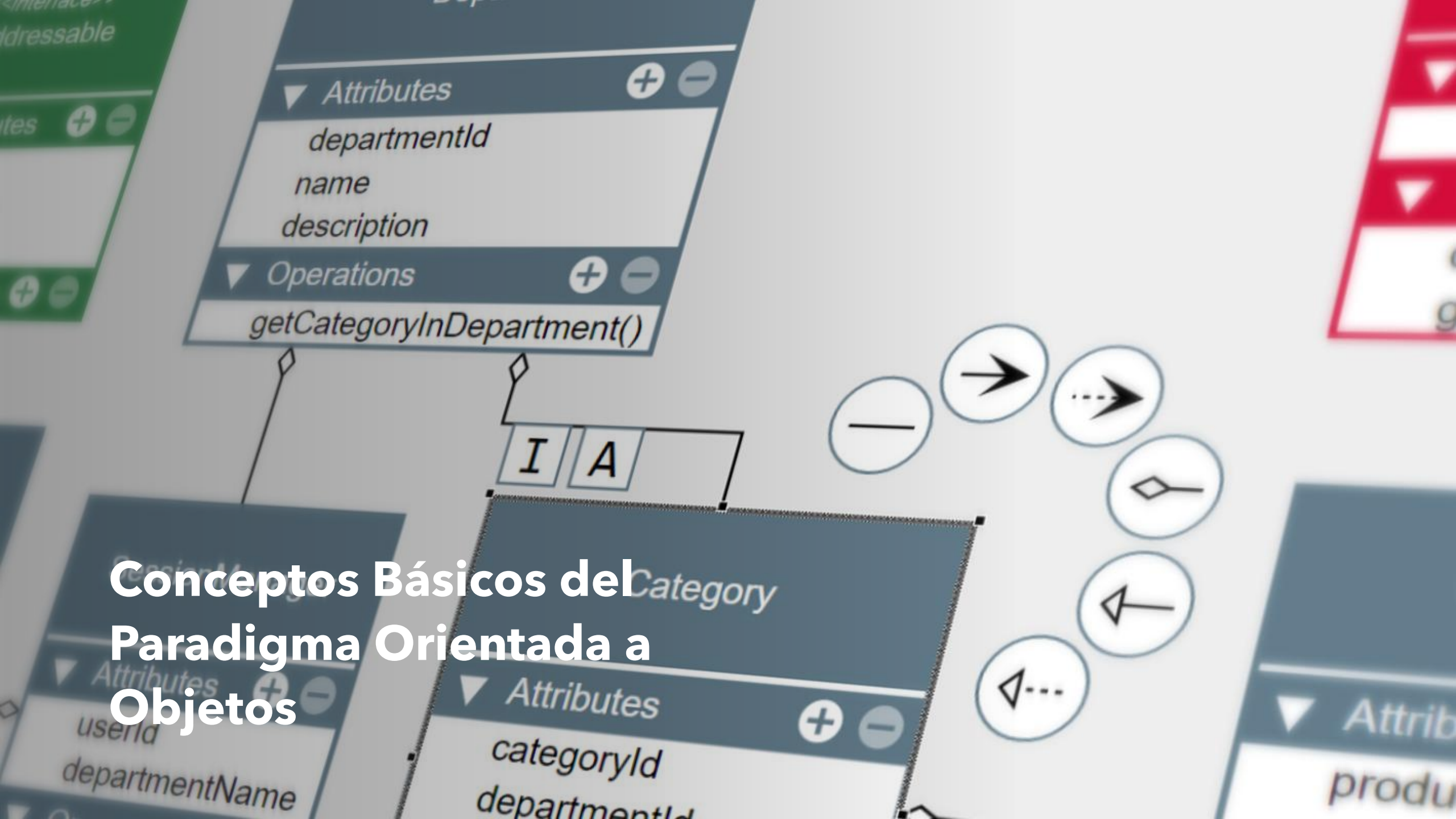
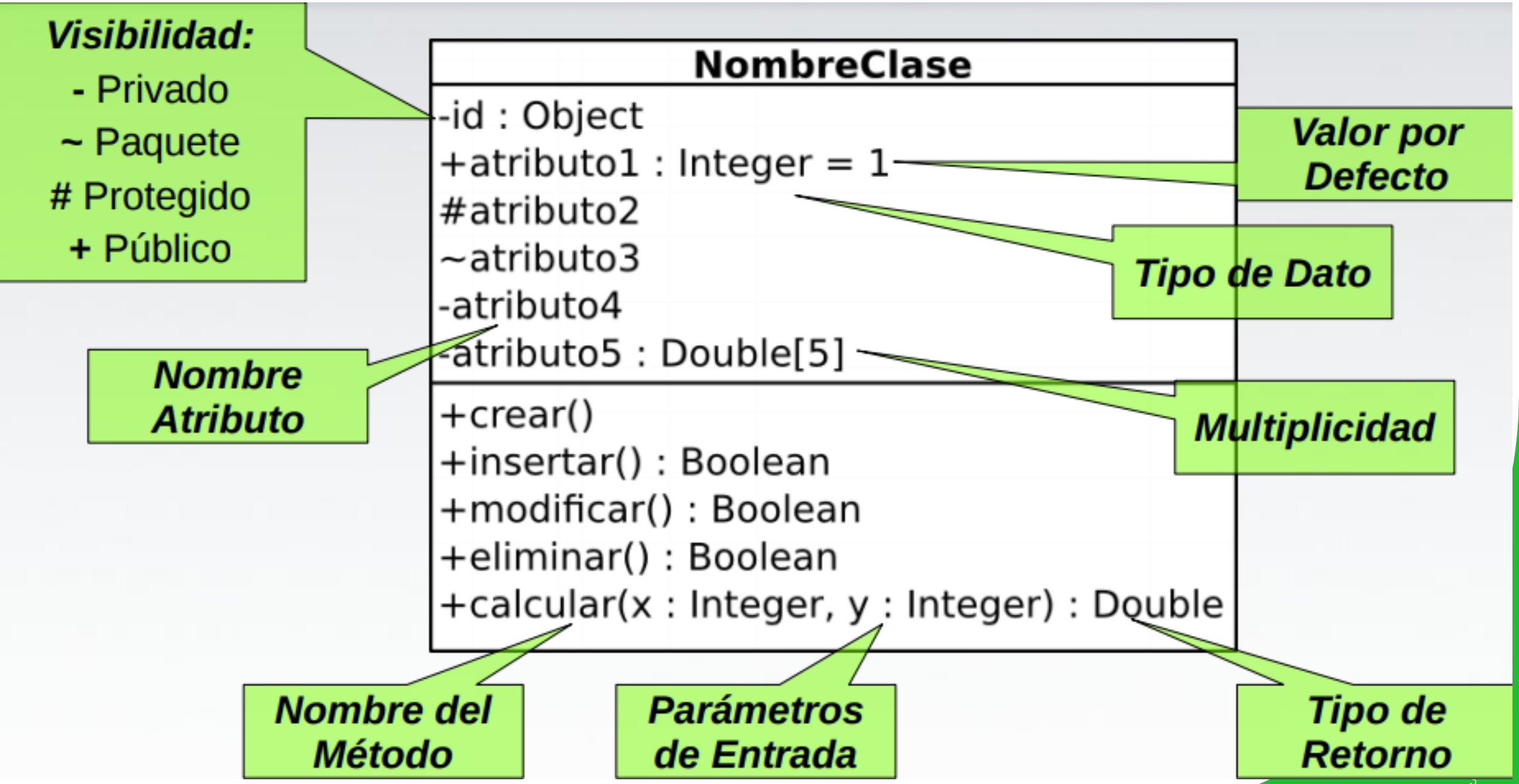
