

TC3003: Diseño y Arquitectura de Software

Dr. Juan Manuel González Calleros

Email: jmgonzale@itesm.mx

Twitter: [@Juan__Gonzalez](https://twitter.com/Juan__Gonzalez)

Facebook: **Juan Glez Calleros**

Reuniones pedir cita



3.PATRONES DE DISEÑO DE SOFTWARE

Patrones de Diseño de Software

- Simulador de Patos
 - Una app para mostrar los tipos de patos
 - Nadadores (swim())
 - Quackeadores (quack())
 - Presentación (display ())
 - ¿Cómo queda su clase Duck?
 - Dejen de lado los atributos por el momento.

Simulador de Patos

- ¿Qué métodos son abstractos y se deben sobre-escribir en las subclases?
- ¿Qué código puedo escribir una vez y sirve para todos?

<i>Duck</i>
<code>quack()</code> <code>swim()</code> <code>display()</code> <code>// OTHER duck-like methods...</code>

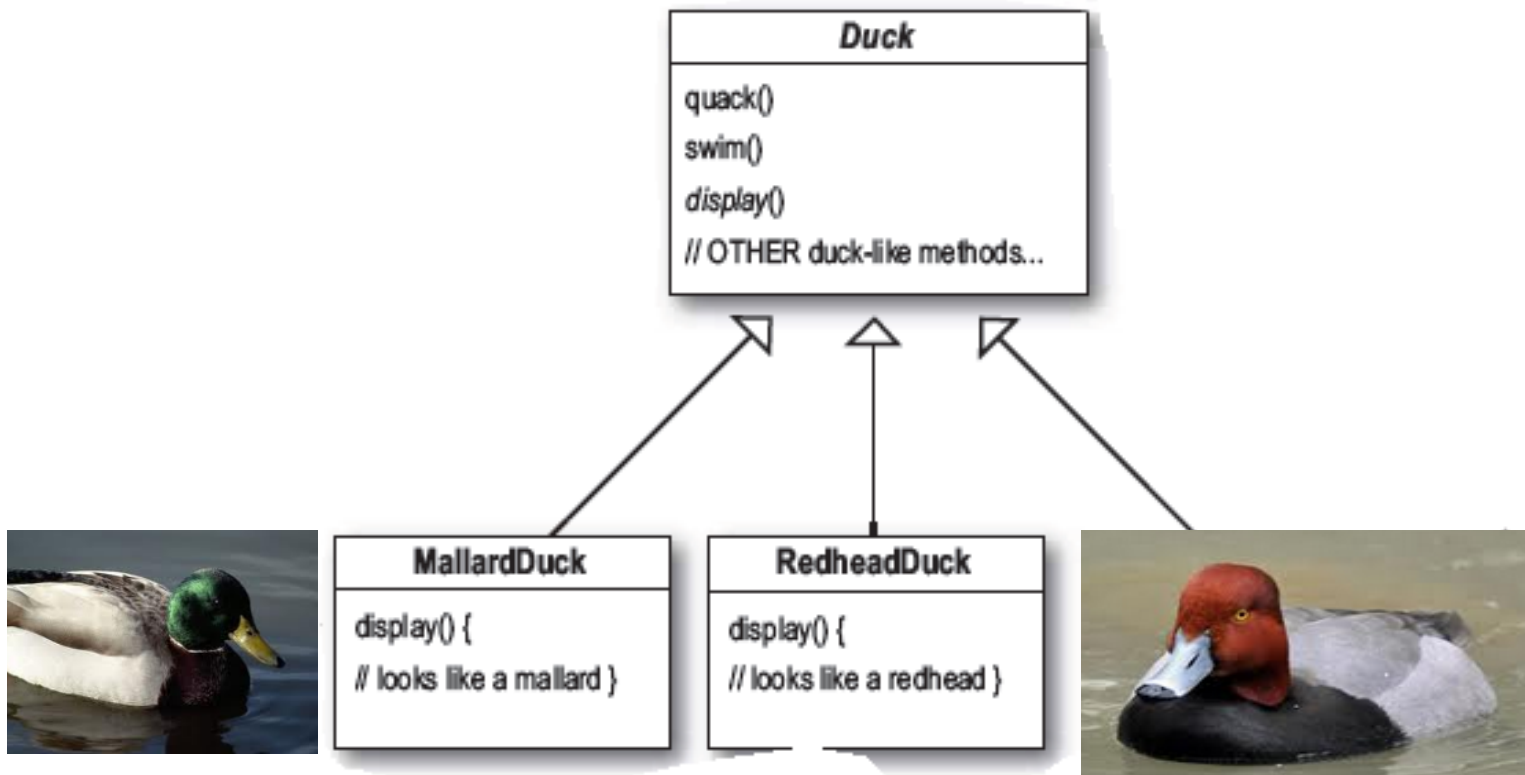
Simulador de Patos

- ¿Qué métodos son abstractos y se deben sobre-escribir en las subclases? **display()**
 - ¿Qué código puedo escribir una vez y sirve para todos? **quack()** y **swim ()**
- ¿Todos de acuerdo?**

<i>Duck</i>
<code>quack()</code> <code>swim()</code> <code>display()</code> <code>// OTHER duck-like methods...</code>

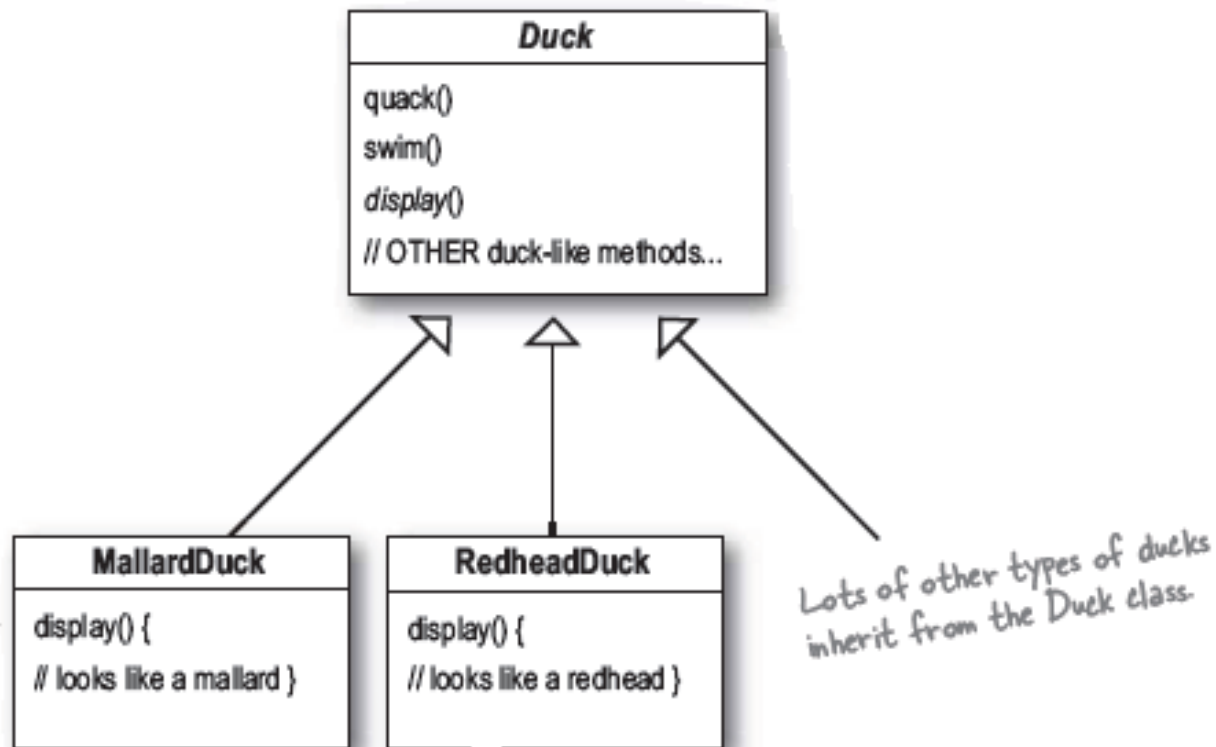
Simulador de Patos

- El método **display()** se define en cada subclase pues es diferente para cada pato



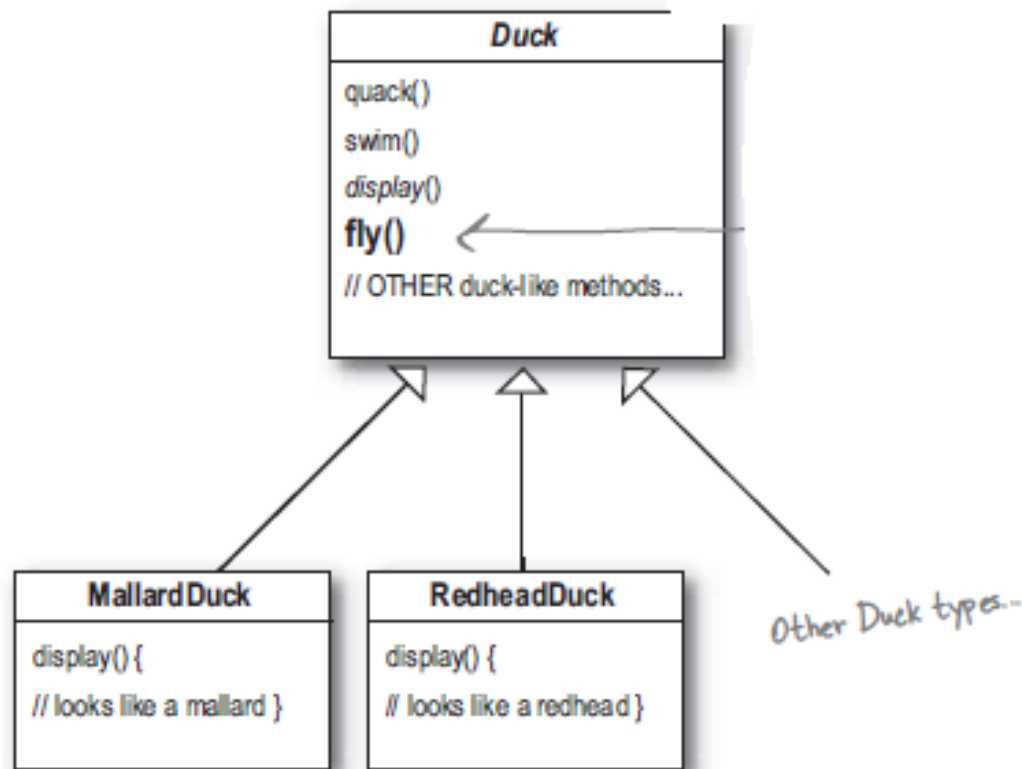
Simulador de Patos

- Los patos queremos que vuelen
- ¿Qué cambia?



