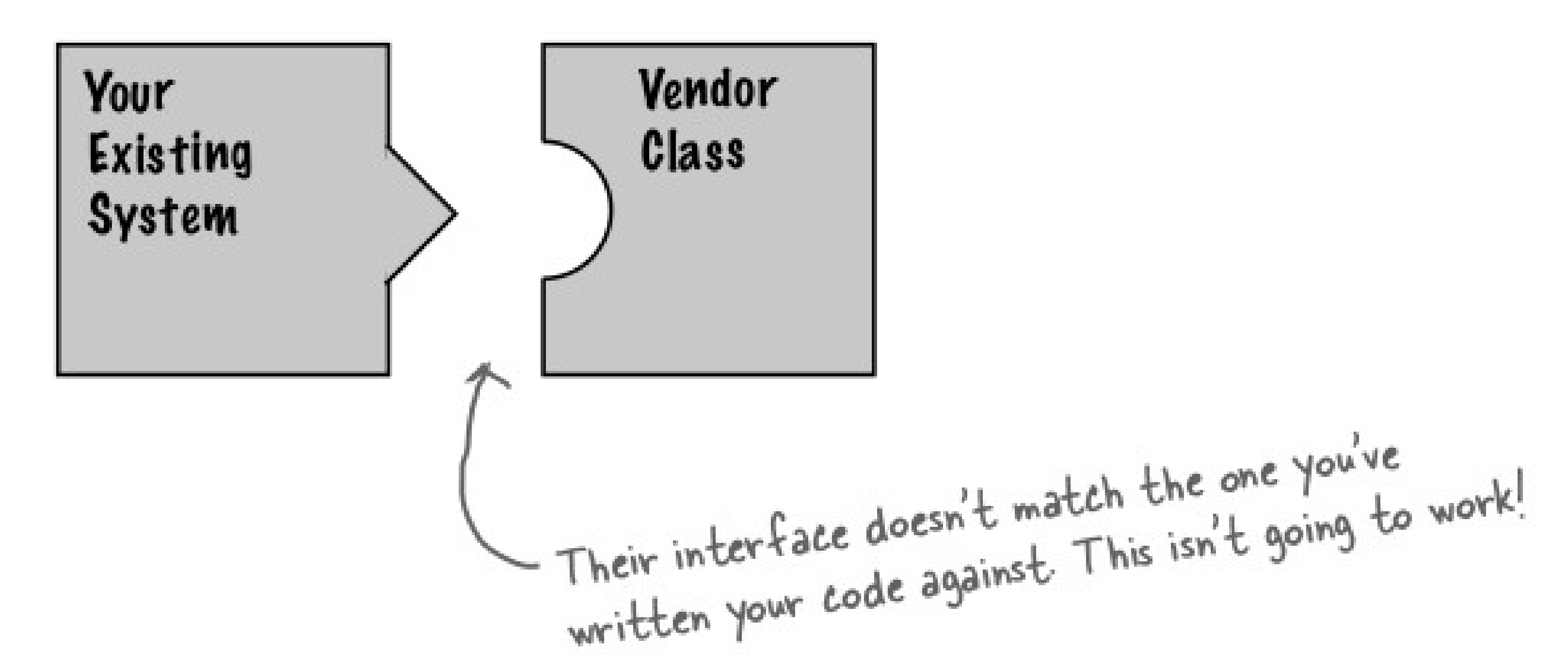
Patrón de Diseño: Adapter

Supongamos que existe un sistema de software que necesitas para trabajar con una nueva clase "Vendedor"