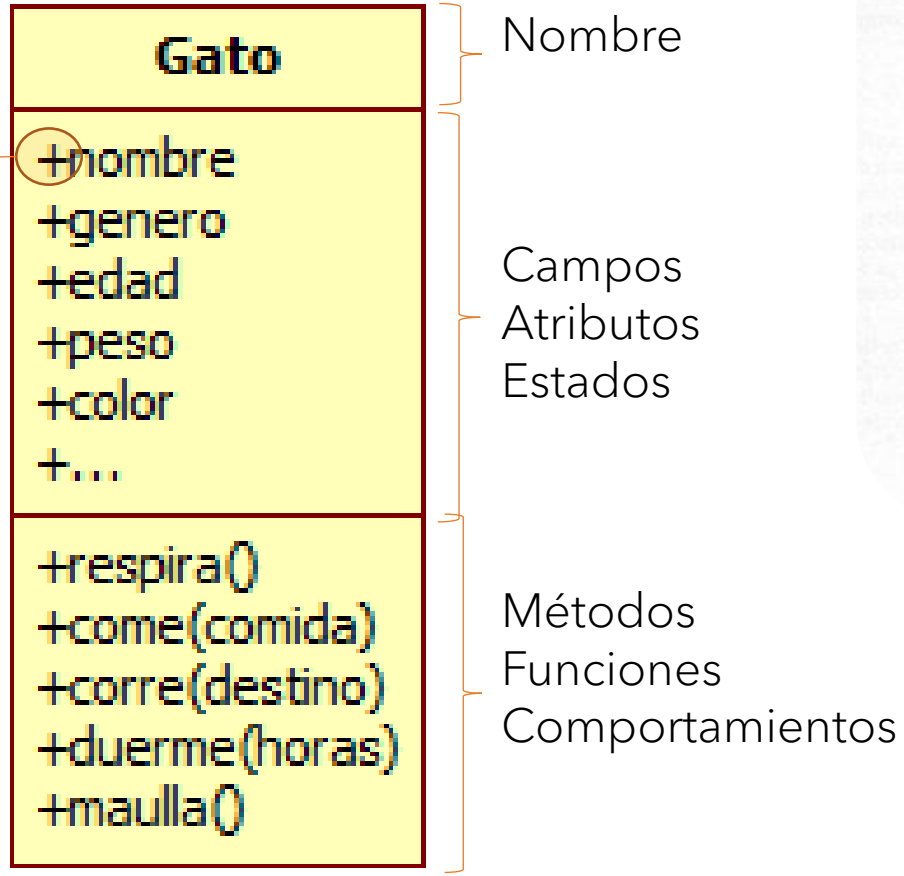
Introducción a los patrones