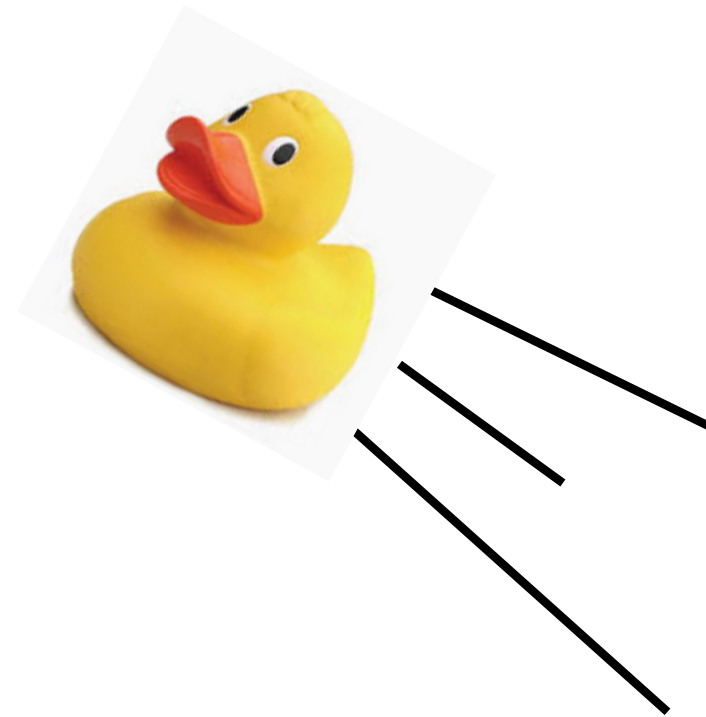
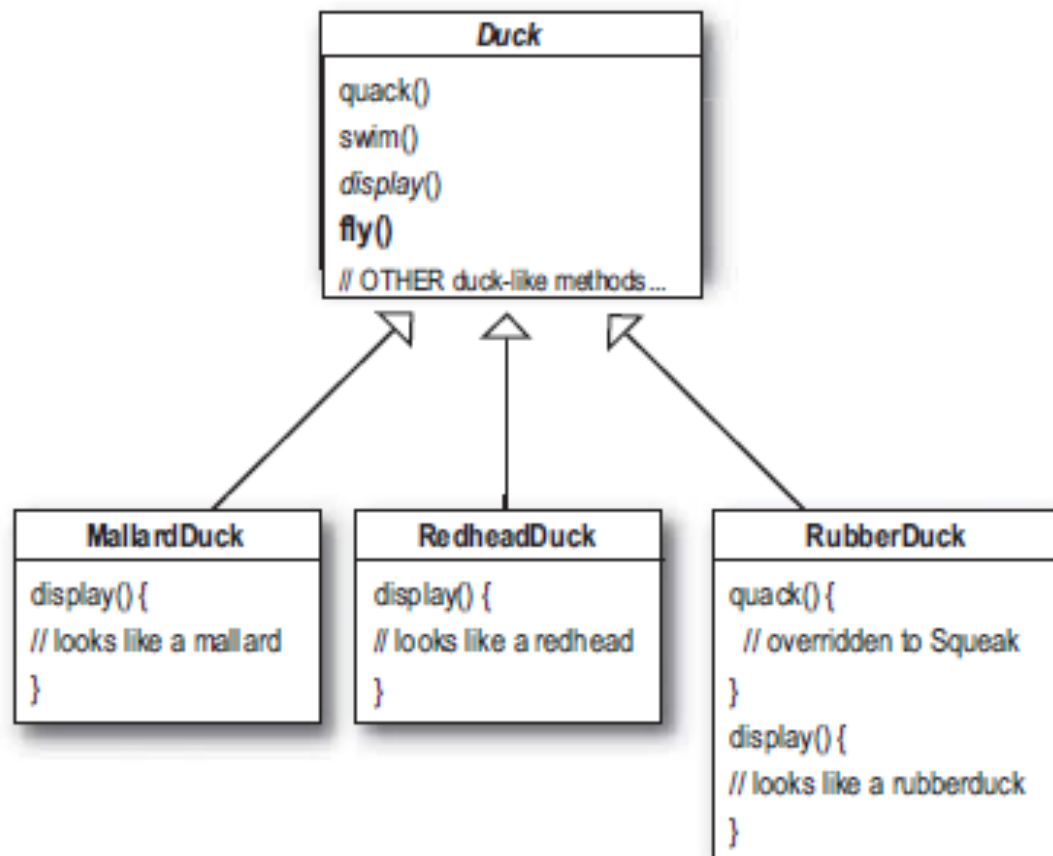
Simulador de Patos

- Los patos queremos que vuelen
- ¿Qué cambia?



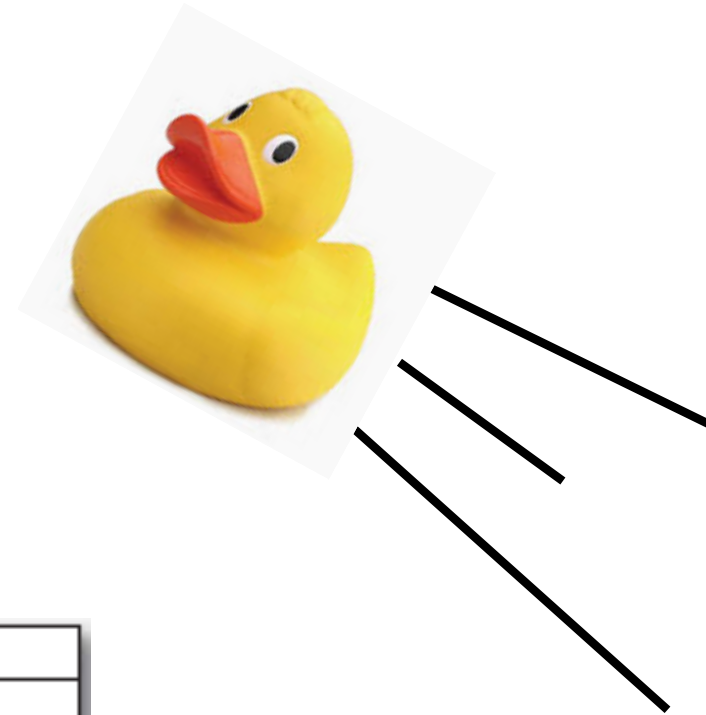
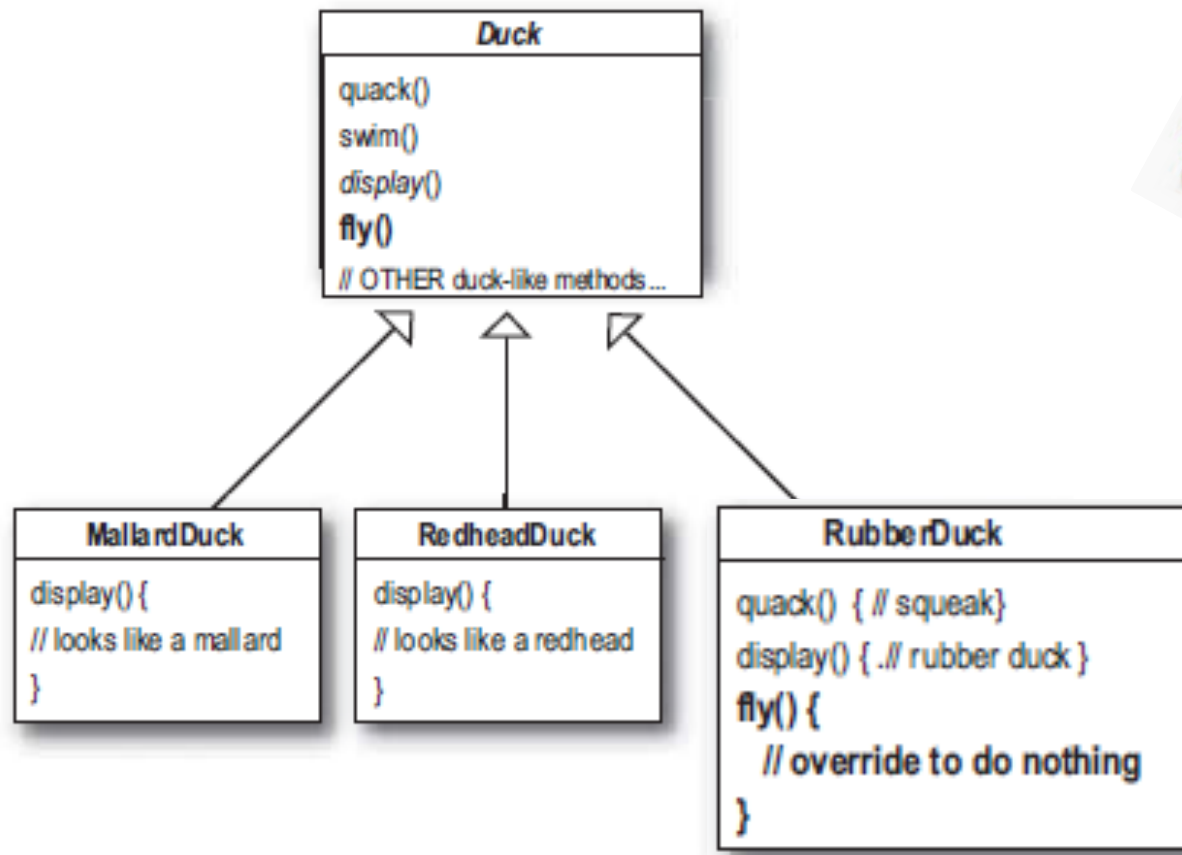
Simulador de Patos