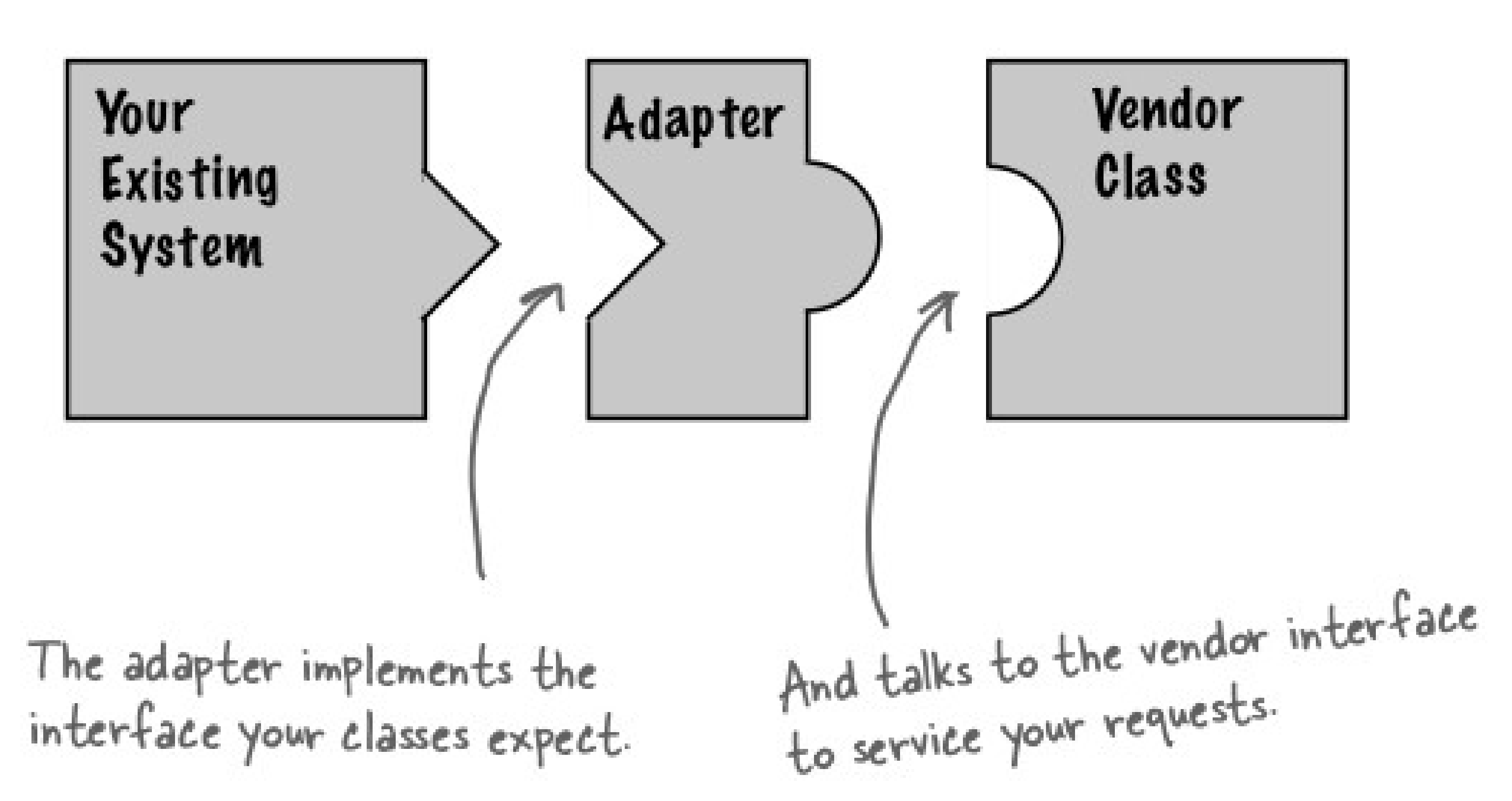
Pero el nuevo diseño de la interfaz del nuevo vendedor es diferente al anterior.

Entonces…


Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!

¿Cómo solucionarías esto?

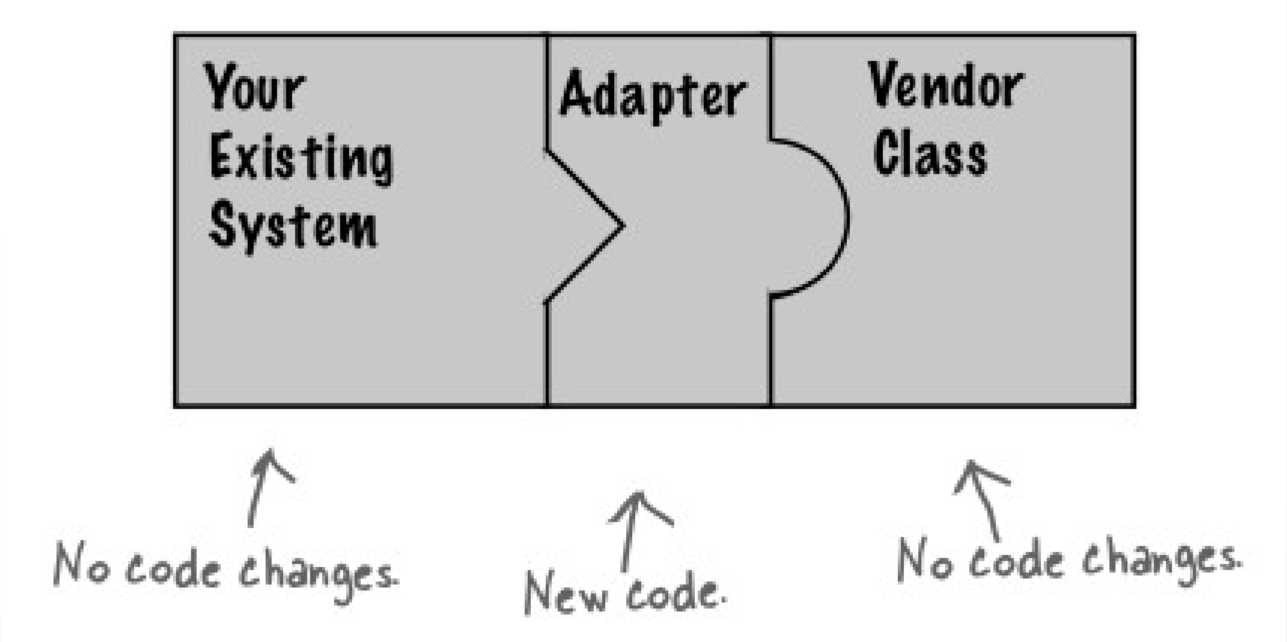
Crear una clase que se adapte a la nueva interfaz vendedor e implemente la clase que espera el código existente.



The adapter implements the interface your classes expect.

And talks to the vendor interface to service your requests.

El adaptador actúa como intermediario, recibiendo las peticiones del “cliente” y convertirlas en peticiones que tienen sentido en la clase “Vendedor”.