Conceptos Básicos del Paradigma Orientada a Objetos


Visibilidad:
- Privado
~ Paquete
# Protegido
+ Público
NombreClase
-id : Object
+atributo1 : Integer = 1
#atributo2
~atributo3
-atributo4
-atributo5 : Double[5]
+crear()
+insertar() : Boolean
+modificar() : Boolean
+eliminar() : Boolean
+calcular(x : Integer, y : Integer) : Double
Valor por Defecto
Tipo de Dato
Nombre Atributo
Multiplicidad
Nombre del Método
Parámetros de Entrada
Tipo de Retorno

La clase gato

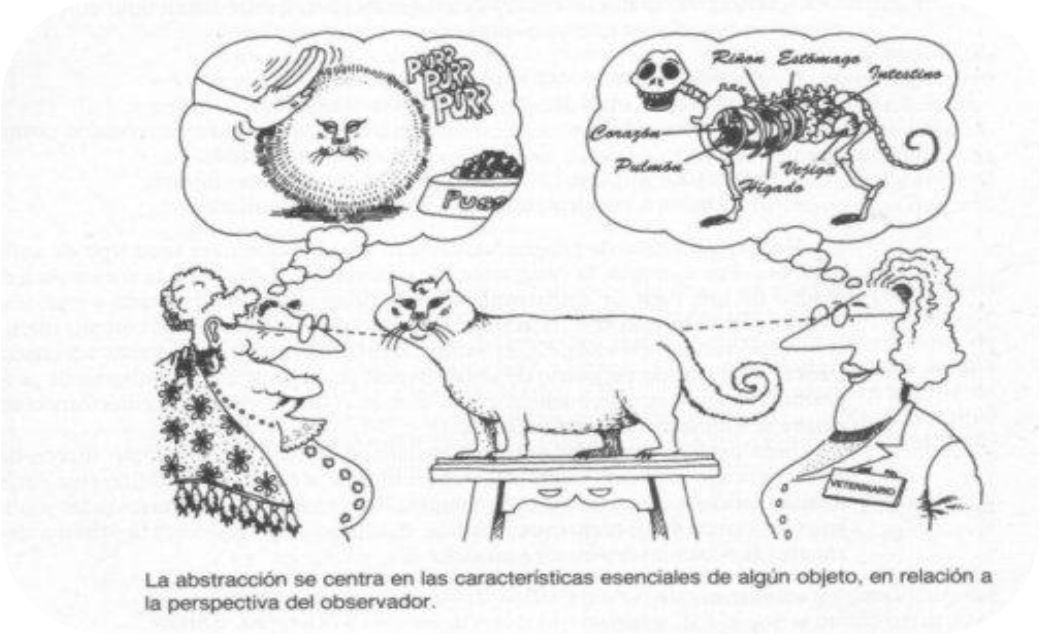


Visibilidad
+ publico

Nombre

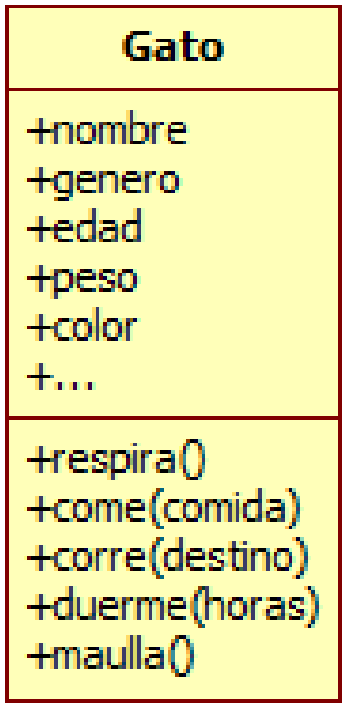
Campos
Atributos
Estados

Métodos
Funciones
Comportamientos



La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

Los objetos son instancias concretas de las clases



Con Botas: Gato

nombre: con Botas
genero: Masculino
edad: 23
peso: 5.6
color: naranja

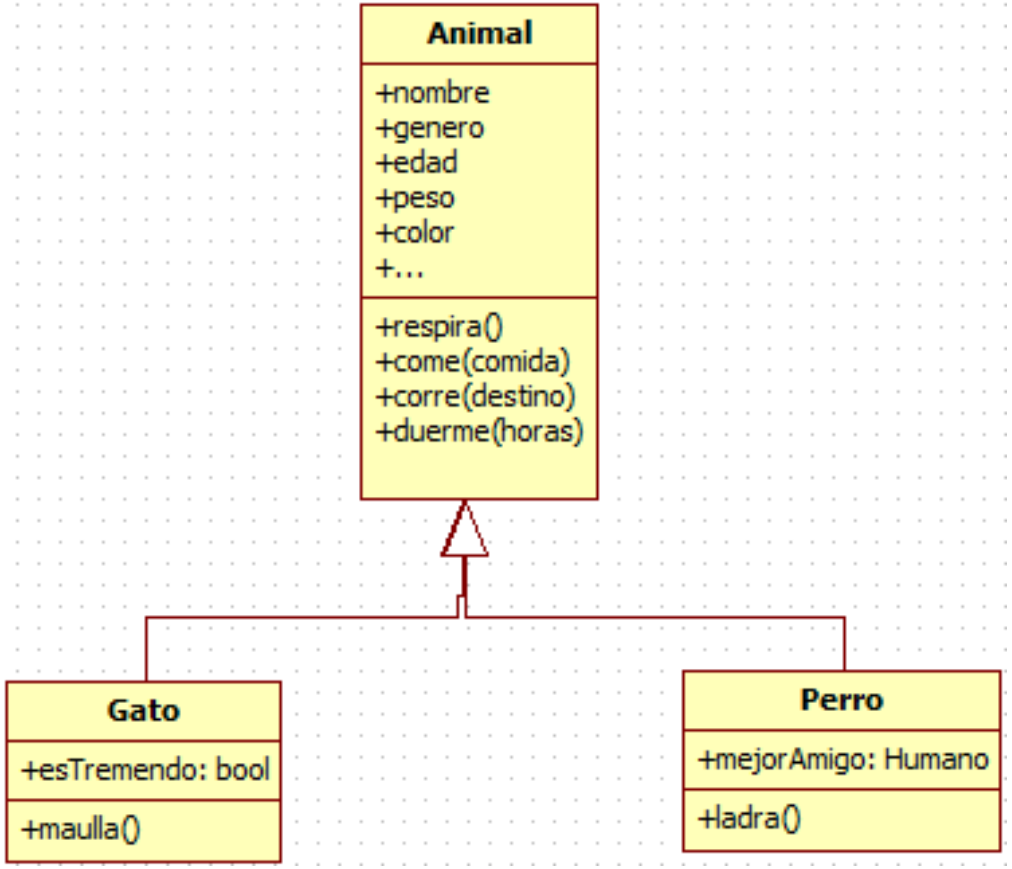
Garfield: Gato

nombre: Garfield
genero: Masculino
edad: 8
peso: 9
color: naranja

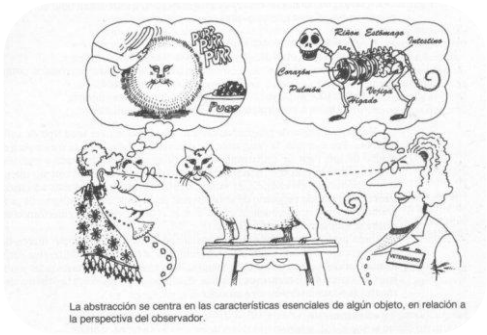
La generalización

- La clase padre (Animal), se denomina superclase.
- Sus hijos son subclases. Las subclases heredan el estado y el comportamiento de su padre, definiendo solo atributos o comportamientos que difieren.

Por lo tanto, la clase Gato tendría el método de maullido, y la clase Pedro el método de ladra.



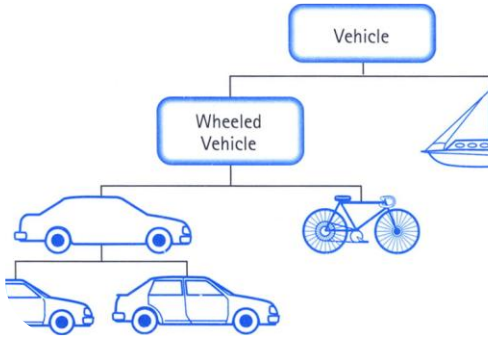
El paradigma orientado a objetos se basa en cuatro pilares



Abstracción

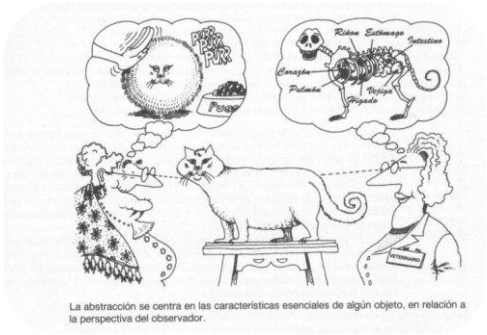
Polimorfismo

Encapsulación



Herencia

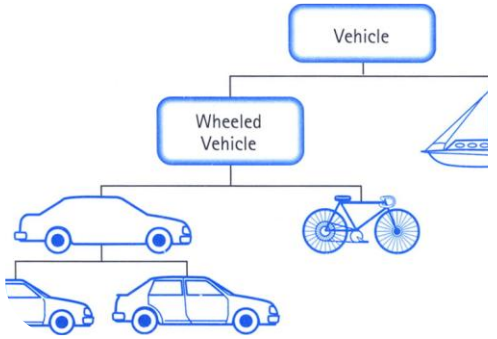
El paradigma orientado a objetos se basa en cuatro pilares



