COMPOUND PATTERNS

Chandan R. Rupakheti

Week 9-1

Today …

Compound patterns!

That's right, we are now talking about patterns made of patterns!

Duck Reunion

This time the ducks are going to show you how patterns can coexist and even cooperate within the same solution.

1

First, we'll create a Quackable interface.

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

Quackables only need to do one thing well: Quack!

2

Now, some Ducks that implement Quackable

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

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

3

Wouldn't be much fun without other Ducks too

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

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

A RubberDuck that makes a squeak when it quacks.

Now we need a simulator …

```
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();
    }
}
```

Here's our main method to get everything going.

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.

```
File Edit Window Help ItBetterGetBetterThanThis
% java DuckSimulator
Duck Simulator
Quack
Quack
Kwak
Squeak
```

Honk Honk …

When ducks are around, geese can't be far

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

A Goose is a honker, not a quacker.

We need a goose adapter

```
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.

Now geese should work …

```
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.

There's the goose! Now the Goose can quack with the rest of the Ducks.

```
File Edit Window Help GoldenEggs
% java DuckSimulator
Duck Simulator: With Goose Adapter
Quack
Quack
Kwak
Squeak
Honk
```

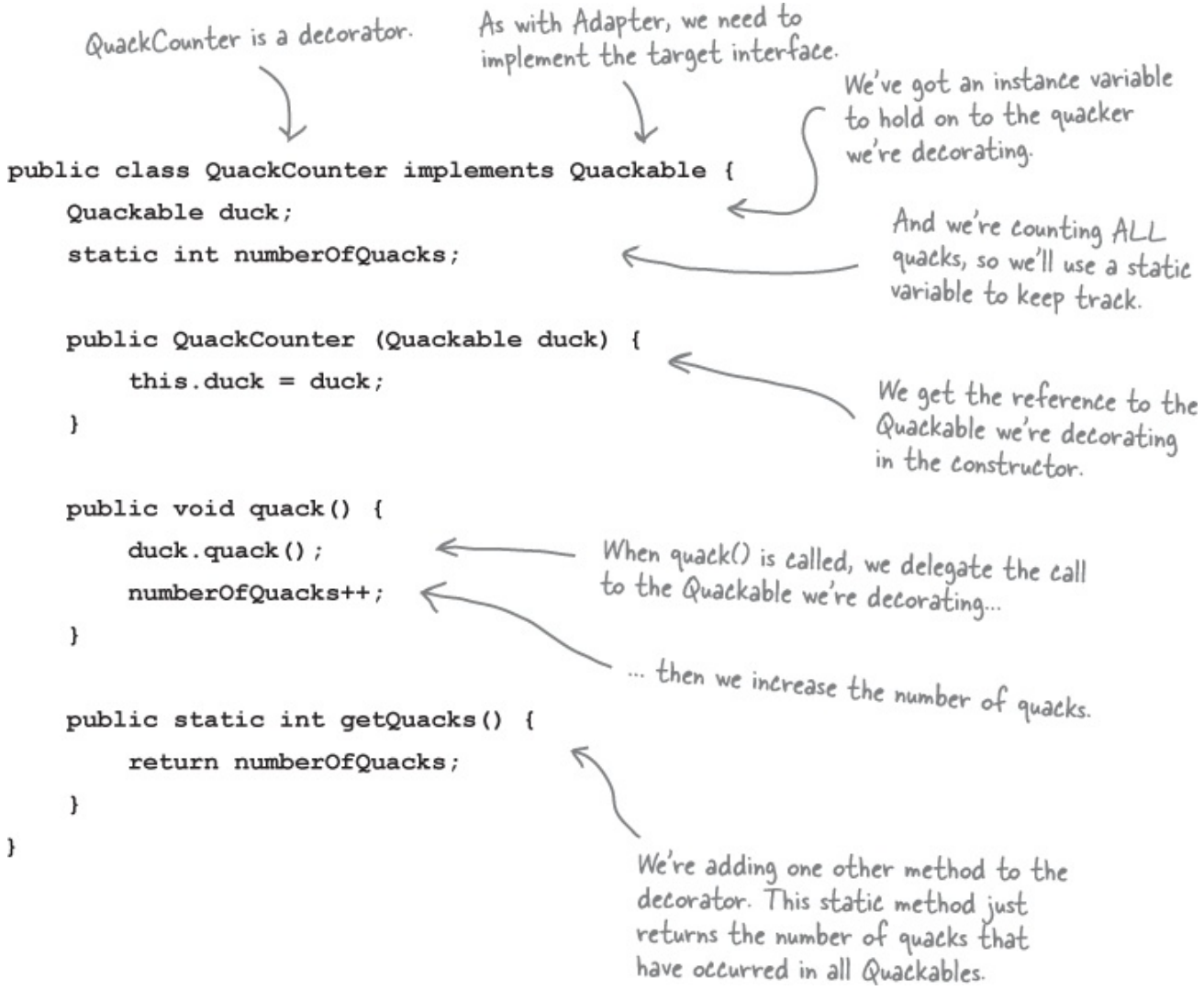
Quackology

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?