- No todos vuelan, ah!, incluso no todos graznan



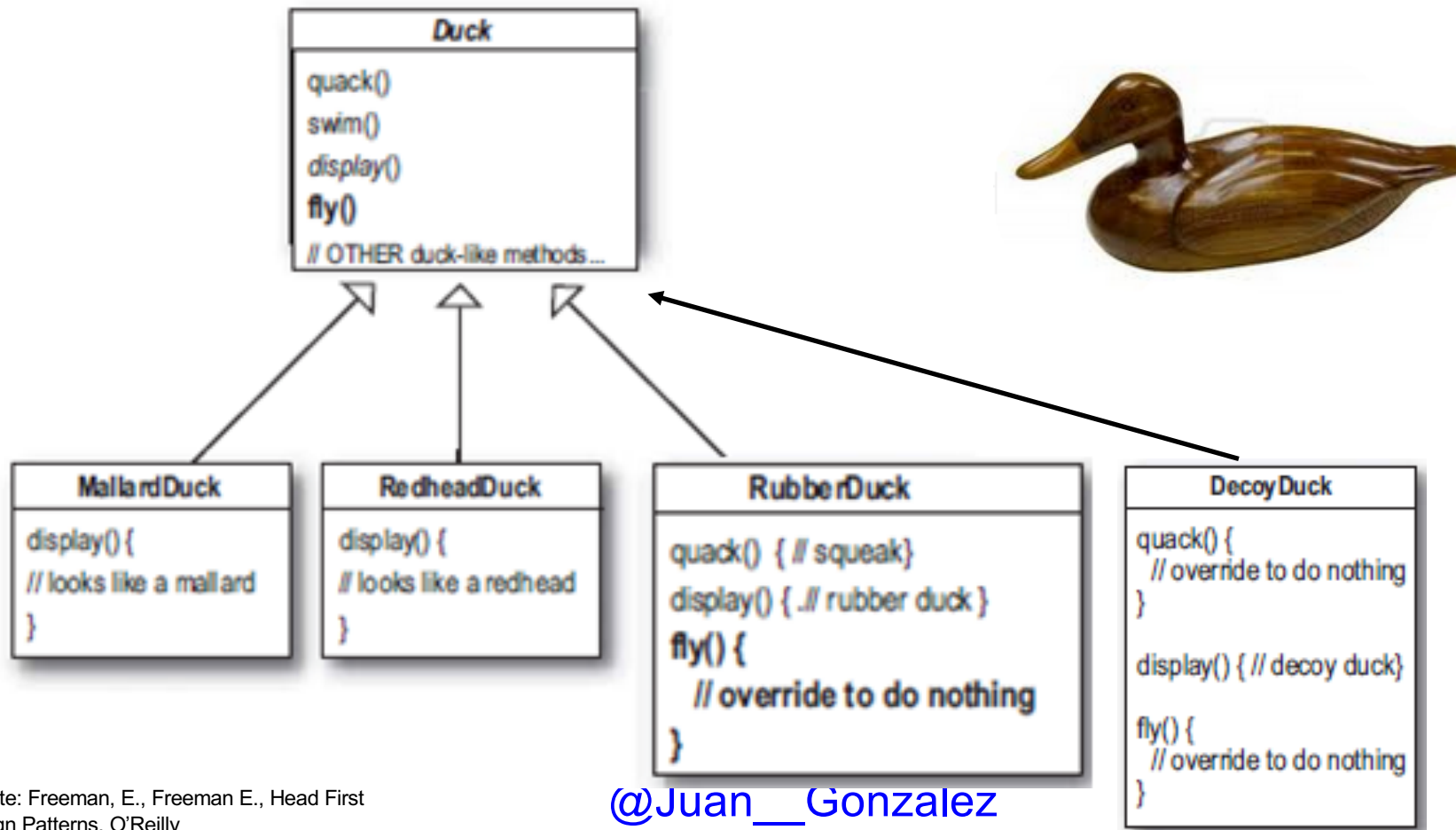
Simulador de Patos

- ¿Qué dicen si sobre escribimos?



Simulador de Patos

- ¿Y si queremos un patito de madera, hacemos lo mismo?

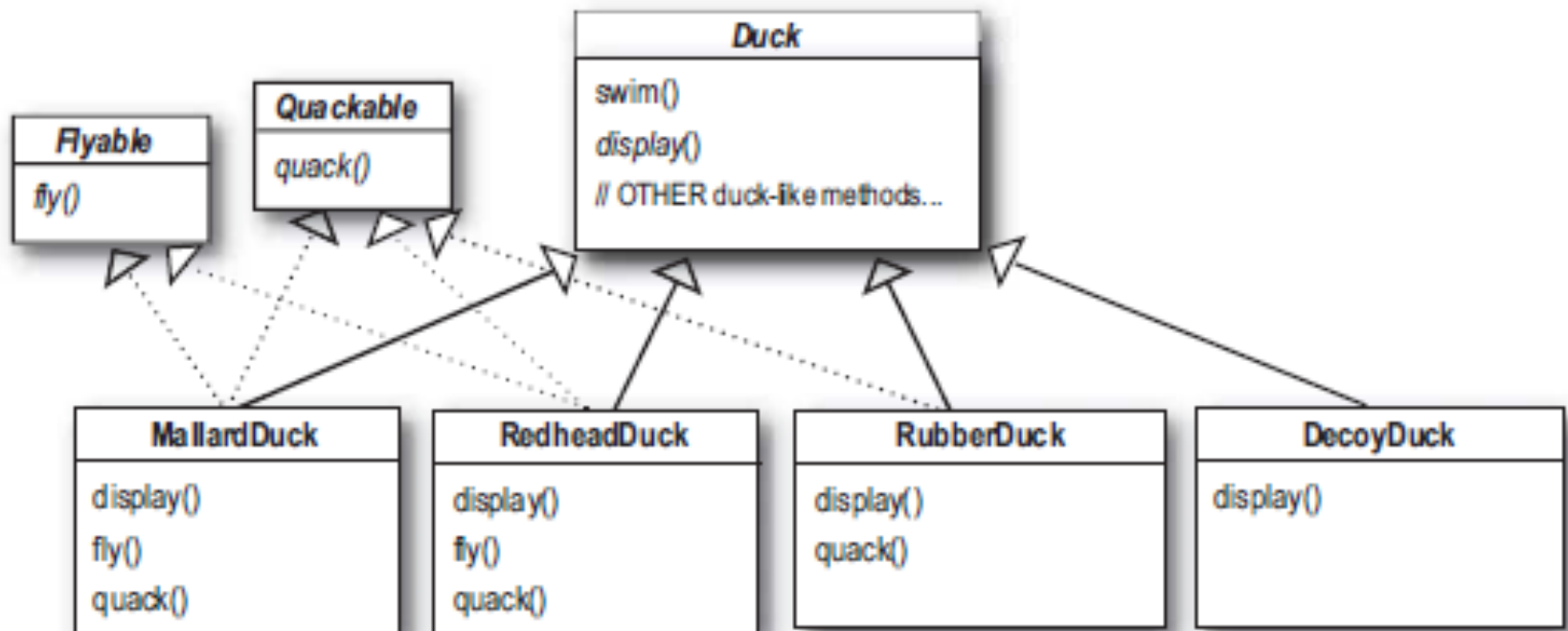


Simulador de Patos

- ¿Cuáles son las desventajas de usar subclases para agregar comportamiento específico a cada pato?
 - El código es duplicado en las subclases
 - Cambios de funcionalidad son difíciles
 - Difícil obtener conocimiento del comportamiento de los patos, cada quien puede hacer lo que quiere
 - Cambios podrían hacer que los patos hagan cosas que no deben

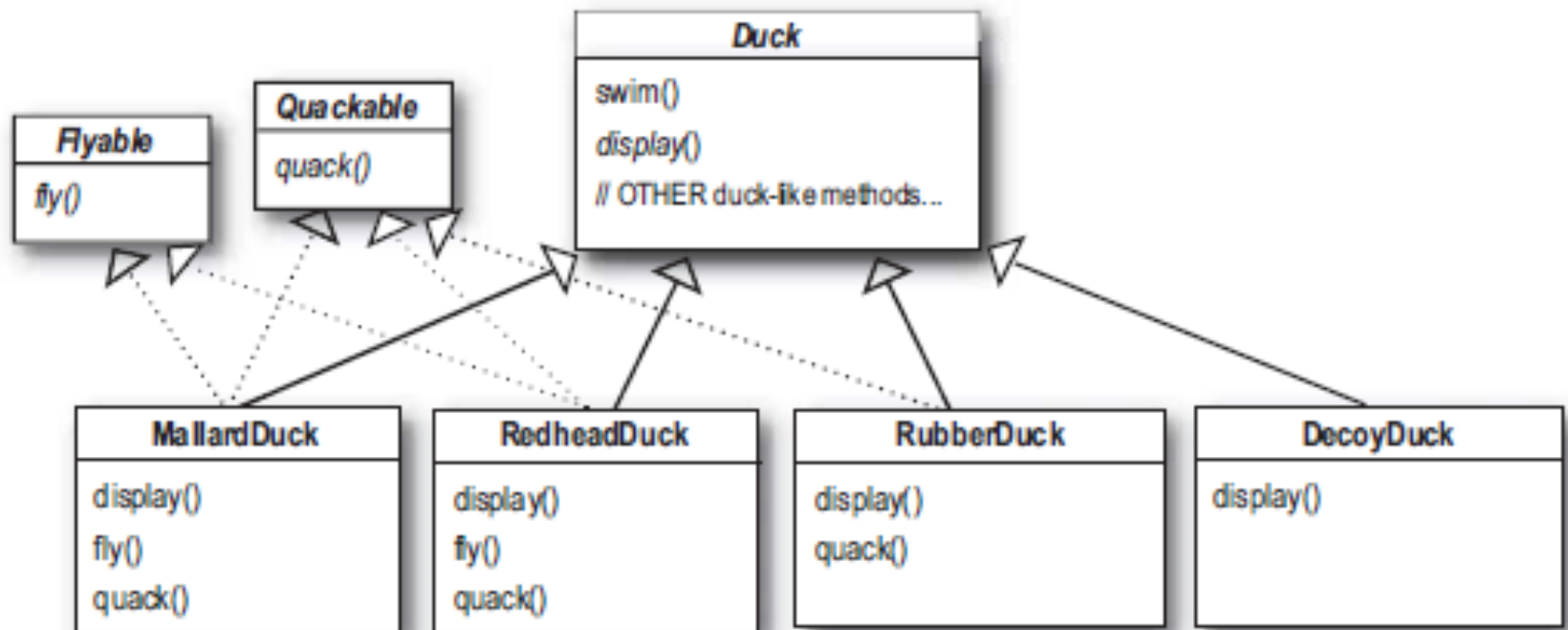
Simulador de Patos

- Plan B- Interfaces.
 - Evitemos tener que programar cada clase pero igual tengo que escribir métodos,



Simulador de Patos

- Plan B- Interfaces.
 - Evitemos tener que programar cada clase pero igual tengo que escribir métodos, **que pasa si hay que modificar los 48 patos del sistema**



Simulador de Patos

- ¿Y entonces no que el POO muy bueno?

Recuerda la contante del software

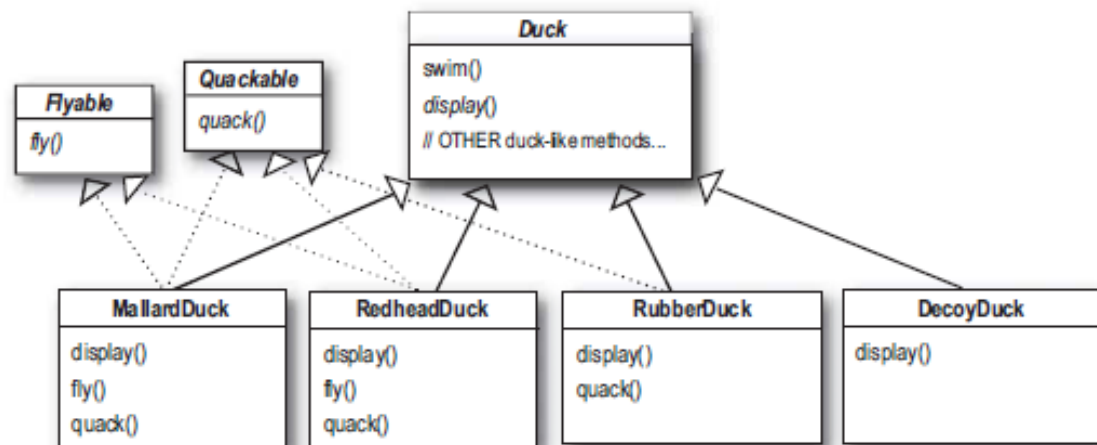
- Has una lista de cosas que han provocado cambios en el código que has hecho.
 - El profe decide que quiere otra cosa u otra funcionalidad
 - Cambio del gestor de base de datos, adquisición de datos en otro formato
 - , , , ,

Recuerda la contante del software

- Has una lista de cosas que han provocado cambios en el código que has hecho.
 - El profe decide que quiere otra cosa u otra funcionalidad
 - Cambio del gestor de base de datos, adquisición de datos en otro formato
 - , , , ,