Abstracción

Polimorfismo

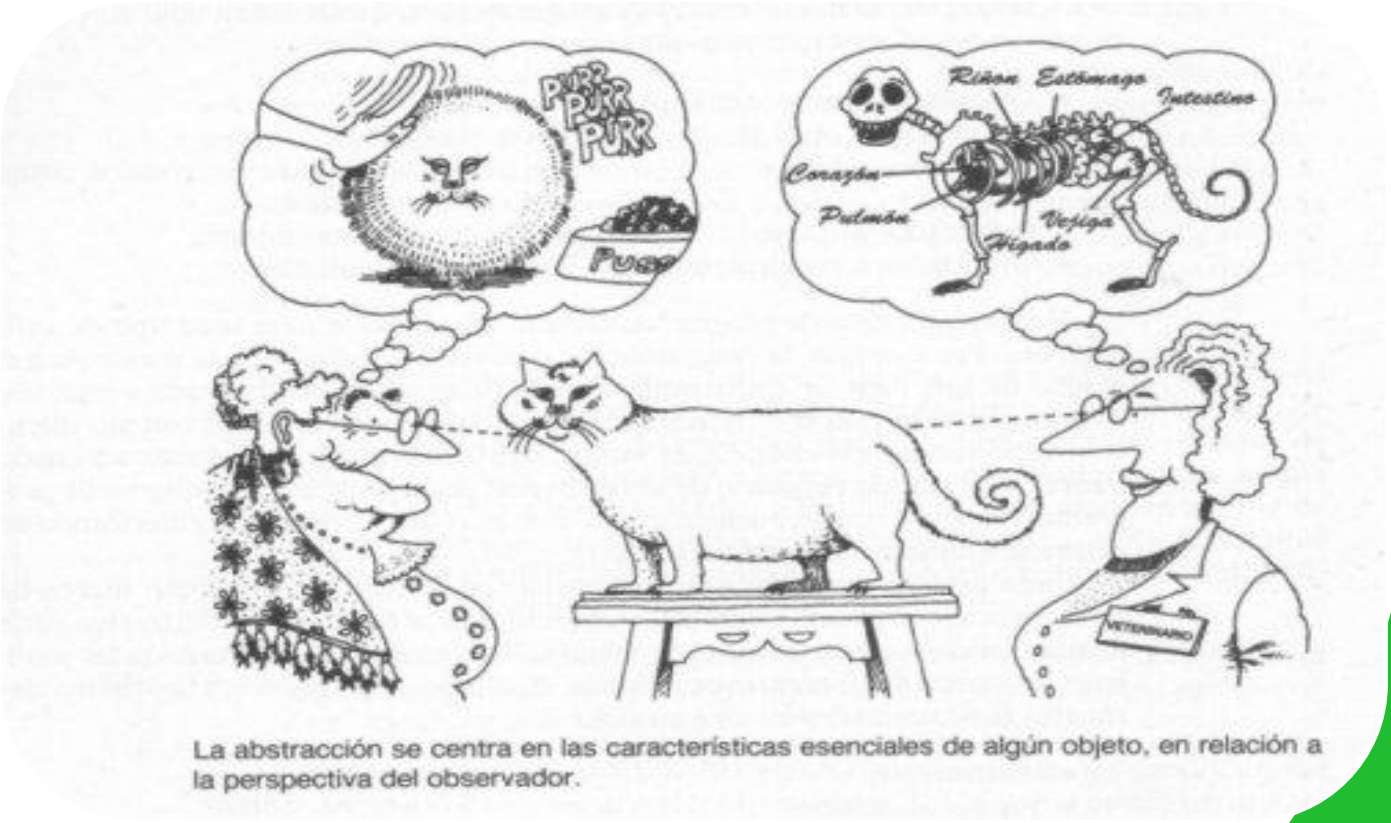
Encapsulación



Herencia

Abstracción

- La mayoría de las veces, cuando estás creando un programa con programación orientada a objetos, le das forma a los objetos del programa basados en objetos del mundo real. Sin embargo, los objetos del programa no representan los originales con 100% de precisión (y rara vez se requiere que lo hagan). En cambio, sus objetos solo modelan atributos y comportamientos de objetos reales en un contexto específico, ignorando el resto.

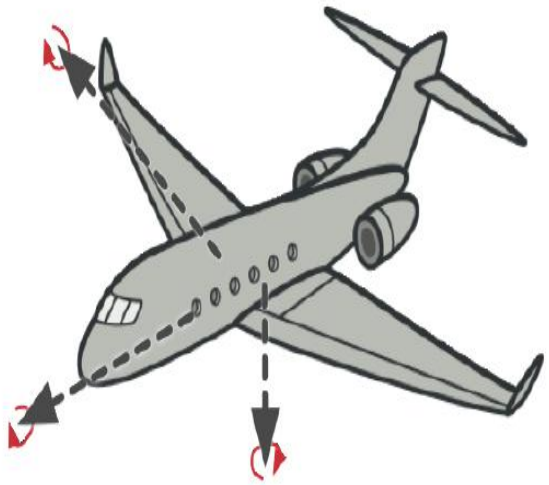


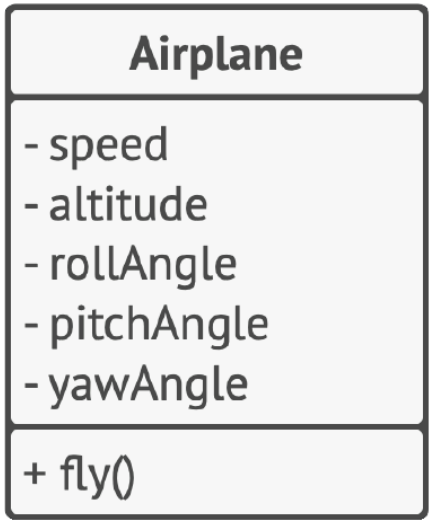
La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

Abstracción

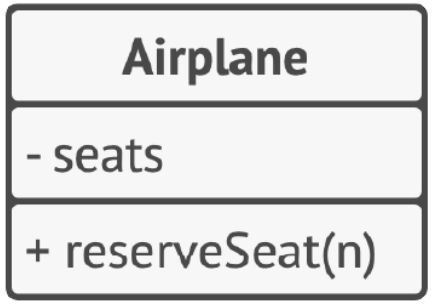
- Por ejemplo, una clase Avión probablemente podría existir para:

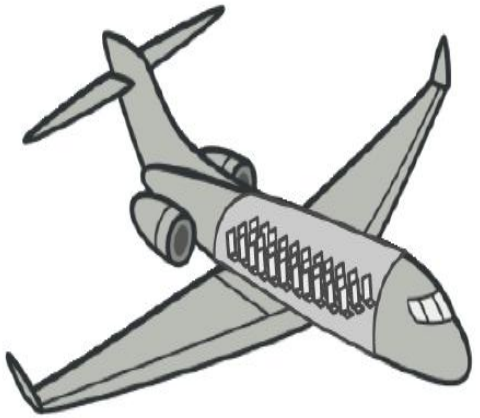
Un simulador de vuelo requiere de los detalles relacionados con el vuelo real



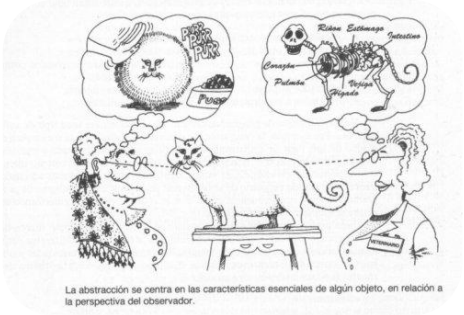


Una aplicación de reserva de vuelos que solo le importa el asiento en un mapa y qué asientos están disponibles.



