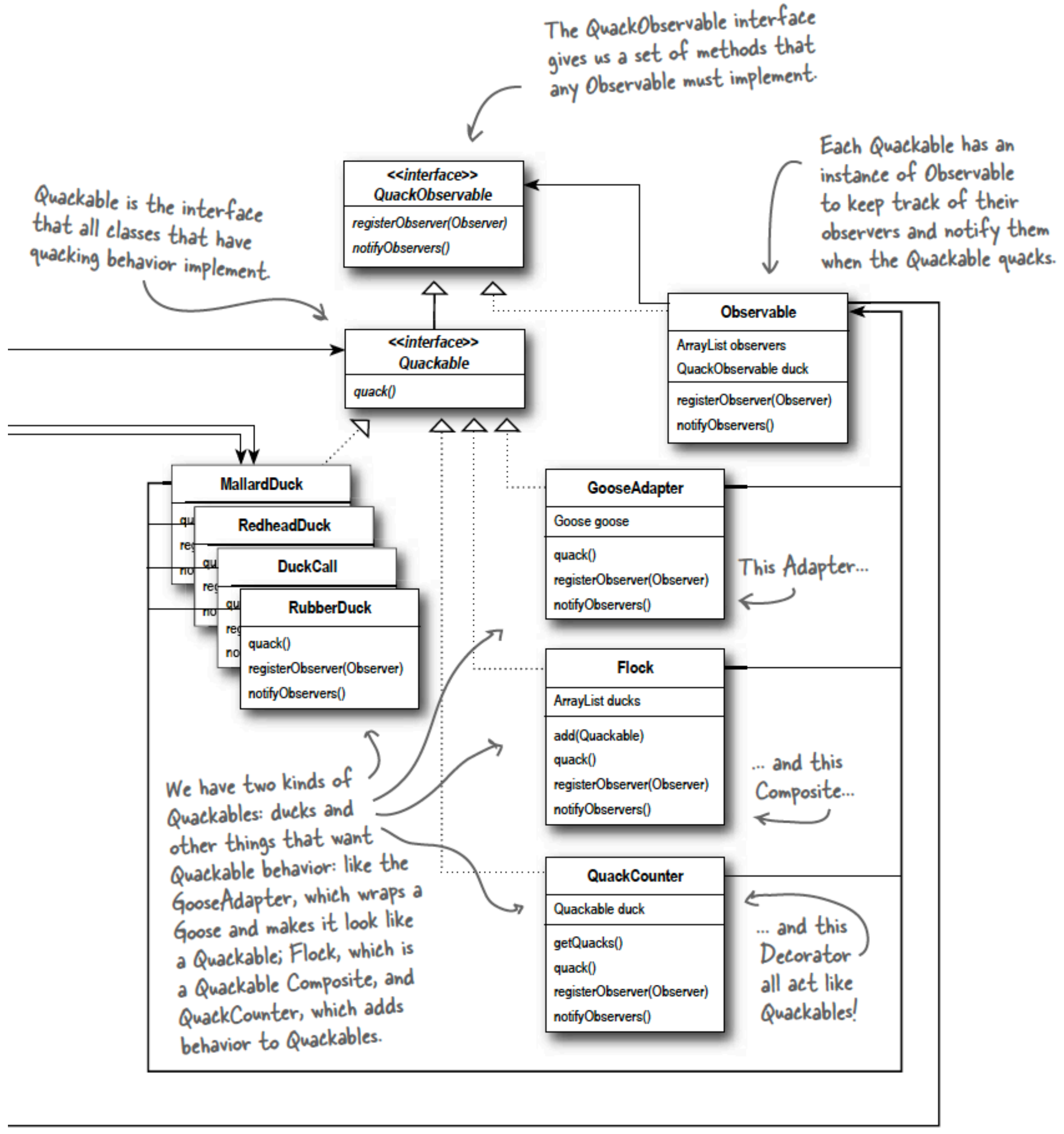
```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {

        // create duck factories and ducks here

        // create flocks here

        System.out.println("\nDuck Simulator: With Observer");
        Quackologist quackologist = new Quackologist();
        flockOfDucks.registerObserver(quackologist);

        simulate(flockOfDucks);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

All we do here is create a Quackologist and set him as an observer of the flock.

This time we'll we just simulate the entire flock.

Let's give it a try and see how it works!


DuckSimulator
The DuckSimulator uses a factory to create Ducks.
AbstractDuckFactory
createMallardDuck()
createRedheadDuck()
createDuckCall()
createRubberDuck()
DuckFactory
createMallardDuck()
createRedheadDuck()
createDuckCall()
createRubberDuck()
CountingDuckFactory
createMallardDuck()
createRedheadDuck()
createDuckCall()
createRubberDuck()
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.
<<interface>> Observer
update(QuackObservable)
Quackologist
update(QuackObservable)
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.
<<interface>> QuackObservable
registerObserver(Observer)
notifyObservers()
Each Quackable has an instance of Observable to keep track of their observers and notify them when the Quackable quacks.
Quackable is the interface that all classes that have quacking behavior implement.
<<interface>> Quackable
quack()
Observable
ArrayList observers
QuackObservable duck
registerObserver(Observer)
notifyObservers()
MallardDuck
RedheadDuck
DuckCall
RubberDuck
quack()
registerObserver(Observer)
notifyObservers()
GooseAdapter
Goose goose
quack()
registerObserver(Observer)
notifyObservers()
This Adapter...
Flock
ArrayList ducks
add(Quackable)
quack()
registerObserver(Observer)
notifyObservers()
... and this Composite...
QuackCounter
Quackable duck
getQuacks()
quack()
registerObserver(Observer)
notifyObservers()
... and this Decorator all act like Quackables!
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.