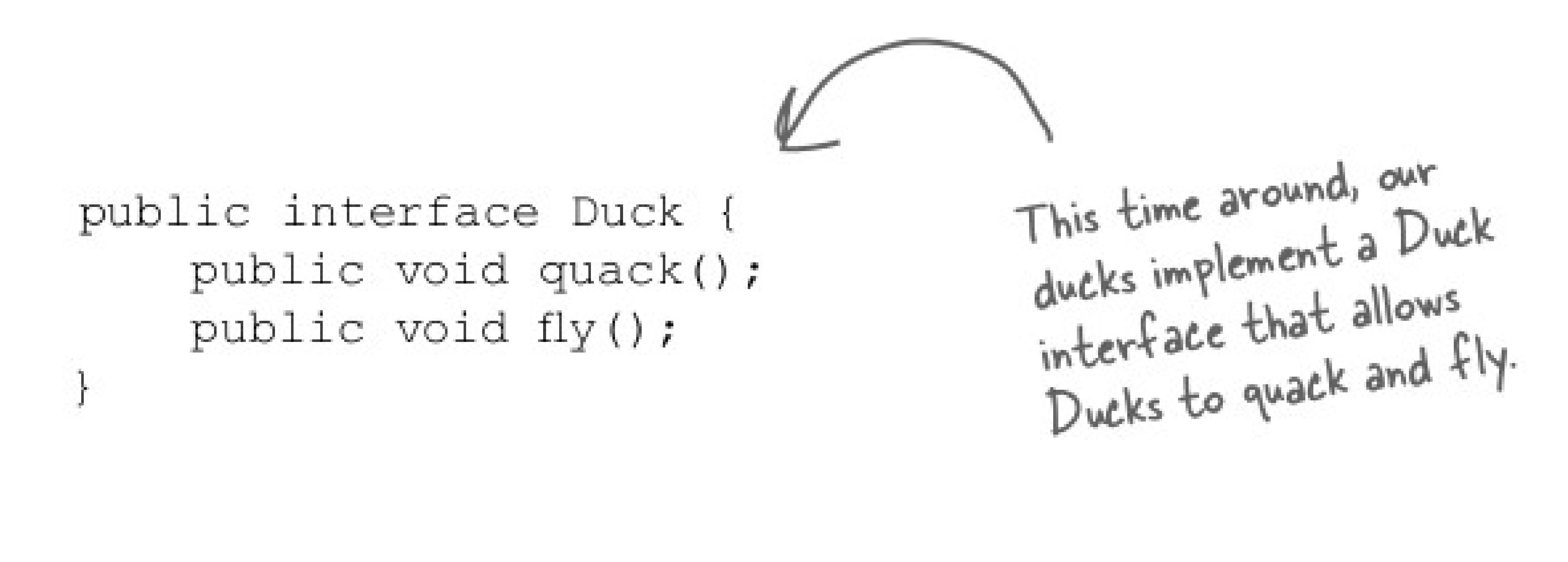


Si camina como un pato y haces quack como un pato, entonces ...