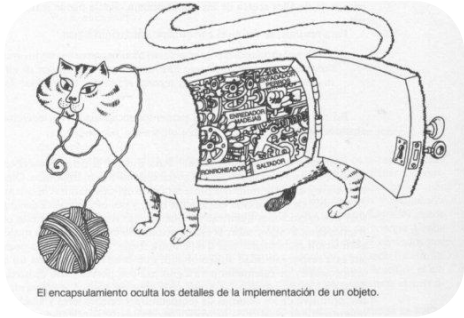


El paradigma orientado a objetos se basa en cuatro pilares

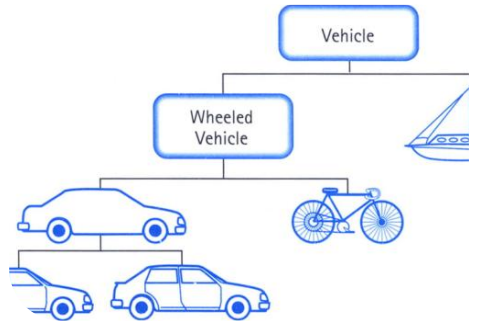


Abstracción

Polimorfismo



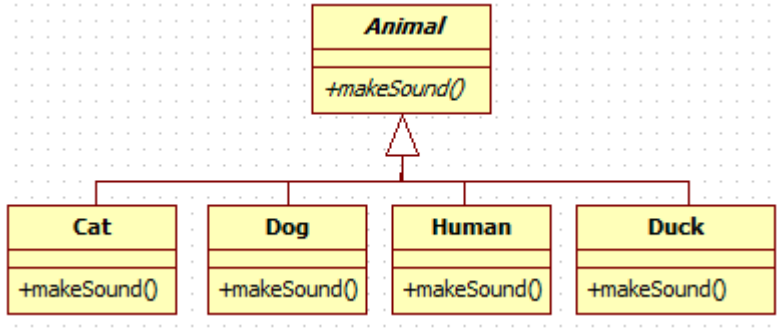
Encapsulación



Herencia

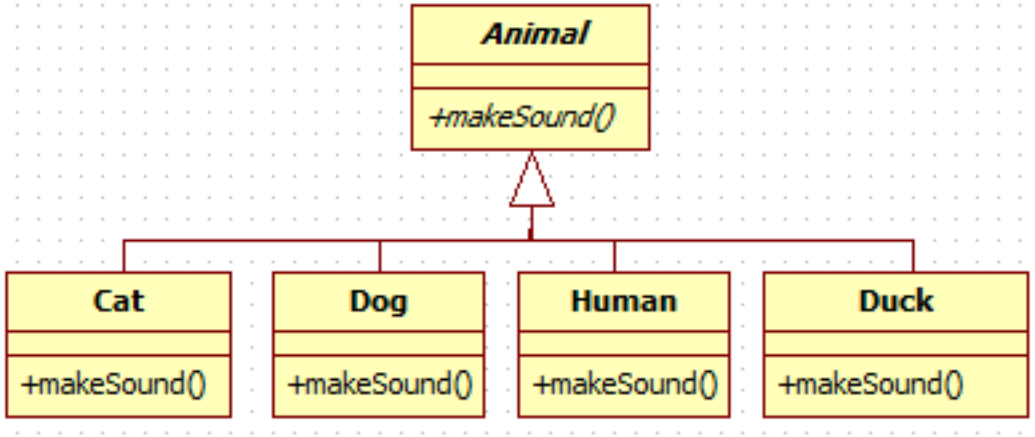
Polimorfismo

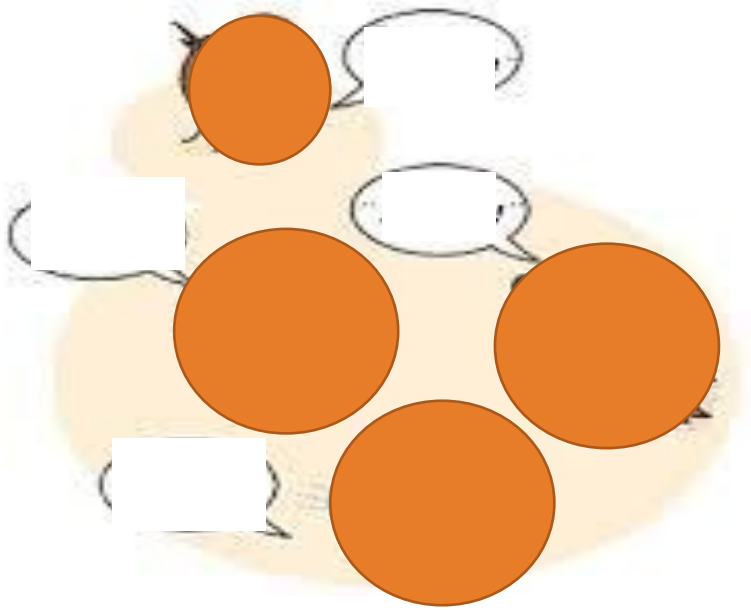
- La mayoría de los animales pueden hacer sonidos. Podemos anticipar que todas las subclases deberán definir lo que significa makeSound. Por lo tanto podemos declararlo abstracto. Esto nos permite omitir cualquier implementación predeterminada del método en la superclase, pero obliga a todas las subclases a aparecer con los suyos.



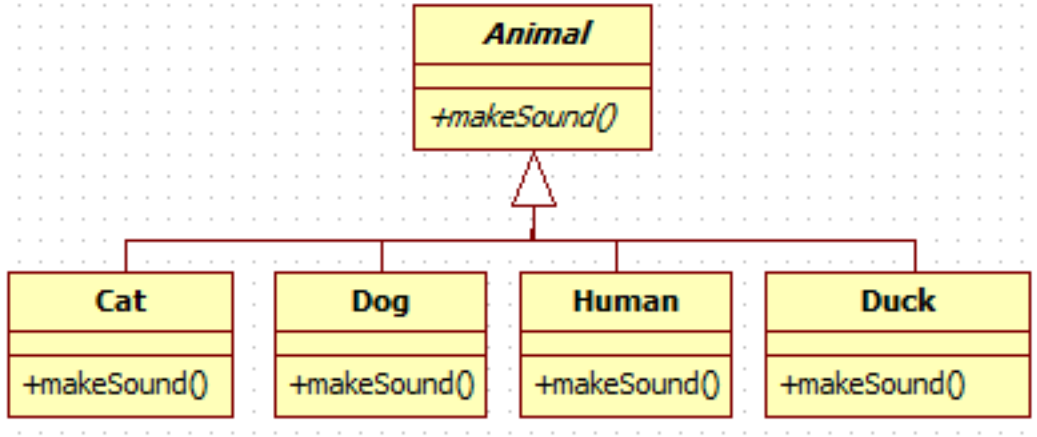
Polimorfismo

- Imagina que metemos muchos animales a una bolsa y no sabemos lo que son

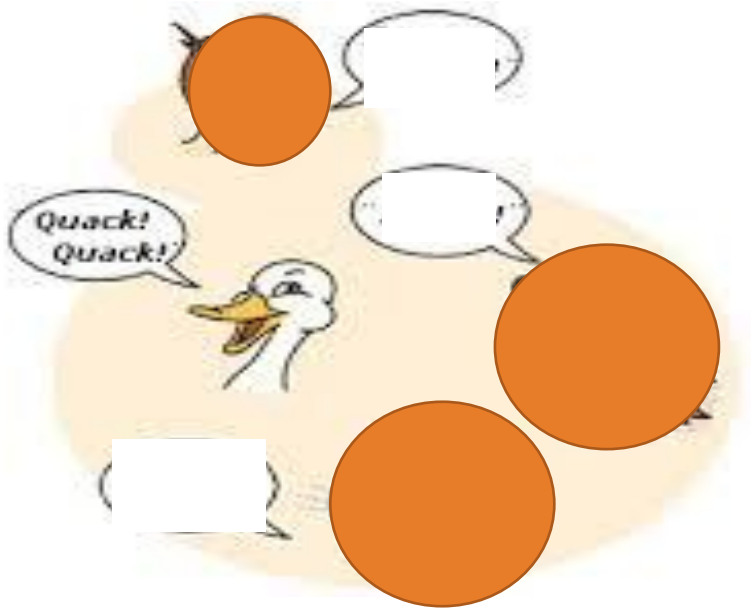




Polimorfismo

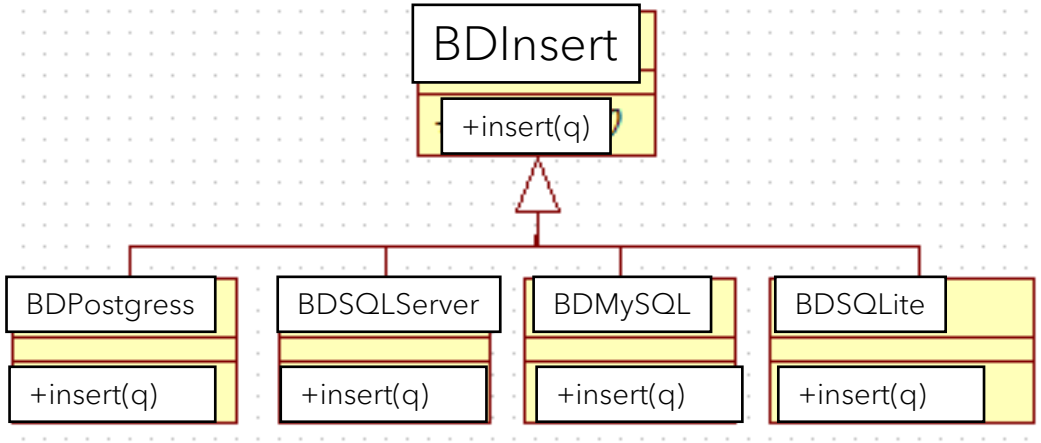


- Imagina que metemos muchos animales a una bolsa y no sabemos lo que son



En la vida real si los motivamos, un apretón, podríamos oírlos y determinar lo que son ☺