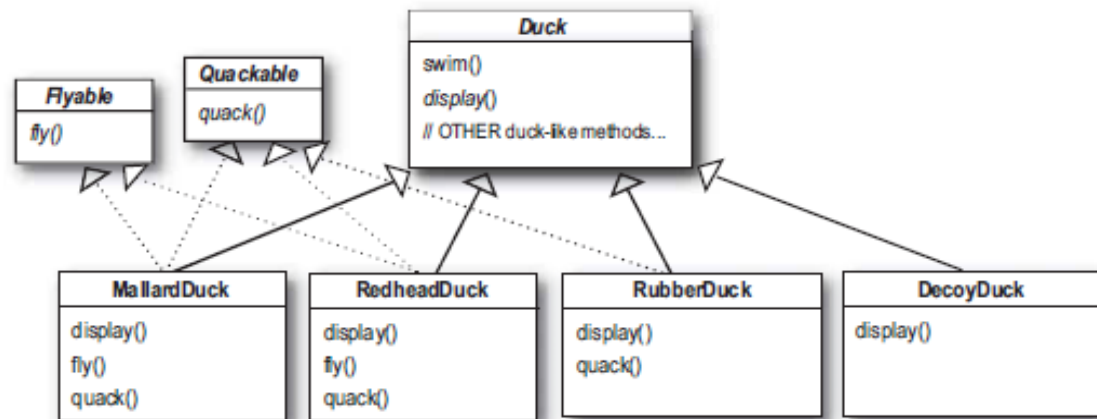
Simulador de Patos

- El problema de la herencia múltiple:
 - Cambios constantes de clases
 - No todos vuelan y hacen quack
 - Modificar clases que se vean afectadas no es fácil
 - Efecto esperado → muchos errores



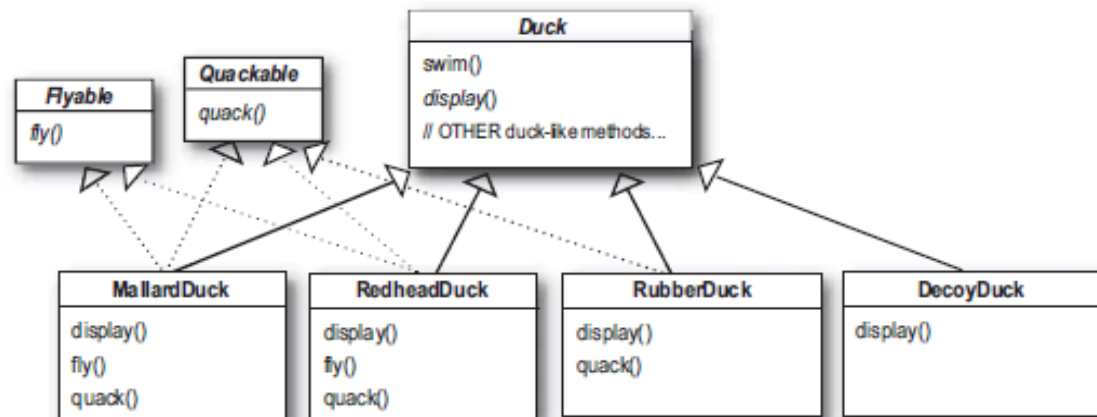
Simulador de Patos

- **Principio de diseño:** Estrategia, encapsula los eventos que cambian y sepáralos del resto de la aplicación.
 - Asegurar que no nos afecten
 - Menor número de consecuencias por error en código
 - -mayor flexibilidad



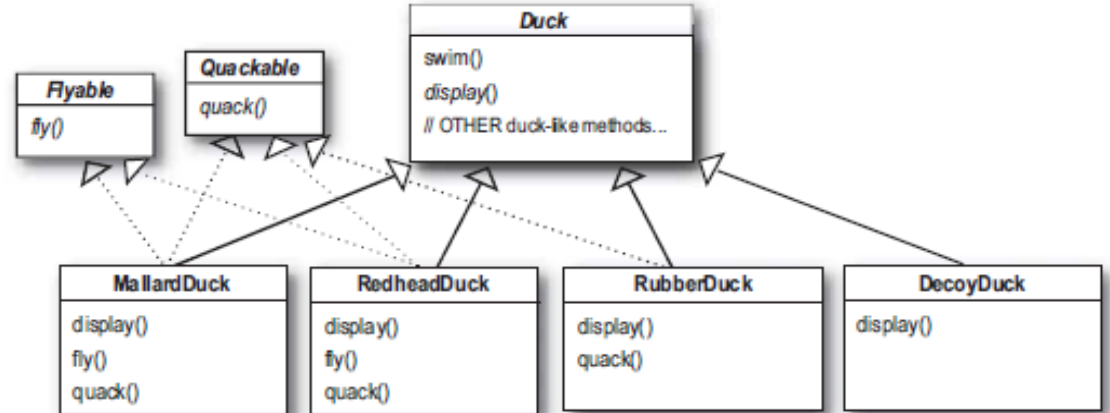
Simulador de Patos

- ¿Qué eventos nos están afectando?



Simulador de Patos

- ¿Qué eventos nos están afectando?
 - fly()
 - Quack()

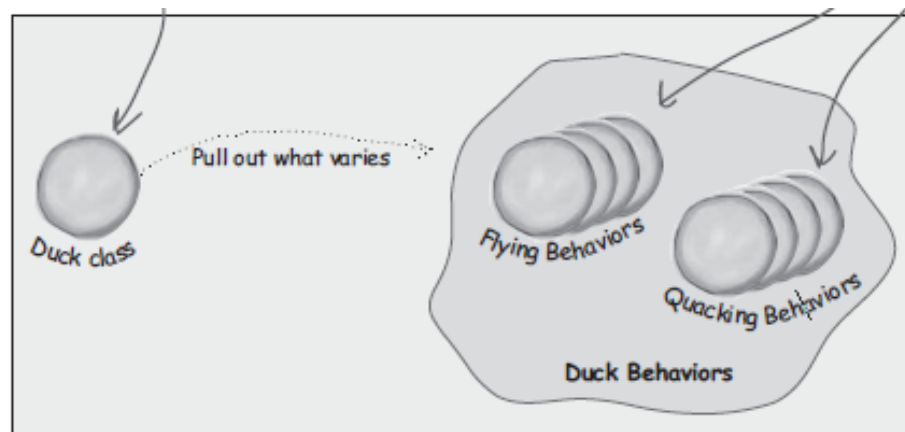


Simulador de Patos

- Dejemos la clase Duck intacta y separemos estos comportamientos
 - ¿Cómo?

Simulador de Patos

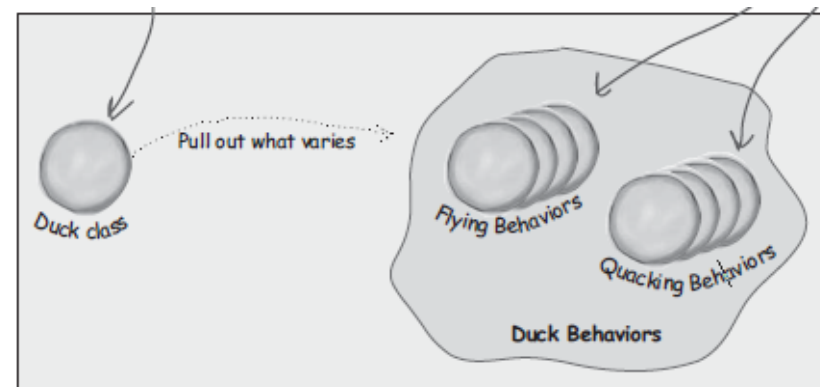
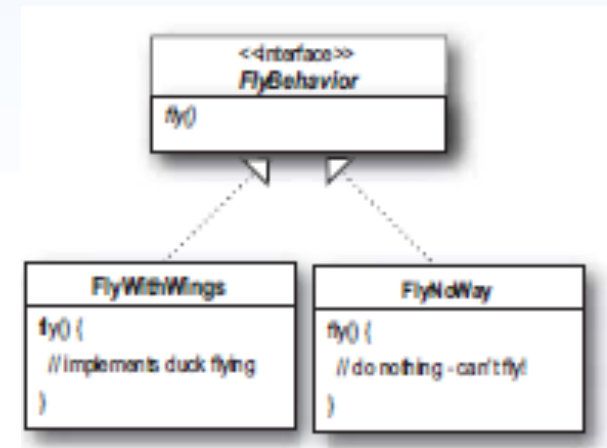
- Dejemos la clase Duck intacta y separemos estos comportamientos
 - ¿Cómo?
 - **Recuerda que cada comportamiento es único y como tal vamos a requerir clases que lo defina**



DISEÑAR EL COMPORTAMIENTO DEL PATO

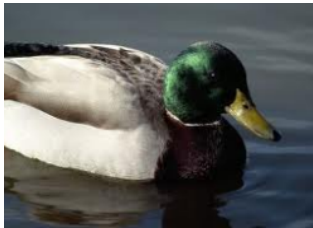
Diseñar el comportamiento del Pato

- El comportamiento del pato reside en una clase separada
 - Una clase que implementa el comportamiento de una forma particular
 - Volar
 - No hacer nada
- El pato no sabe nada de la implementación de sus comportamientos



Diseñar el comportamiento del Pato

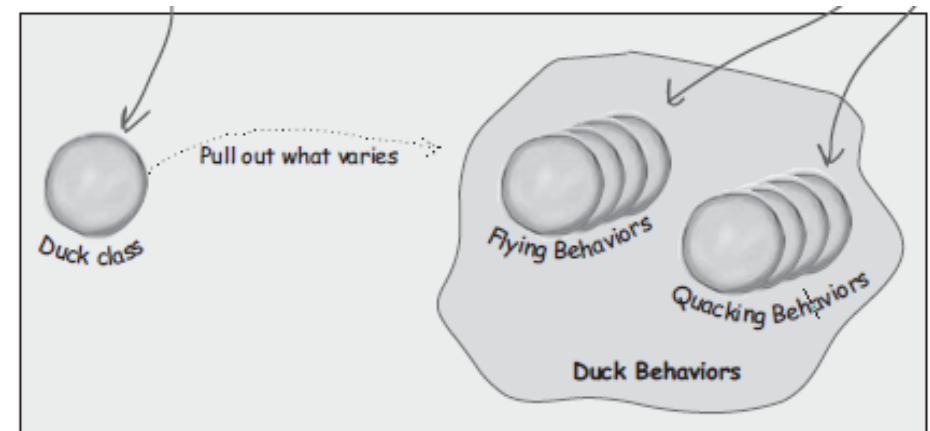
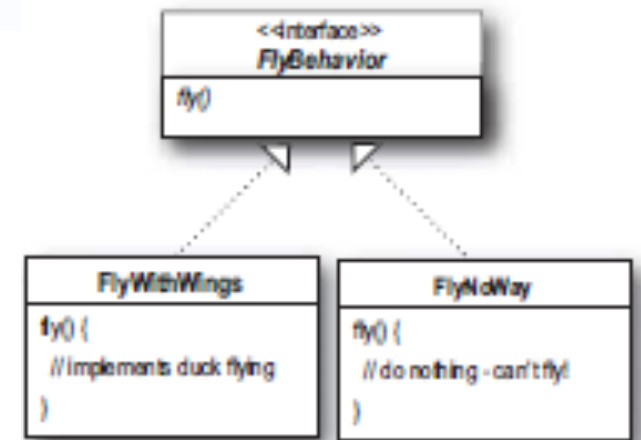
- Queremos agregar comportamientos a la clase Pato Mallard y especificar un tipo de vuelo