Si camina como un pato y haces quack como un pato, entonces

podría ser un pavo envuelto con un adaptador de pato

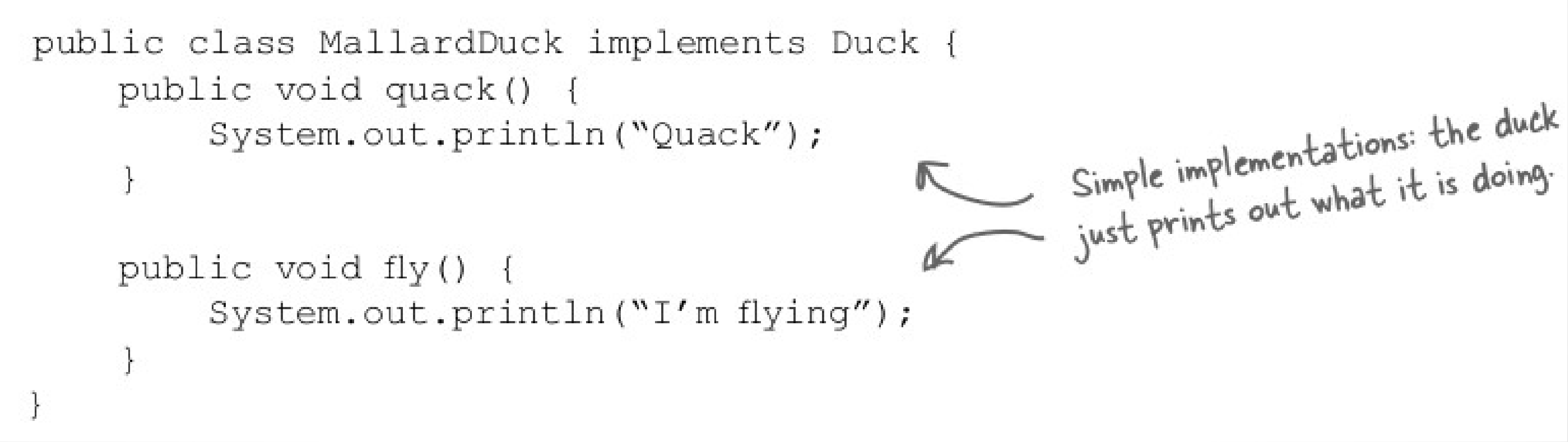
Simulación de estanque de Patos:



```
public interface Duck {
    public void quack();
    public void fly();
}
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

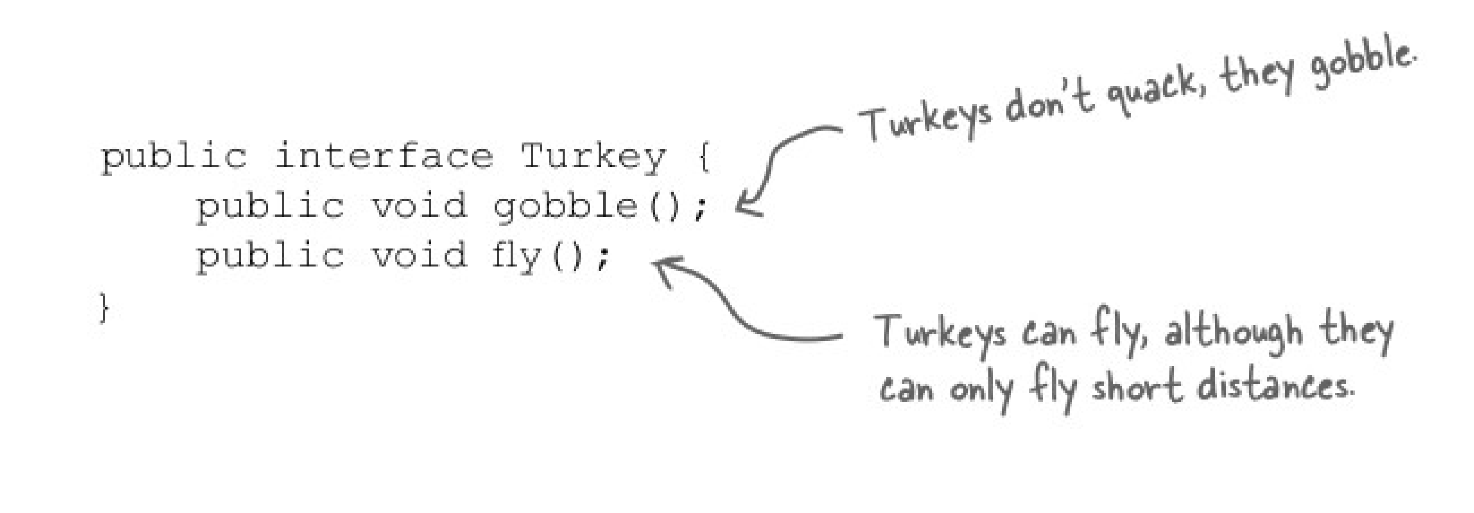
Implementación de una Subclase Pato:



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

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing.

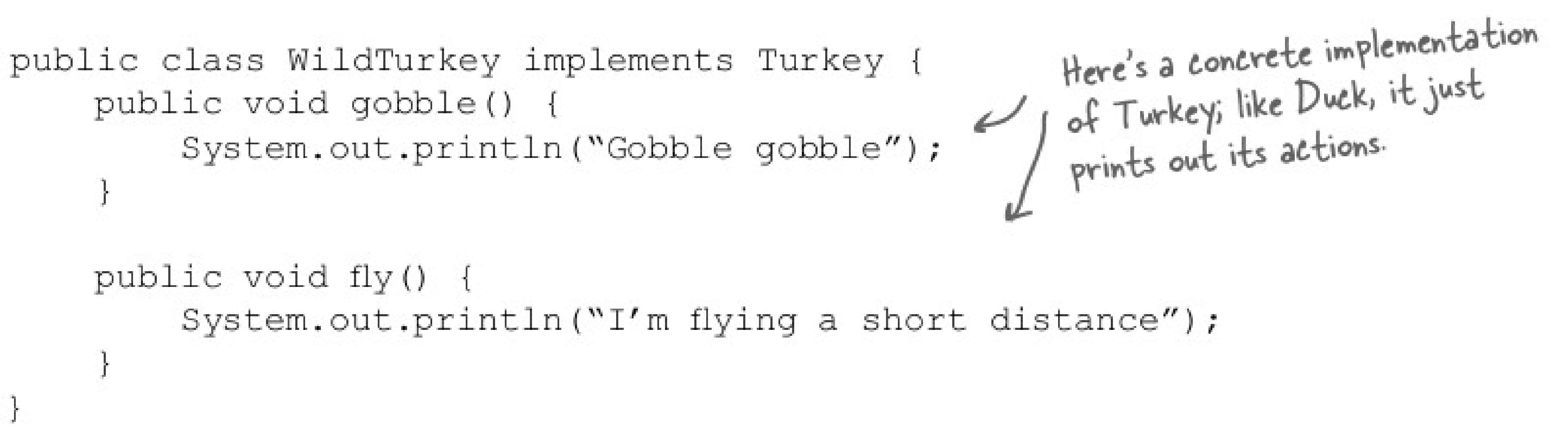
Ahora, introducimos una nueva ave: PAVO



```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don’t quack, they gobble.