Polimorfismo



- Un programa no sabe que Animal **(a)** esta en la bolsa (bag)

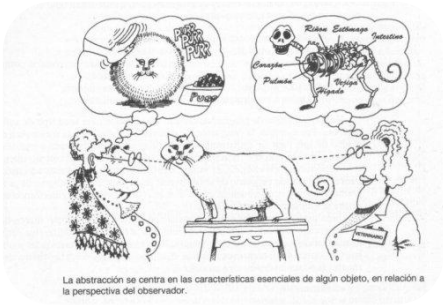
```
1 bag = [new Cat(), new Dog(), new Duck(), new Human()];
2
3 foreach (Animal a : bag)
4 a.makeSound()
5
6 // Hola!
7 // Quack quack!
```

gracias al **polimorfismo** qué método ejecutara la clase que va saliendo de la bolsa

Polimorfismo

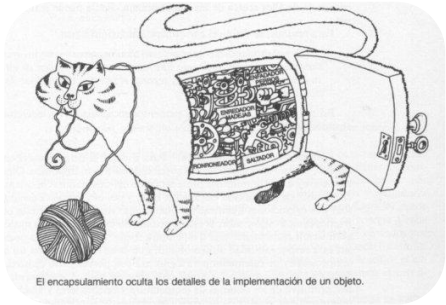
- La capacidad de un programa para detectar la verdadero clase de un objeto y llamar a su implementación incluso cuando su el tipo real es desconocido en el contexto actual.
- También puedes pensar en el polimorfismo como la capacidad de un objeto para "pretender" ser otra cosa, por lo general una clase que se extiende o una interfaz que implementa.
- En nuestro ejemplo, los perros y gatos en la bolsa se hacían pasar por animales genéricos.

El paradigma orientado a objetos se basa en cuatro pilares

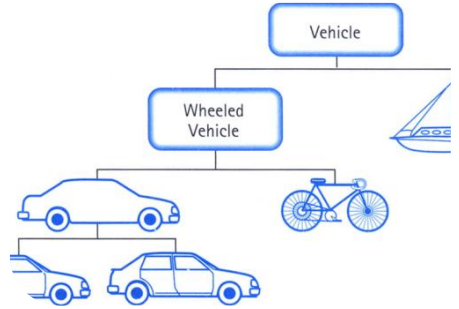


Abstracción

Polimorfismo



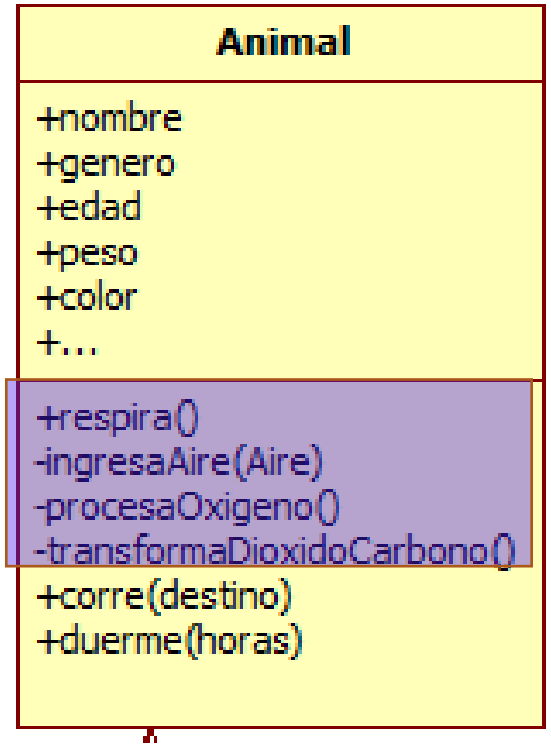
Encapsulación



Herencia

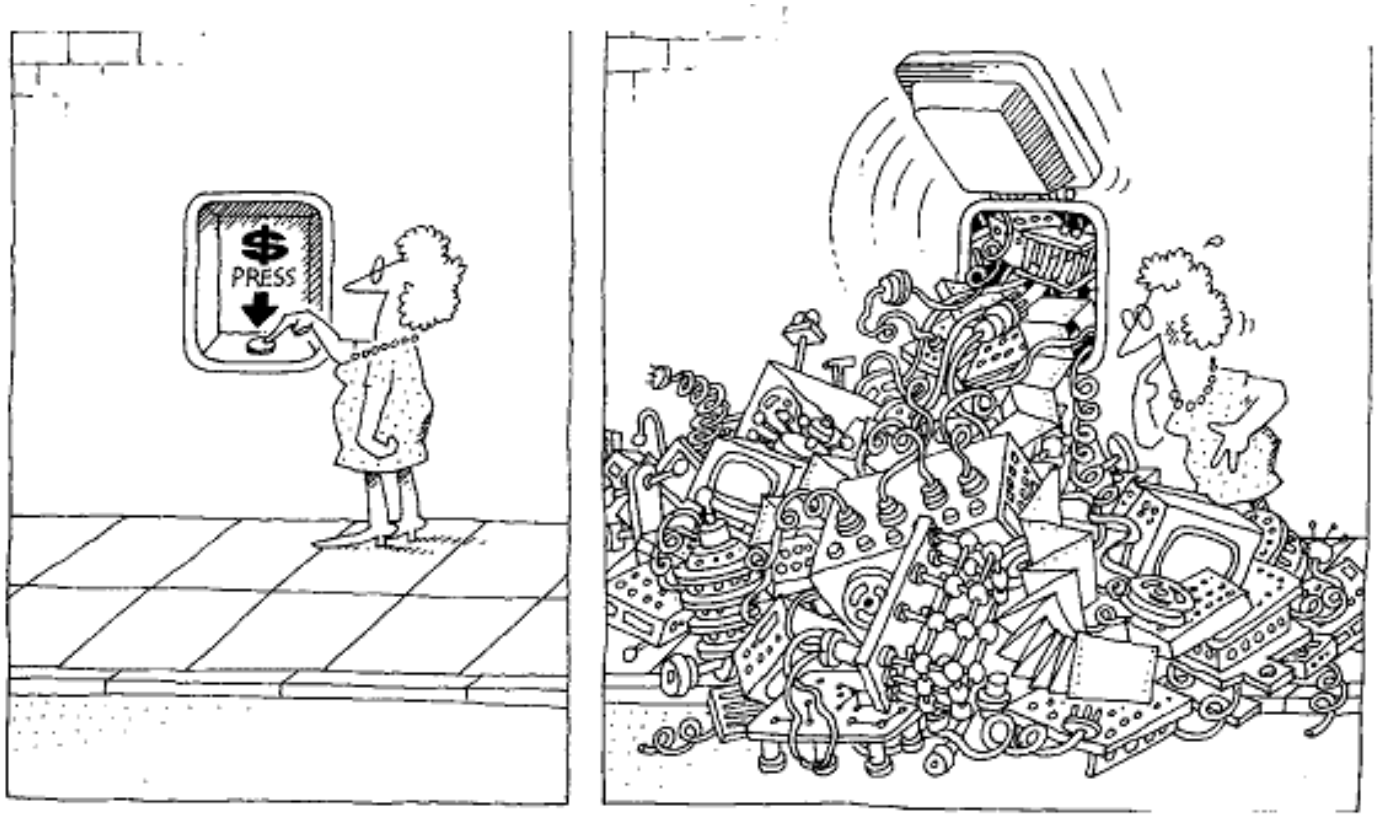
Encapsulación

- La encapsulación es la capacidad de un objeto para ocultar partes de su estado y comportamientos de otros objetos, exponiendo sólo una interfaz (funciones públicas o prtegidas) con el resto del programa.
- Encapsular algo significa hacerlo privado, y por lo tanto, sólo accesible desde dentro de los métodos de su propia clase. Otros objetos solo saben que el animal respira, no cómo lo hace, eso queda encapsulado en el objeto animal con métodos privados.