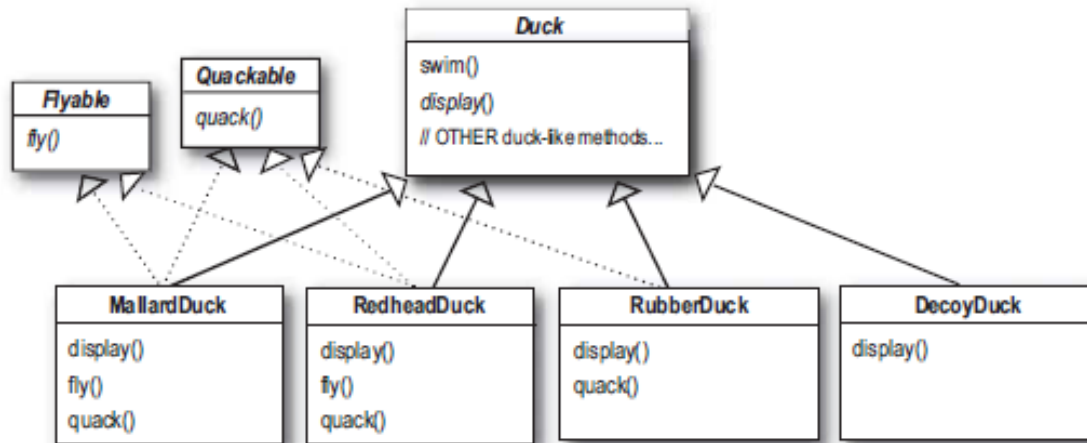
- El pato debe tener el método `setFlyBehavior()` de tal forma que dinámicamente pueda modificar su forma de volar
- **Principio de Diseño 2:** programa en las interfaces no en la implementación de una clase

Diseñar el comportamiento del Pato

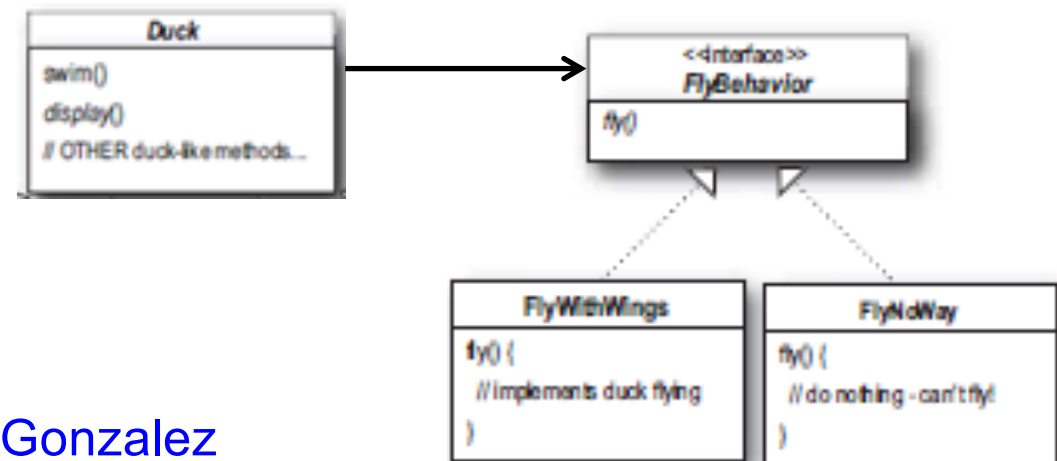
- Se puede usar una Interfaz para programar cada comportamiento
- Cada implementación de un comportamiento implementará la interfaz
- La clase Duck ya no implementa el vuelo ni el quack



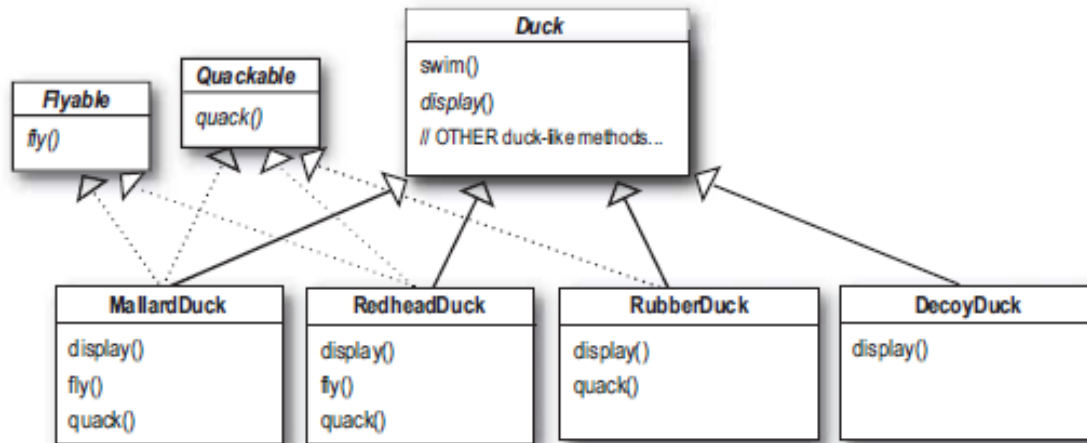
Diseñar el comportamiento del Pato



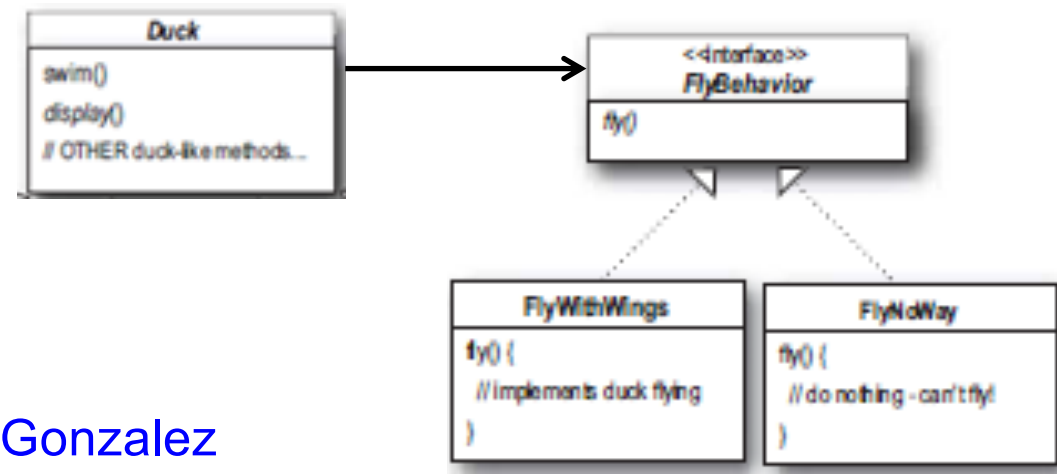
El comportamiento no se escribe en la clase Duck



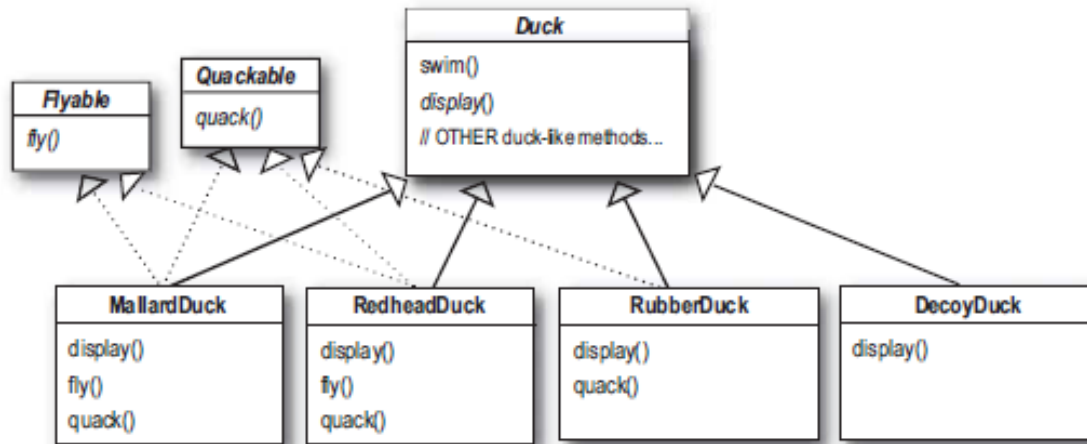
Diseñar el comportamiento del Pato



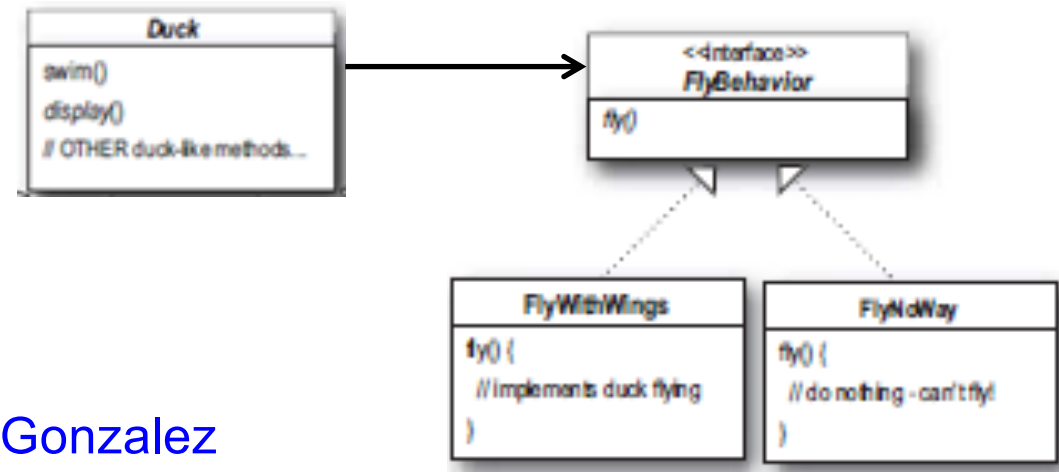
- Cambiar aquí es hacer más código y es difícil ubicarlo