Turkeys can fly, although they can only fly short distances.



```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println(“Gobble gobble”);
    }

    public void fly() {
        System.out.println(“I’m flying a short distance”);
    }
}
```

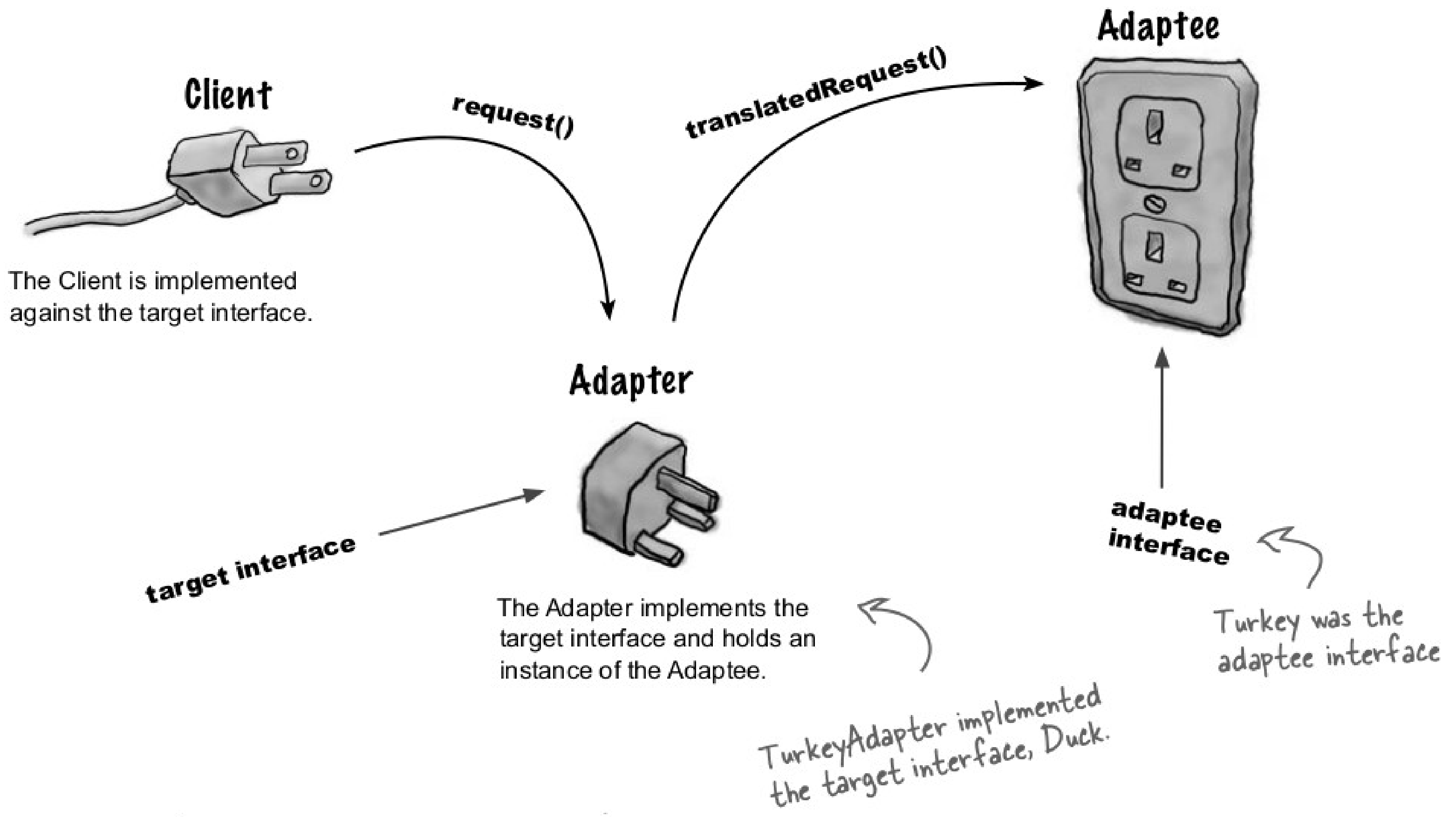
Here’s a concrete implementation of Turkey; like Duck, it just prints out its actions.

¿Qué pasa si no existen suficientes objetos pato?

Qué tal si utilizamos algunos objetos pavo en su lugar

Solución…

Crear una clase “Adaptador”.