Los servicios a nuestro alcance esconden su complejidad.

Solo usamos Interfaces (funciones y métodos) públicos
+ retirar(cantidad)
+ depositar (cantidad)



Interfaces

- La parte pública de un objeto abierta a interacciones con otros objetos
- Interfaces y clases/métodos abstractos de la mayoría de la programación los lenguajes se basan en los conceptos de abstracción y encapsulación.
- En los lenguajes de programación orientados a objetos modernos, el mecanismo de interfaz (normalmente declarado con la palabra clave interface o protocol) le permite definir contratos (obligaciones a las subclases) de implementar la versión específica de las funciones declaradas en la interfaz.

  Esa es una de las razones por las que las interfaces sólo se preocupan por las funciones de los objetos y por qué no se puede declarar un atributo en una interfaz.

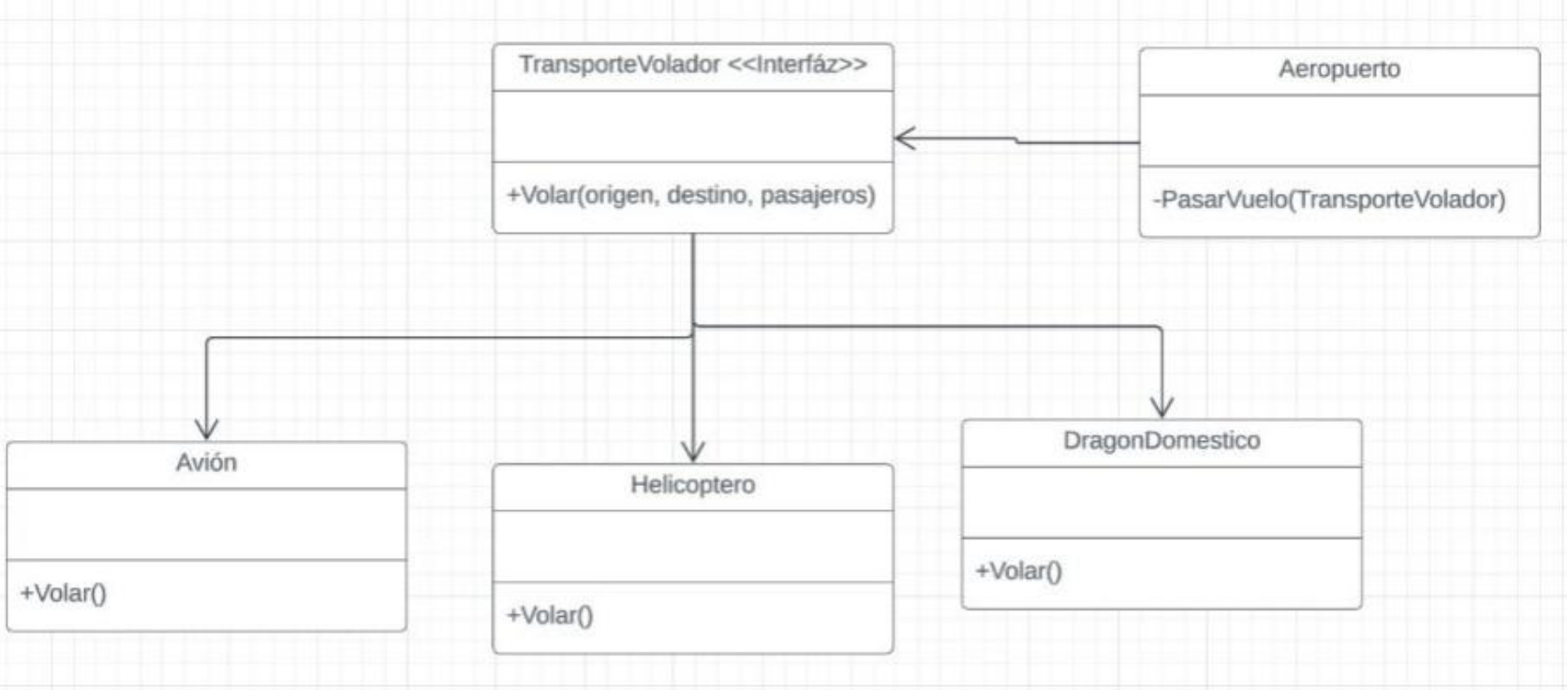
Orden de presentaciones (8 minutos)

- Equipo 6 Torres Sarmiento, Luis R., Vargas Tecuatl, Jose F., Vergara Mora, Jorge L., Villaverde Perez, Esmeralda V., Zarate Carreon, Jesus C.
- Equipo 1 Josue G., Amador Hernandez, Rodrigo, Aparicio Martinez, Francisco, Arias Santos, Angelica A., Blanco Peña, Eduardo A.
- Equipo 5 Prior Hernandez, Reychel, Rodriguez Maldonado, Jose A., Rojas Flores, Jose D., Salinas Gil, Diego, Santiago Ibañez, Jose L.
- Equipo 2 Briones Aguilar, Alfredo, Castillo Rodriguez, Emilio, Espinosa Alonso, Alejandro, Flores Flores, Jose E., Galicia Gonzalez, Jose D., Imhoff Rudolf Maximiliano
- Equipo 4 Leon Nieto, Amri, Lopez Morales, Javier, Martinez Lopez, Roberto, Mendez Brito, Carlos F., Ojeda Rodriguez, Osiris
- Equipo 3 Gomez Cruz, Marlen G., Guerra Bedolla, Eduardo, Guillen Sanchez, Maria D., Hernandez Aguilar, Irvin A., Ibarra Aguilar, Edwin