Diseñar el comportamiento del Pato

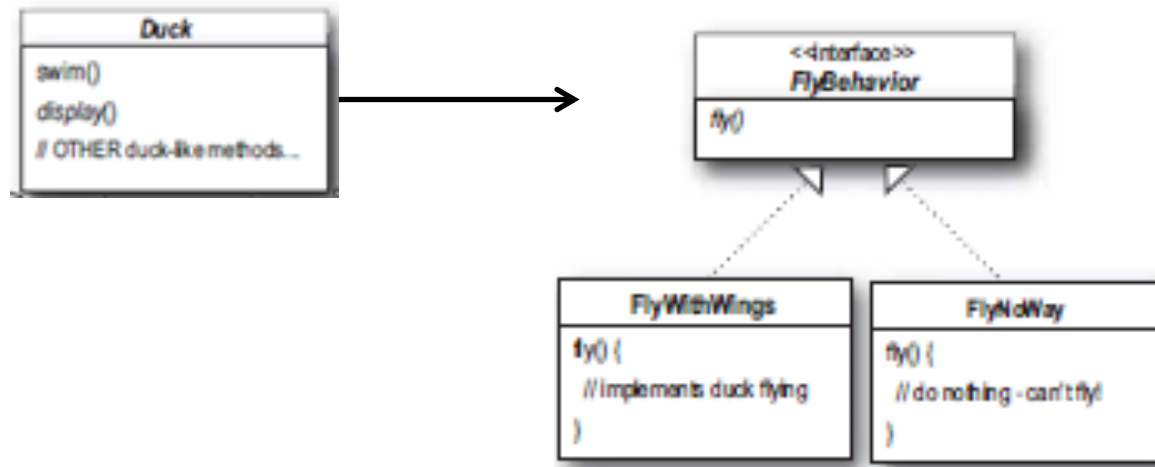


- Ahora es más fácil de rastrear donde debo hacer cambios



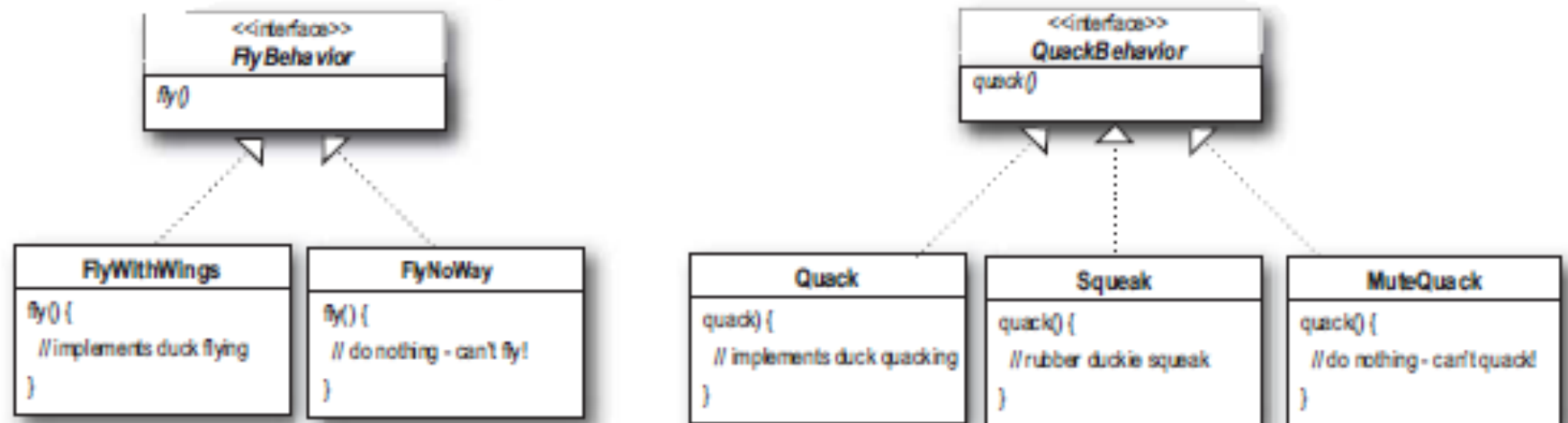
Diseñar el comportamiento del Pato

- El objetivo es encapsular el comportamiento,
- Explotar el polimorfismo programando supertipos (FlyBehaviour) en lugar del objeto usado en tiempo de ejecución (Duck)



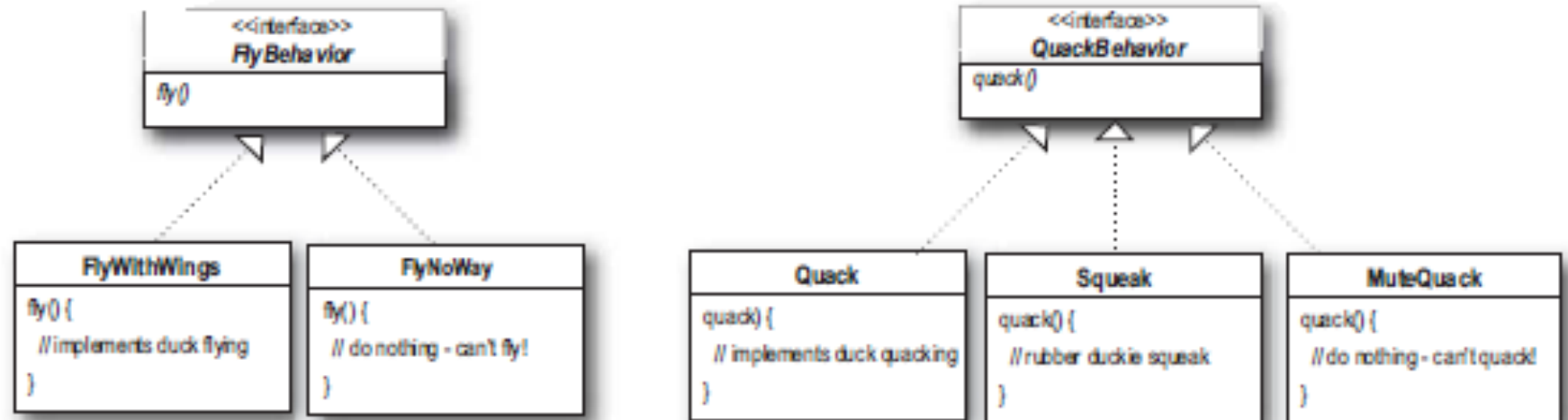
Implementación de cada comportamiento

Interfaces que todas las subclases deben implementar



Implementación de cada comportamiento

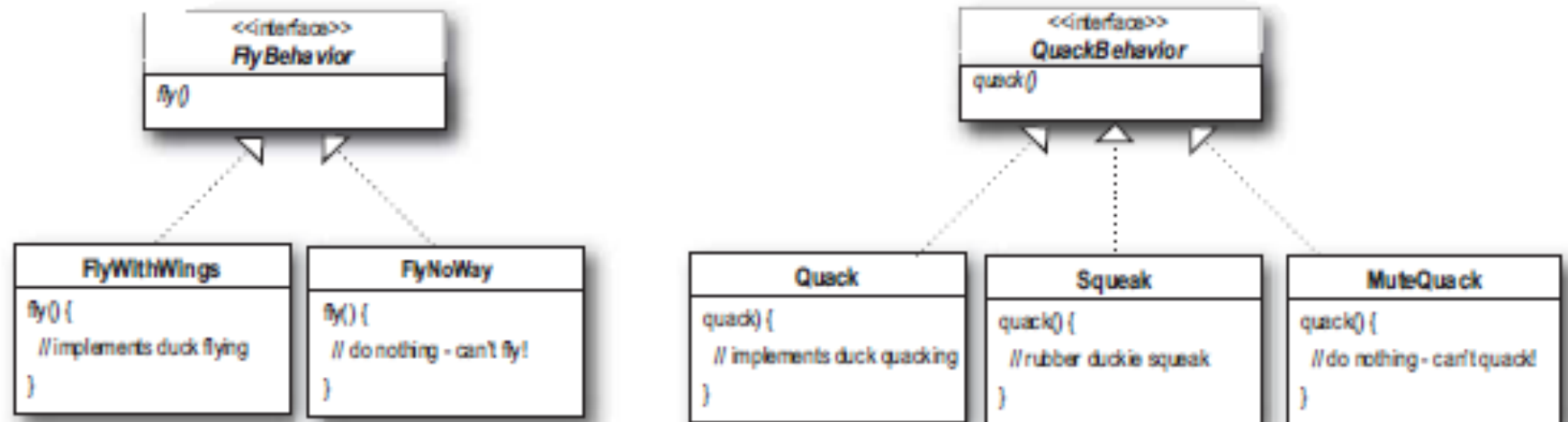
Interfaces que todas las subclases deben implementar



Patos con alas y sin alas

Implementación de cada comportamiento

Interfaces que todas las subclases deben implementar



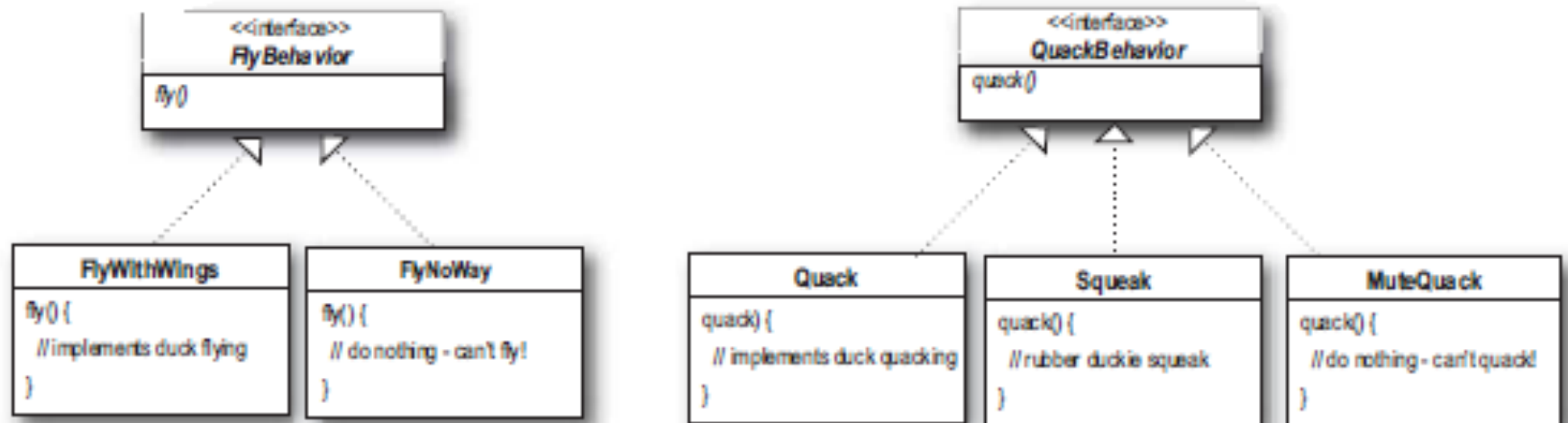
Patos con alas y sin alas

Los que hacen quack, chillan, no hacen nada

Implementación de cada comportamiento

Ahora todos los patos podrán usar esto ya no se encuentra escondido en la clase particular de algún pato

Interfaces que todas las subclases deben implementar



Patos con alas y sin alas

Los que hacen quack, chillan, no hacen nada

Preguntas Simples

- ¿Qué hacemos si queremos un pato con cohete propulsor?

Preguntas Simples

- ¿Qué hacemos si queremos un pato con cohete propulsor?
 - **Agregar la clase flyRocketPowered()**

Preguntas Simples

- ¿Se imaginan una clase que requiera el comportamiento `quack()` que no sea pato?