Let's make Quackologists happy




```
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;
    }
}
```

Updated Simulator for decorating ducks

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

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

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

        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();
    }
}
```

Each time we create a Quackable, we wrap it with a new decorator.

The park ranger told us he didn't want to count geese honks, so we don't decorate it.

Here's where we gather the quacking behavior for the Quackologists.

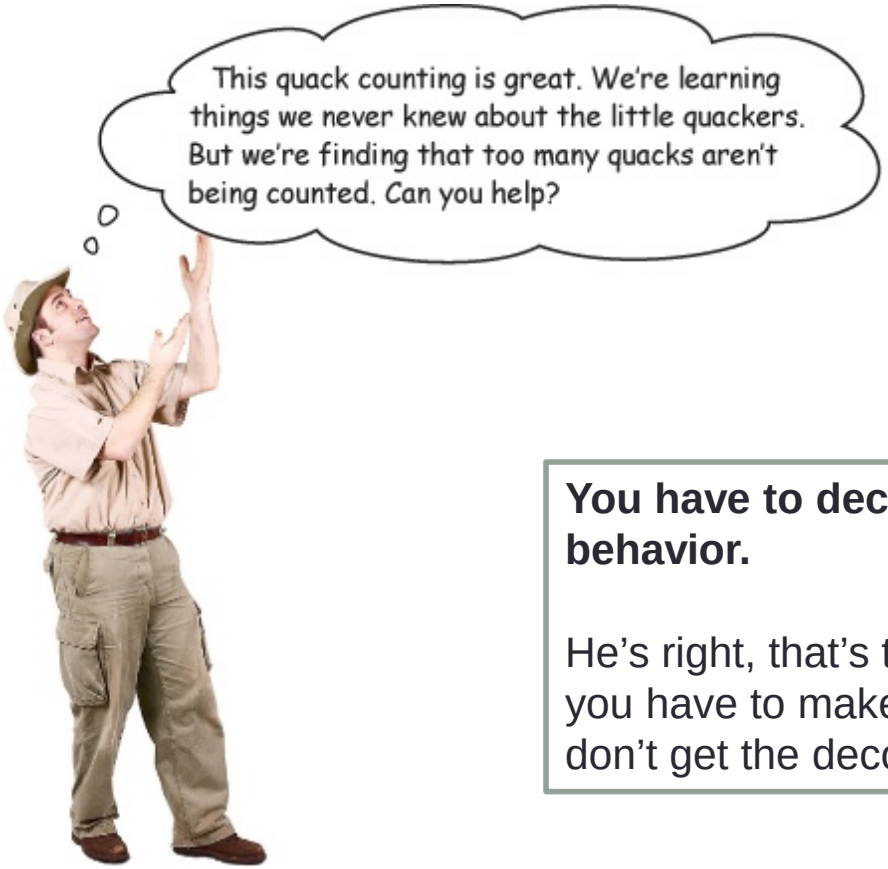
Nothing changes here; the decorated objects are still Quackables.

Here's the output!

```
File Edit Window Help DecoratedEggs
% java DuckSimulator
Duck Simulator: With Decorator
Quack
Quack
Kwak
Squeak
Honk
4 quacks were counted
%
```

Remember, we're not counting geese.

Help!!! Quacks aren't being counted properly!



You have to decorate objects to get decorated behavior.

He's right, that's the problem with wrapping objects: you have to make sure they get wrapped or they don't get the decorated behavior.

We need a factory to produce ducks

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.