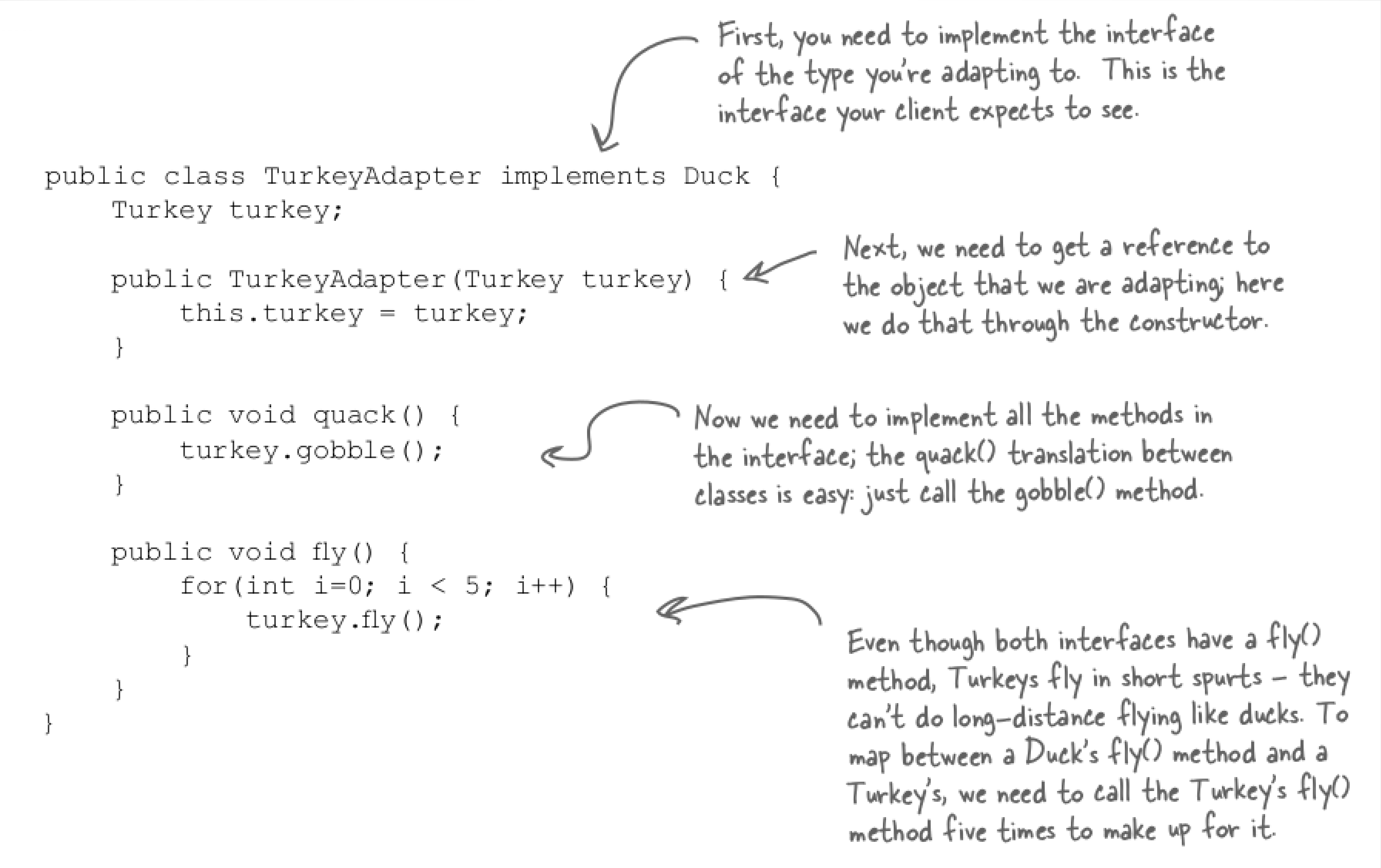


The Client is implemented against the target interface.

The Adapter implements the target interface and holds an instance of the Adaptee.

TurkeyAdapter implemented the target interface, Duck.