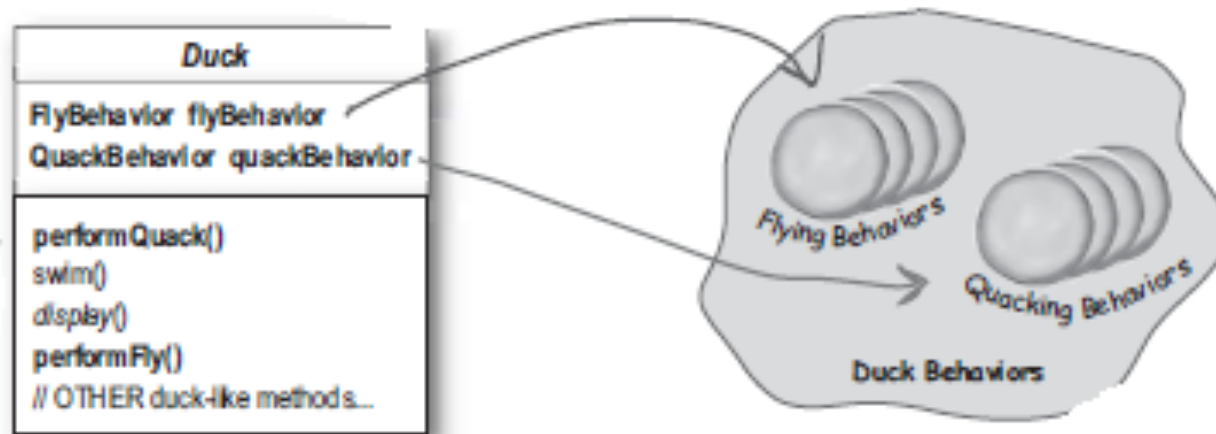
Preguntas Simples

- ¿Se imaginan una clase que requiera el comportamiento `quack()` que no sea pato?
 - **Sonido de llamada en mi celular**

INTEGRANDO EL COMPORTAMIENTO AL PATO

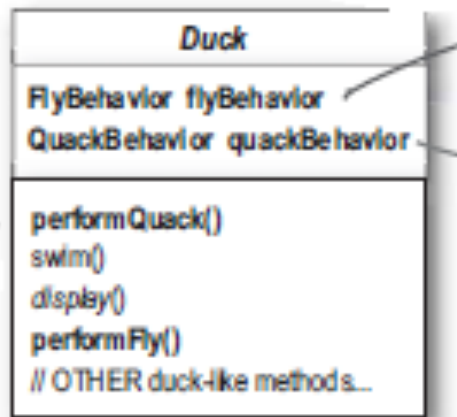
Integrando el comportamiento al Pato

- Para que el pato haga quack y vuele debemos agregar variables que lo permitan



Integrando el comportamiento al Pato

- Para que el pato haga quack y vuele debemos agregar variables que lo permitan

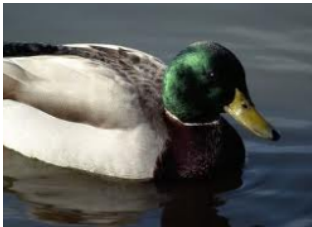


```
public class Duck {
    QuackBehavior quackBehavior;
    // more

    public void performQuack() {
        quackBehavior.quack();
    }
}
```

Integrando el comportamiento al Pato

- Ahora veamos como inicializar un objeto



```
public class Duck {  
    QuackBehavior quackBehavior; 4  
    // more  
  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
}
```

```
public class MallardDuck extends Duck {  
  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```

Ejercicio hacer el código de los patos

- Simplemente mandar a imprimir a pantalla los mensajes
 - Para el método fly
 - “Vuelo”
 - “No vuelo”
 - Para el metodo quack
 - “quack”
 - “squeeze”
 - “no hago ruido”
- Hacer esto para los 4 patos del ejemplo
 - Decoy
 - Rubber
 - Redhead
 - Mallard

```
public abstract class Duck {  
  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
    public Duck() {  
    }  
  
    public abstract void display();  
  
    public void performFly() {  
        flyBehavior.fly();  
    }  
  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
  
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }  
}
```

```
public interface FlyBehavior {  
    public void fly();  
}
```

```
public class FlyWithWings implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying!!");  
    }  
}
```

```
public class FlyNoWay implements FlyBehavior {  
    public void fly() {  
        System.out.println("I can't fly");  
    }  
}
```

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

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

```
public class MuteQuack implements QuackBehavior {  
    public void quack() {  
        System.out.println("<< Silence >>");  
    }  
}
```

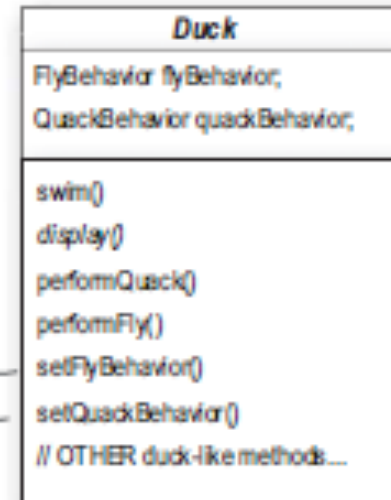
```
public class Squeak implements QuackBehavior {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}
```

```
public class MiniDuckSimulator {  
    public static void main(String[] args) {  
        Duck mallard = new MallardDuck();  
        mallard.performQuack();  
        mallard.performFly();  
    }  
}
```

P

Definamos el comportamiento de forma automática

```
public void setFlyBehavior(FlyBehavior fb) {  
    flyBehavior = fb;  
}  
  
public void setQuackBehavior(QuackBehavior qb) {  
    quackBehavior = qb;  
}
```



Vamos a crear un pato que no vuela

```
public class ModelDuck extends Duck {  
    public ModelDuck() {  
        flyBehavior = new FlyNoWay();  
        quackBehavior = new Quack();  
    }  
  
    public void display() {  
        System.out.println("I'm a model duck");  
    }  
}
```

Definamos ahora un pato cohete

```
public class FlyRocketPowered implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying with a rocket!");  
    }  
}
```

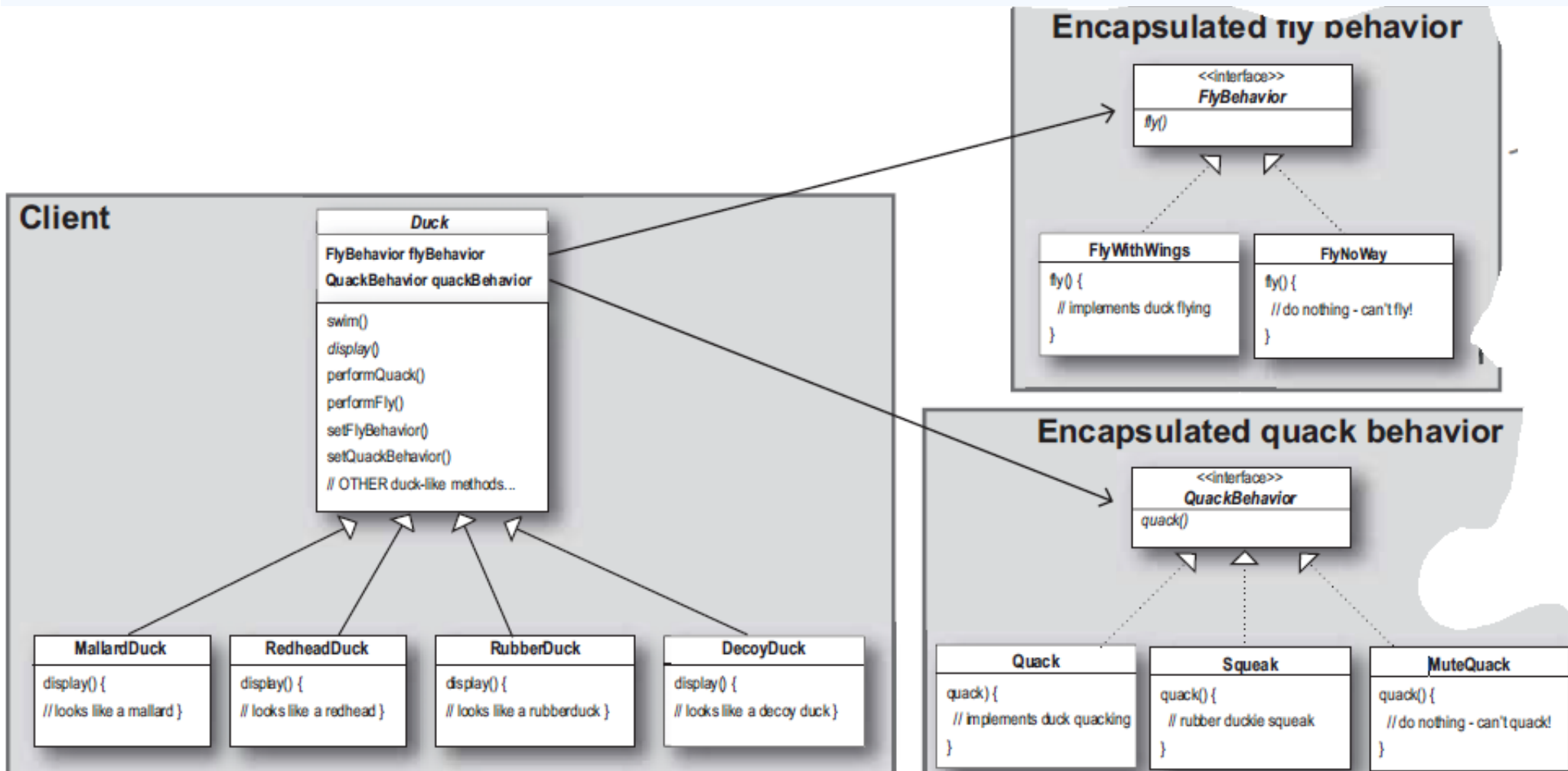
Hagamos un nuevo main

```
public class MiniDuckSimulator {  
    public static void main(String[] args) {  
        Duck mallard = new MallardDuck();  
        mallard.performQuack();  
        mallard.performFly();
```

```
        Duck model = new ModelDuck();  
        model.performFly();  
        model.setFlyBehavior(new FlyRocketPowered());  
        model.performFly();
```

```
    }  
}
```

La encapsulación

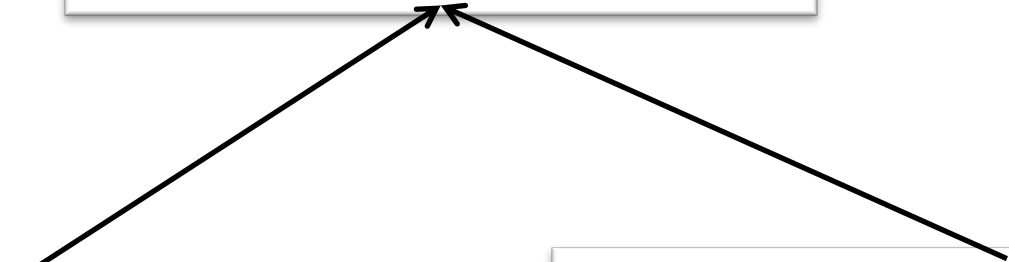


Clase Duck