```
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();
    }
}
```

```
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());
    }
}
```

Set up the simulator to use the factory

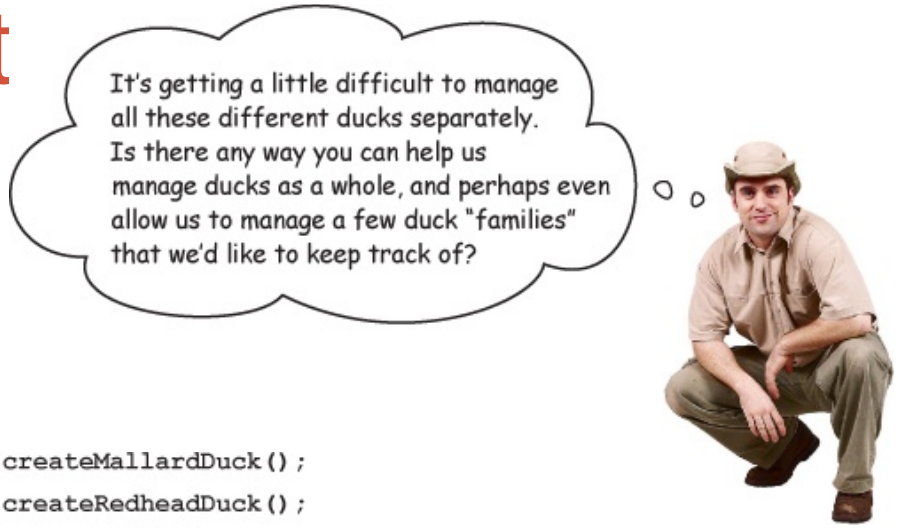
```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        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.

Duck Management



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);
```

He wants to manage a flock of ducks.
Well, actually a flock of Quackables

Let's create a flock of ducks

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

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

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

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

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

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

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.

Now we need to alter the simulator

```
File Edit Window Help FlockADuck
% java DuckSimulator

Duck Simulator: With Composite - Flocks
Duck Simulator: Whole Flock Simulation
Quack
Kwak
Squeak
Honk
Quack
Quack
Quack
Quack

Duck Simulator: Mallard Flock Simulation
Quack
Quack
Quack
Quack

The ducks quacked 11 times
```

Here's the first flock.

And now the mallards.

The data looks good (remember the goose doesn't get counted).

```
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();

        Quackable mallardOne = duckFactory.createMallardDuck();
        Quackable mallardTwo = duckFactory.createMallardDuck();
        Quackable mallardThree = duckFactory.createMallardDuck();
        Quackable mallardFour = duckFactory.createMallardDuck();

        flockOfMallards.add(mallardOne);
        flockOfMallards.add(mallardTwo);
        flockOfMallards.add(mallardThree);
        flockOfMallards.add(mallardFour);

        flockOfDucks.add(flockOfMallards);

        System.out.println("\nDuck Simulator: Whole Flock Simulation");
        simulate(flockOfDucks);

        System.out.println("\nDuck Simulator: Mallard Flock Simulation");
        simulate(flockOfMallards);

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

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

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.

Here we're creating a little family of mallards...

...and adding them to the Flock of mallards.

Then we add the Flock of mallards to the main flock.

Let's test out the entire Flock!

Then let's just test out the mallard's Flock.

Finally, let's give the Quackologist the data.

Nothing needs to change here; a Flock is a Quackable!

Real-time tracking of quacks …

Observer Pattern!

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.

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

So, we extend the Quackable interface with QuackObserver.

Now, we need to make sure all the concrete classes that implement Quackable can handle being a QuackObservable

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.

```
public class Observable implements QuackObservable {
    ArrayList<Observer> observers = new ArrayList<Observer>();
    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 = iterator.next();
            observer.update(duck);
        }
    }
}
```

In the constructor we get passed the QuackObservable that is using this object to manage its observable behavior. Check out the notifyObservers() 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.

Now let's see how a Quackable class uses this helper...

A twist on Observable ducks

```
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 are our two QuackObservable methods. Notice that we just delegate to the helper.

Need to work on the Observer side …

The Observer interface just has one method, update(), which is passed the QuackObservable that is quacking.

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

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.

What if a Quackologist wants to observe an entire flock?

Putting things together

```
File  Edit  Window  Help  DucksAreEverywhere

% java DuckSimulator

Duck Simulator: With Observer
Quack
Quackologist: Redhead Duck just quacked.
Kwak
Quackologist: Duck Call just quacked.
Squeak
Quackologist: Rubber Duck just quacked.
Honk
Quackologist: Goose pretending to be a Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
The Ducks quacked 7 times.
```

After each quack, no matter what kind of quack it was, the observer gets a notification.

And the quackologist still gets his counts.

```
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!

This is the big finale. Five, no, **six** patterns have come together to create this amazing Duck Simulator

Recap

A goose came along and wanted to act like a Quackable too – **Adapter**

Then, the Quackologists decided they wanted to count quacks – **Decorator**

But the Quackologists were worried they'd forget to add the QuackCounter decorator – **Abstract Factory**

We had management problems keeping track of all those ducks and geese and quackables – **Composite**

The Quackologists also wanted to be notified when any Quackable quacked – **Observer**

A ~~bird's~~ duck's eye view of class diagram