Turkey was the adaptee interface



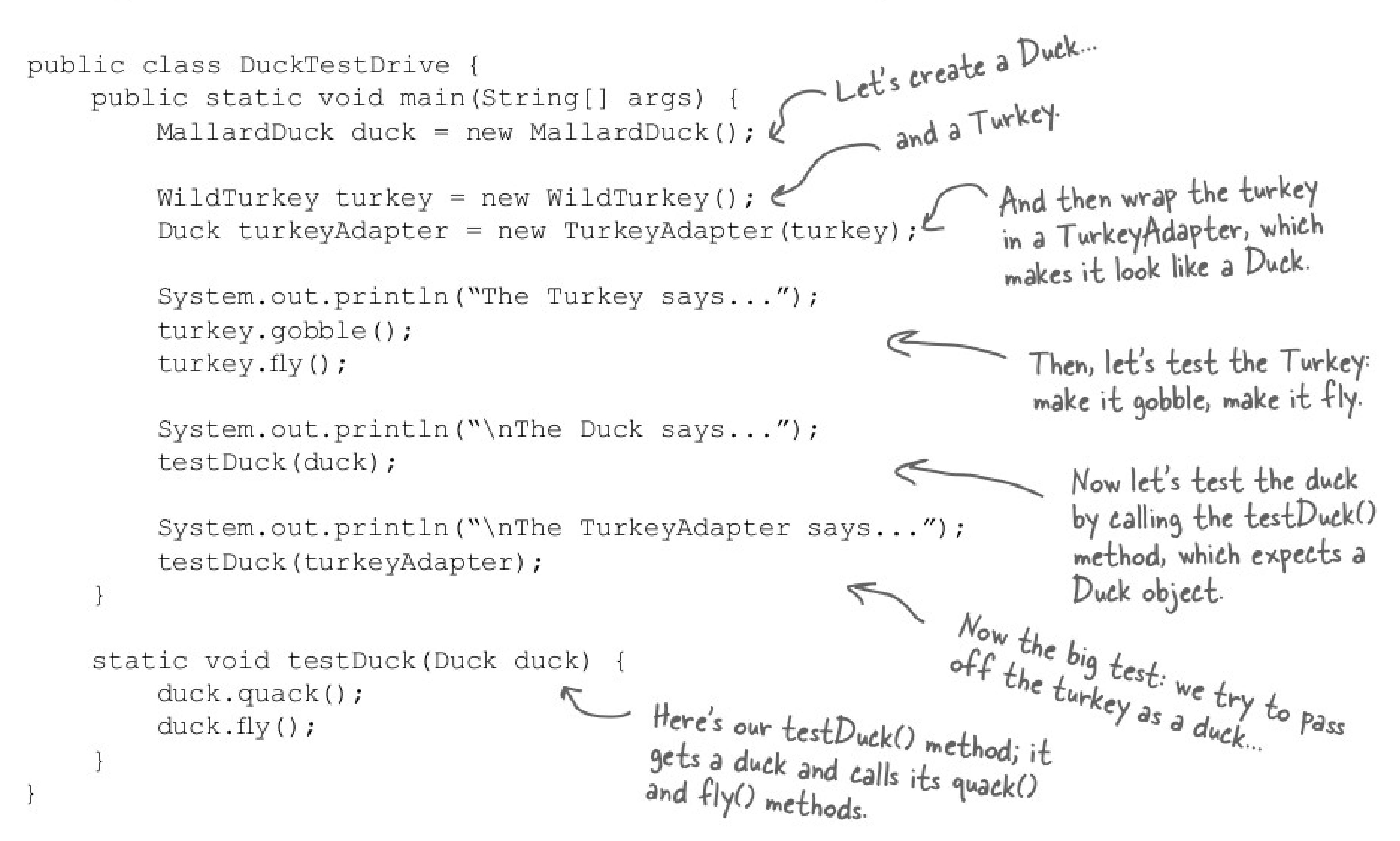
First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.



```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println(“The Turkey says...”);
        turkey.gobble();
        turkey.fly();

        System.out.println(“\nThe Duck says...”);
        testDuck(duck);

        System.out.println(“\nThe TurkeyAdapter says...”);
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck...

and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

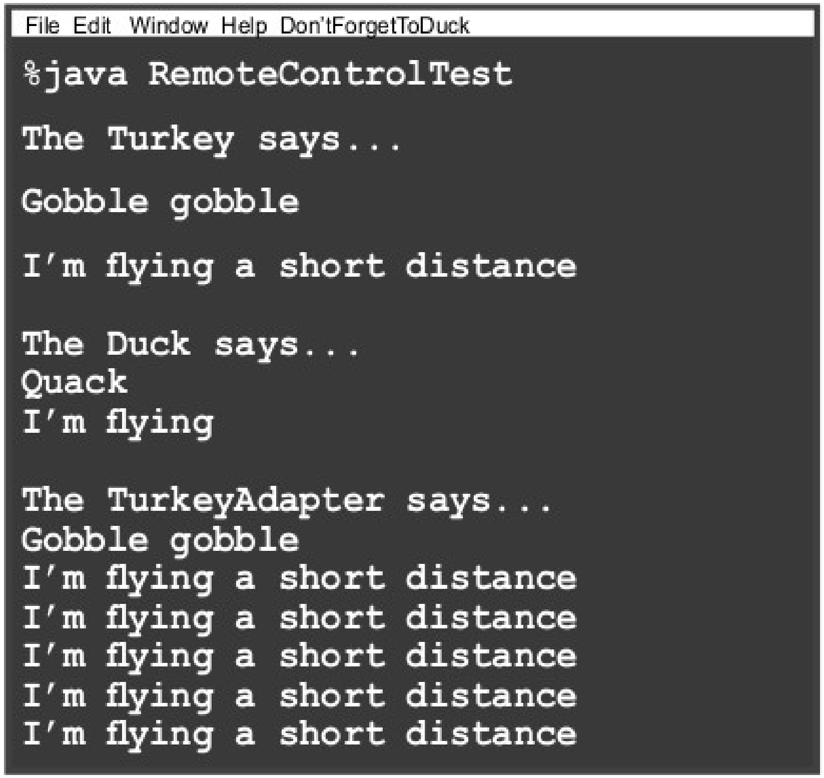
Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Test run



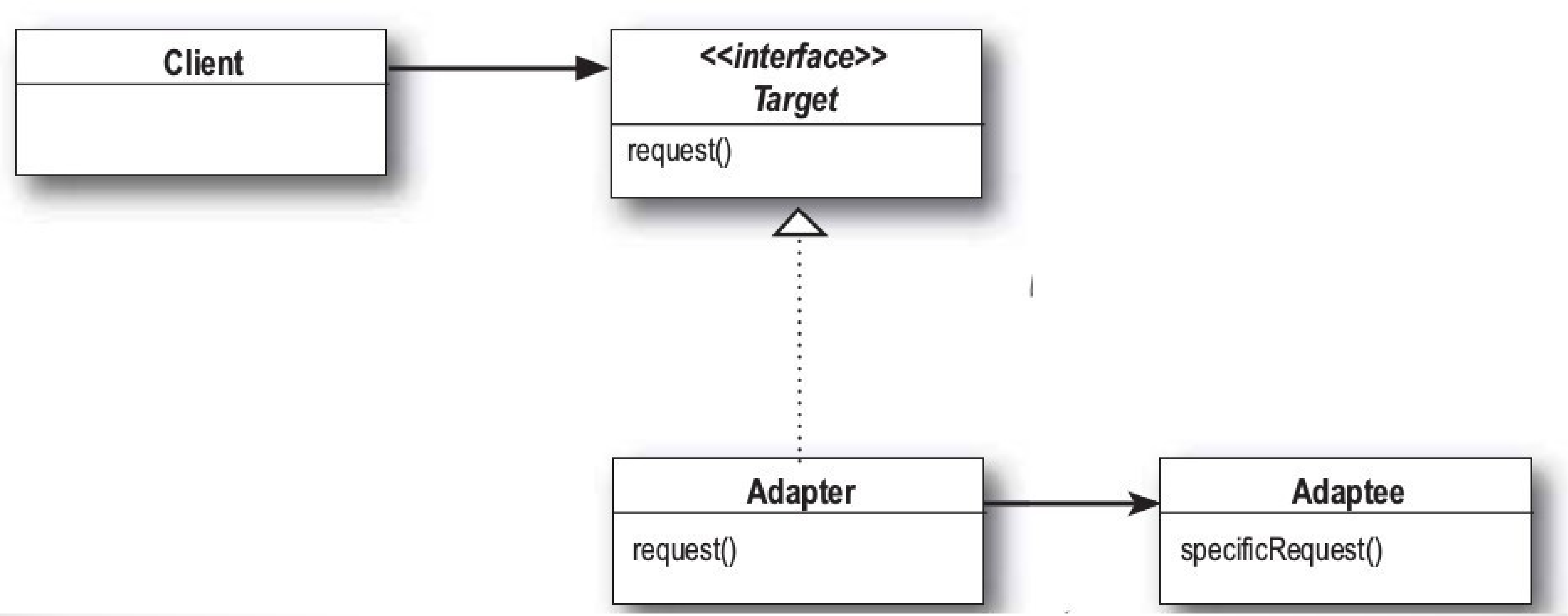
```
%java RemoteControlTest