```
1 public abstract class Duck {
2     FlyBehavior flyBehavior;
3     QuackBehavior quackBehavior;
4
5     //Unico metodo a sobre escribir en todas las subclases
6     public Duck () {
7     }
8
9     public abstract void display ();
10
11     //Dependiendo del constructor de las subclases será el tipo de vuelo
12     public void performFly ()
13     {
14         flyBehavior.fly();
15     }
16
17     //Dependiendo del constructor de las subclases será el tipo de graznido
18     public void performQuack() {
19         quackBehavior.quack();
20     }
21
22     //Este método siempre es el mismo para todos
23     public void swim () {
24         System.out.println("All ducks float, even decoys!");
25     }
26
27     public void showDuck () {
28
29         display();
30         performFly();
31         performQuack();
32         swim();
33     }
34
35 }
```

Interfaz de Vuelo

```
public interface FlyBahavior {  
  
    public void fly();  
  
}
```



```
public class FlyNoWay implements FlyBahavior{  
  
    @Override  
    public void fly() {  
        System.out.println("I can't fly");  
    }  
  
}
```

```
public class FlyWithWings implements FlyBahavior {  
  
    @Override  
    public void fly() {  
        System.out.println("I'm flying");  
    }  
  
}
```

Interfaz Graznido

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

```
public class Quack implements QuackBehavior{  
  
    @Override  
    public void quack() {  
        System.out.println("Quack");  
    }  
  
}
```

```
public class MuteQuack implements QuackBehavior{  
  
    @Override  
    public void quack() {  
        System.out.println("<<Silence>>");  
    }  
  
}
```

```
public class Squeak implements QuackBehavior{  
  
    @Override  
    public void quack() {  
        System.out.println("Squeak");  
    }  
  
}
```


Prueba

```
public class DuckTest {  
  
    public static void main (String args [])  
    {  
        Duck mallardDuck, redHead, decoyDuck, rubberDuck;  
  
        mallardDuck = new MallardDuck ();  
        mallardDuck.showDuck();  
  
        redHead = new RedHeadDuck();  
        redHead.showDuck();  
  
        decoyDuck = new DecoyDuck();  
        decoyDuck.showDuck();  
  
        rubberDuck = new RubberDuck();  
        rubberDuck.showDuck();  
  
        System.exit(0);  
  
    }  
}
```

```
run:  
I'm a real Mallard duck  
I'm flying  
Quack  
All ducks float, even decoys!  
I'm a real Red Head duck  
I'm flying  
Quack  
All ducks float, even decoys!  
I'm a simply Decoy duck  
I can't fly  
<<Silence>>  
All ducks float, even decoys!  
I'm a pretty Rubber duck  
I can't fly  
Squeak  
All ducks float, even decoys!
```

Definamos el comportamiento de forma automática

```
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    //Unico metodo a sobre escribir en todas las subclases
    public Duck () {...}

    public abstract void display ();

    //Dependiendo del constructor de las subclases será el tipo de vuelo
    public void performFly ()
    {...}

    //Dependiendo del constructor de las subclases será el tipo de graznido
    public void performQuack() {...}

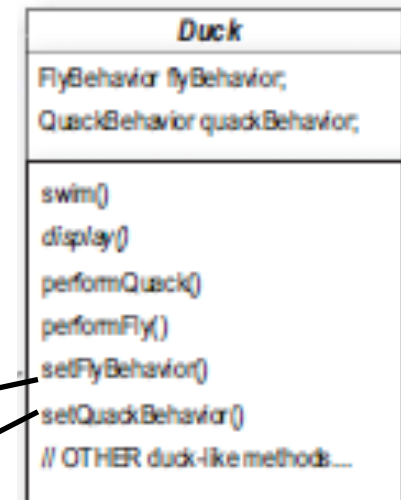
    //Este método siempre es el mismo para todos
    public void swim () {...}
    public void showDuck () {...}

    public void setFlyBehavior(FlyBehavior fb){

        flyBehavior =fb;
    }

    public void setQuackBehavior(QuackBehavior qb){

        quackBehavior =qb;
    }
}
```



Vamos a crear un pato que no vuela

- Llámenle ModelDuck

Vamos a crear un pato que no vuela

- Llámenle ModelDuck

```
public class ModelDuck extends Duck{

    public ModelDuck ()
    {
        flyBehavior = new FlyNoWay ();
        quackBehavior = new Quack ();
    }

    public void display () {
        System.out.println ("I'm a model Duck");
    }

}
```

Definamos ahora un comportamiento de vuelo con cohete

- Llámenle FlyRocketPowered

```
public class ModelDuck extends Duck{

    public ModelDuck ()
    {
        flyBehavior = new FlyNoWay ();
        quackBehavior = new Quack ();
    }

    public void display () {
        System.out.println ("I'm a model Duck");
    }

}
```

Definamos ahora un comportamiento de vuelo con cohete

- Llámenle FlyRocketPowered

Definamos ahora un comportamiento de vuelo con cohete

- Lláménle FlyRocketPowered

```
public class FlyRocketPowered implements FlyBehavior{  
  
    public void fly () {  
  
        System.out.println("I'm flying with a rocket!");  
    }  
}
```

Modifiquemos el main

```
public class DuckTest {  
  
    public static void main (String args [])  
    {  
        Duck mallardDuck, redHead, decoyDuck, rubberDuck;  
  
        mallardDuck = new MallardDuck ();  
        mallardDuck.showDuck();  
  
        redHead = new RedHeadDuck();  
        redHead.showDuck();  
  
        decoyDuck = new DecoyDuck();  
        decoyDuck.showDuck();  
  
        rubberDuck = new RubberDuck();  
        rubberDuck.showDuck();  
  
        Duck model = new ModelDuck();  
        model.performFly();  
        model.setFlyBehavior(new FlyRocketPowered());  
        model.performFly();  
  
        System.exit(0);  
    }  
}
```

```
run:  
I can't fly  
I'm flying with a rocket!
```


Felicidades

- Ya hemos hecho nuestro primer patrón
 - **Strategy.** Define una familia de algoritmos (implementaciones de la interfaz), encapsula cada uno, y los hace intercambiables (setters). El patron hace que la estrategia hace que los algoritmos se adapten independiente de sus clientes usándolos
 - Gracias al patrón rehicimos la clase y ahora nuestro código esta listo para crecer y ser usado de diferentes formas

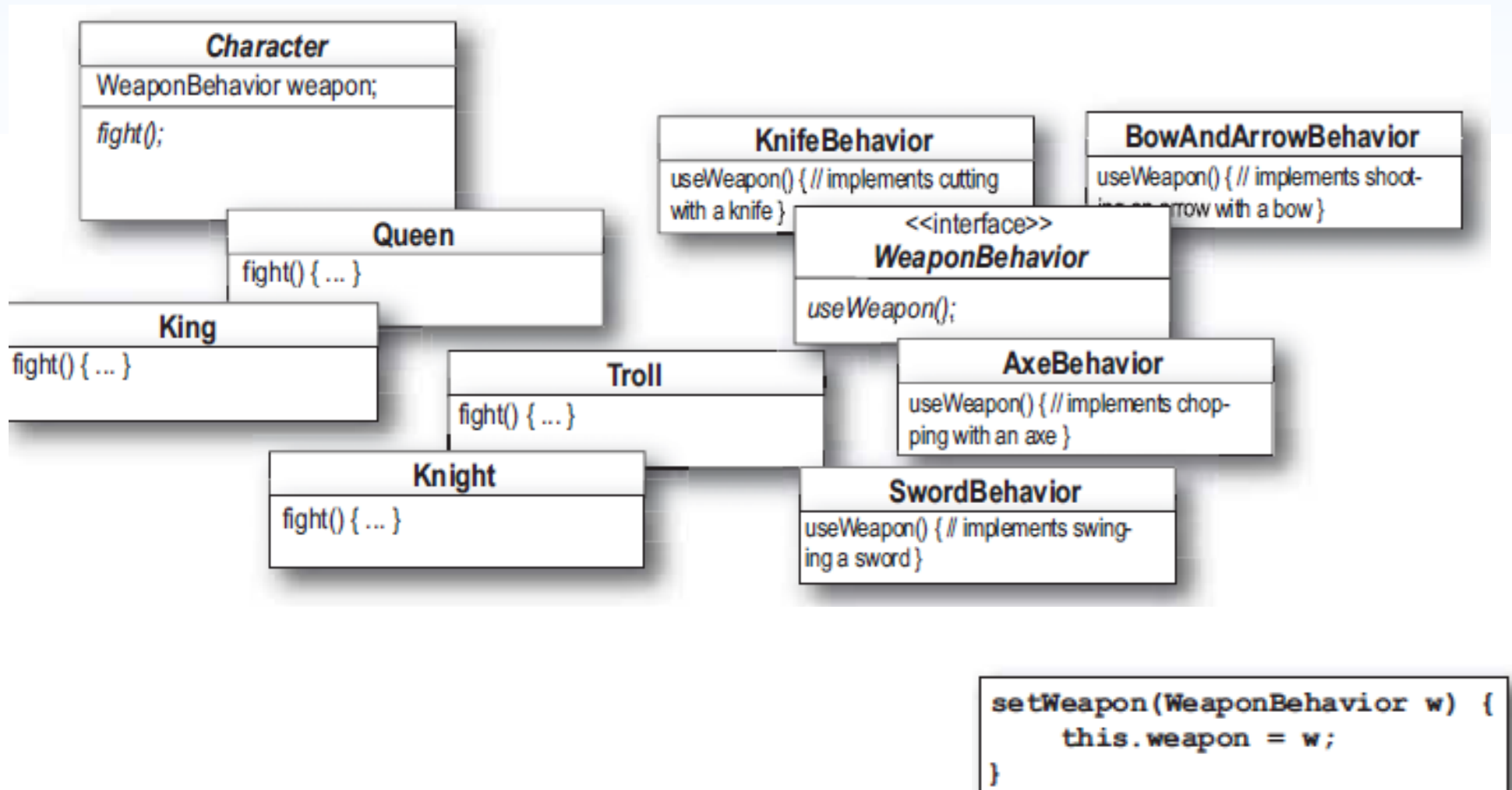
**Y ADEMÁS DE LOS PATOS
QUIÉN USA LA ESTRATEGIA**

Ejercicio – Organiza el desorden

- Clases e interfaces de un juego de acción. Tenemos clases de personajes del juego así como clases de comportamientos con armas que se pueden usar en el juego. Cada personaje puede usar un arma a la vez pero puede usar diferentes armas durante el juego.
1. Organiza las clases
 2. Identifica la clase abstracta, la interface y las ocho clases
 3. Usa las relaciones adecuadas
 - Herencia, asociación, implementa
 4. Quien debe tener el método

```
setWeapon(WeaponBehavior w) {  
    this.weapon = w;  
}
```

Ejercicio – Organiza el desorden



Ahora trata de definir comportamientos de movimiento

- De qué forma se puede mover un personaje:
 - Camina
 - Corre
 - Arrastra
 - Cuncillas

Felicidades

- Ya hemos hecho nuestro primer patrón
 - **Strategy.** Define una familia de algoritmos (implementaciones de la interfaz), encapsula cada uno, y los hace intercambiables (setters). La estrategia hace que los algoritmos se adapten independiente de sus clientes usándolos
 - Gracias al patrón rehicimos la clase y ahora nuestro código esta listo para crecer y ser usado de diferentes formas