The Turkey says...

Gobble gobble

I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

The Turkey gobbles and flies a short distance.

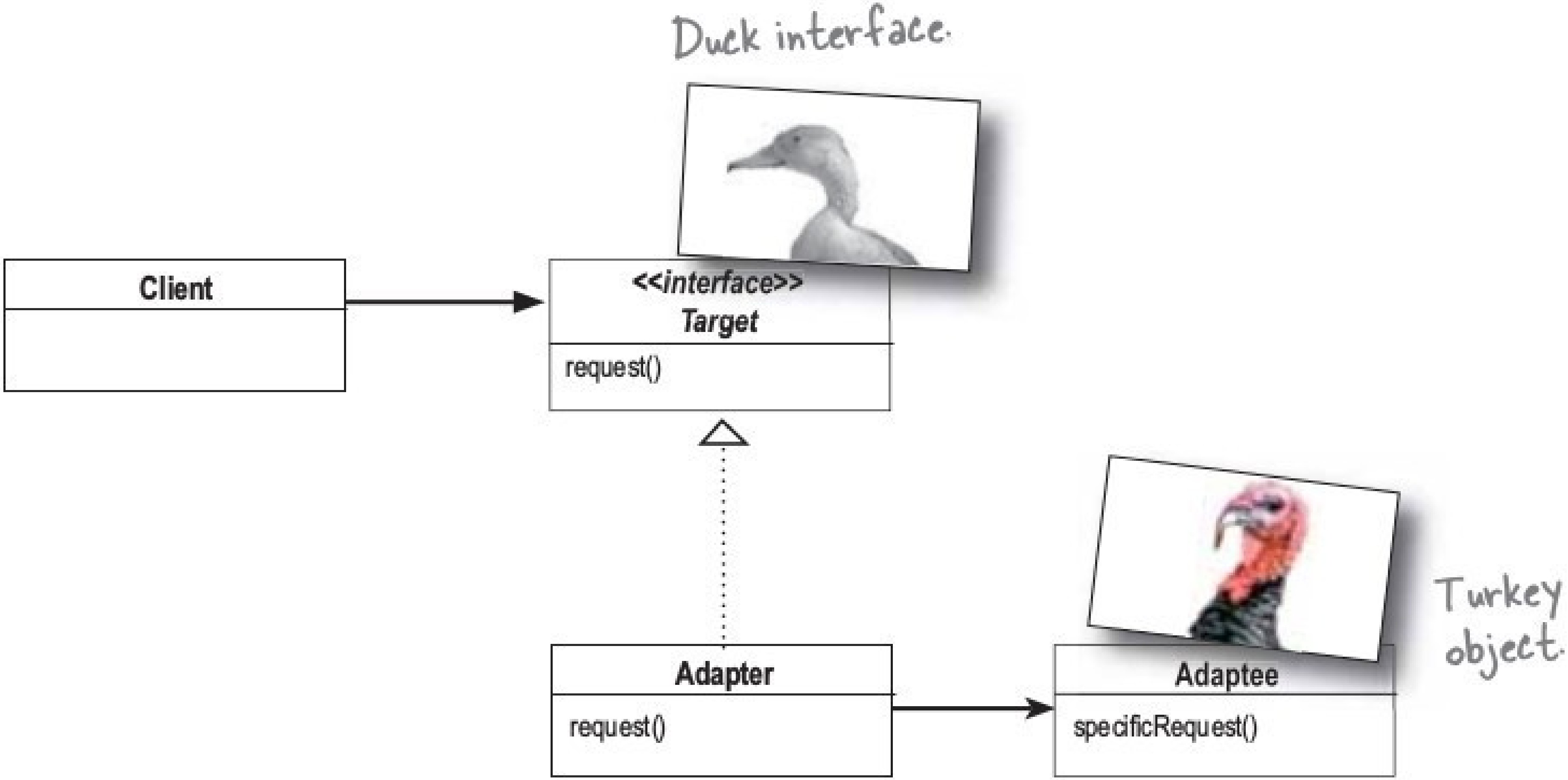
The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

Diagrama de clases:



Aplicandolo:



1) El cliente realiza una solicitud al adaptador llamando a un método usando la interfaz de destino.

2) El adaptador traduce la solicitud en una o más llamadas en el adaptador utilizando la interfaz adaptable.

3) El cliente recibe los resultados de la llamada y nunca sabe que hay un adaptador que realiza la traducción.

Entonces:

El **patrón de diseño “Adapter”** convierte la interfaz de una clase en otra interfaz que se adapte a la que el cliente espera.

Permite a las clases trabajar juntas, a pesar de que sus interfaces sean incompatibles.