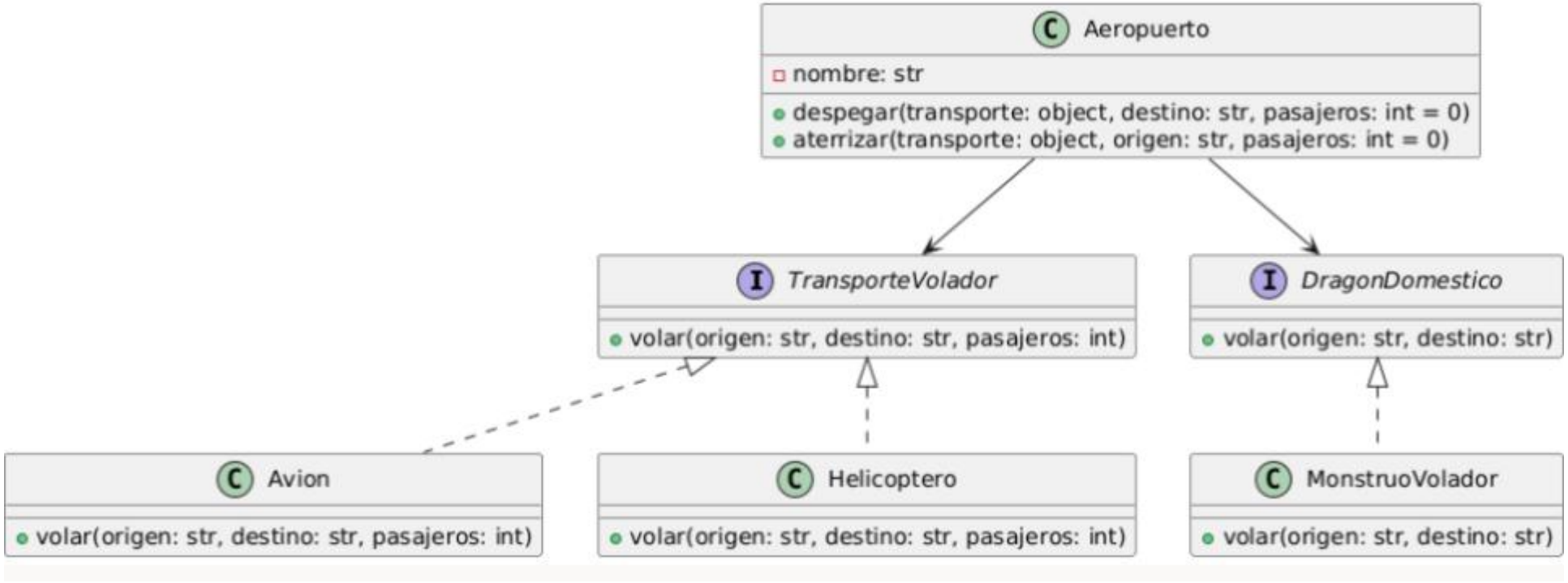
Escenario

- Imagina que tienes una interfaz TransporteVolador con un método de volar (origen, destino, pasajeros). Al diseñar un simulador de tráfico aéreo, podrías restringir la clase Aeropuerto que solo va a funcionar con los transportes aéreos que implementan la interfaz TransporteVolador y una variante del método vuelo. Al hacer esto, puedes estar seguro de que cualquier objeto que le pases al aeropuerto, ya sea un avión, un helicóptero o un monstruo volador DragonDomestico, podría llegar o salir de este tipo de aeropuerto solo si es de este tipo.
- ¿Cómo modelas esto?

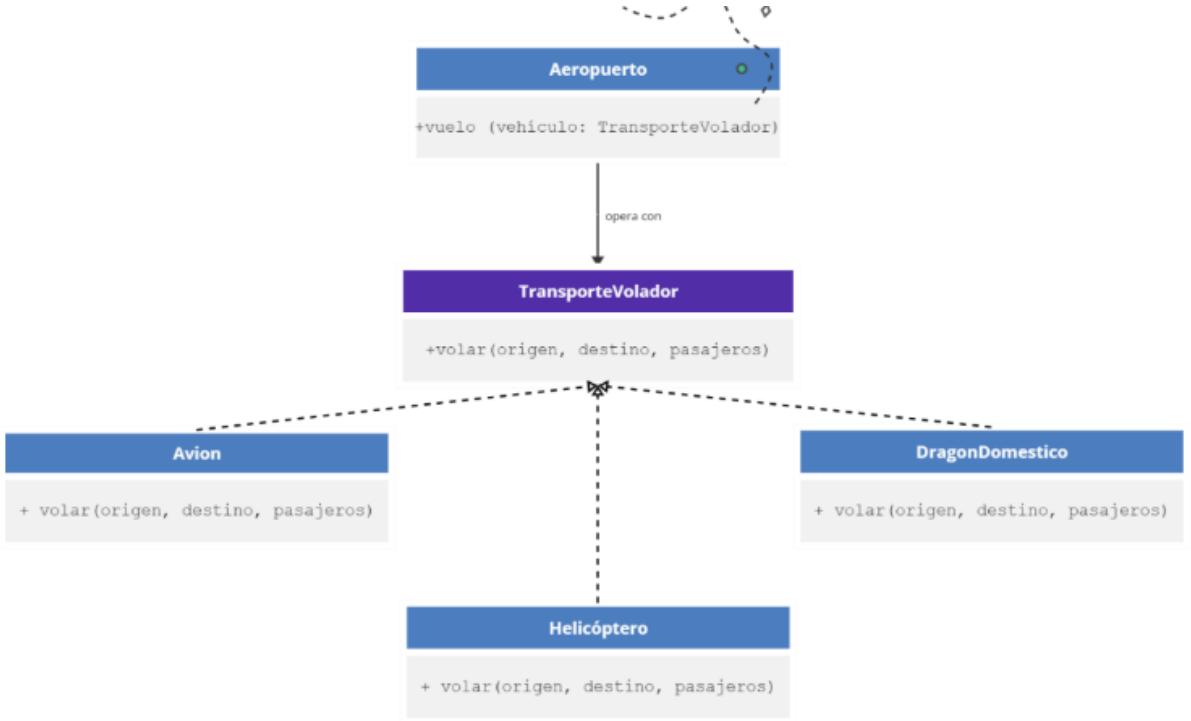
Rodrigo Amador



Francisco Aparicio

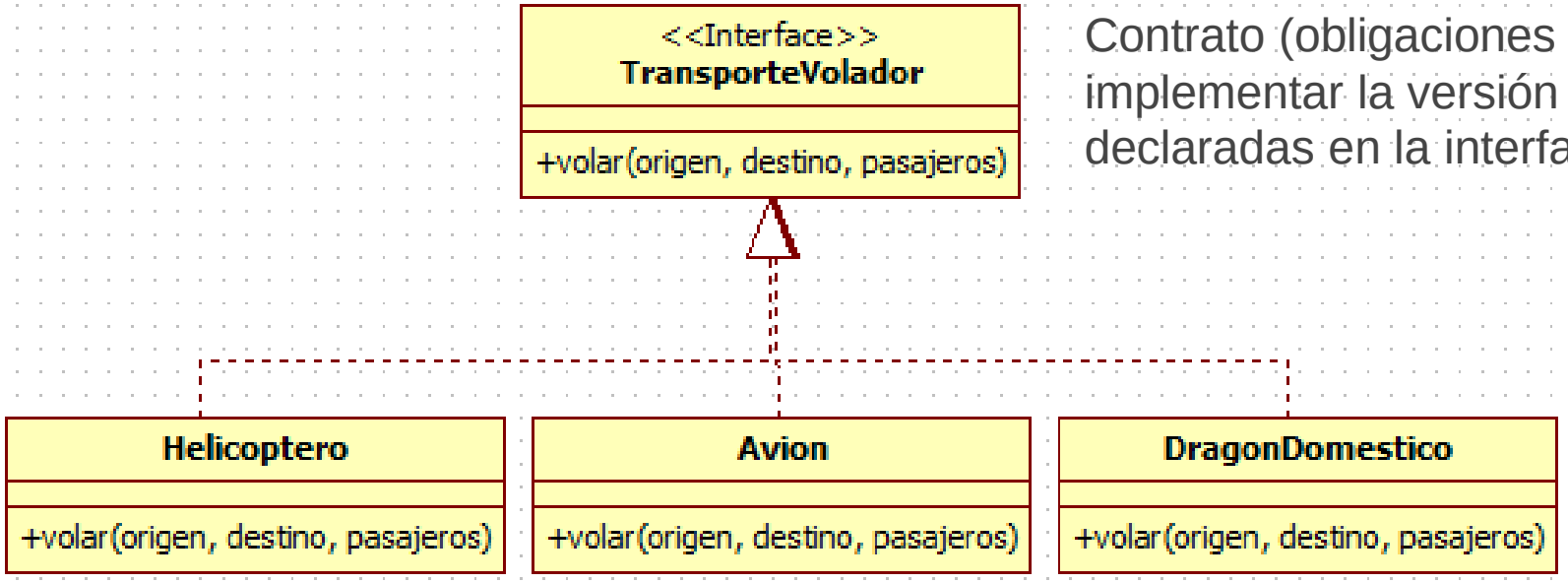


Angelica Arias



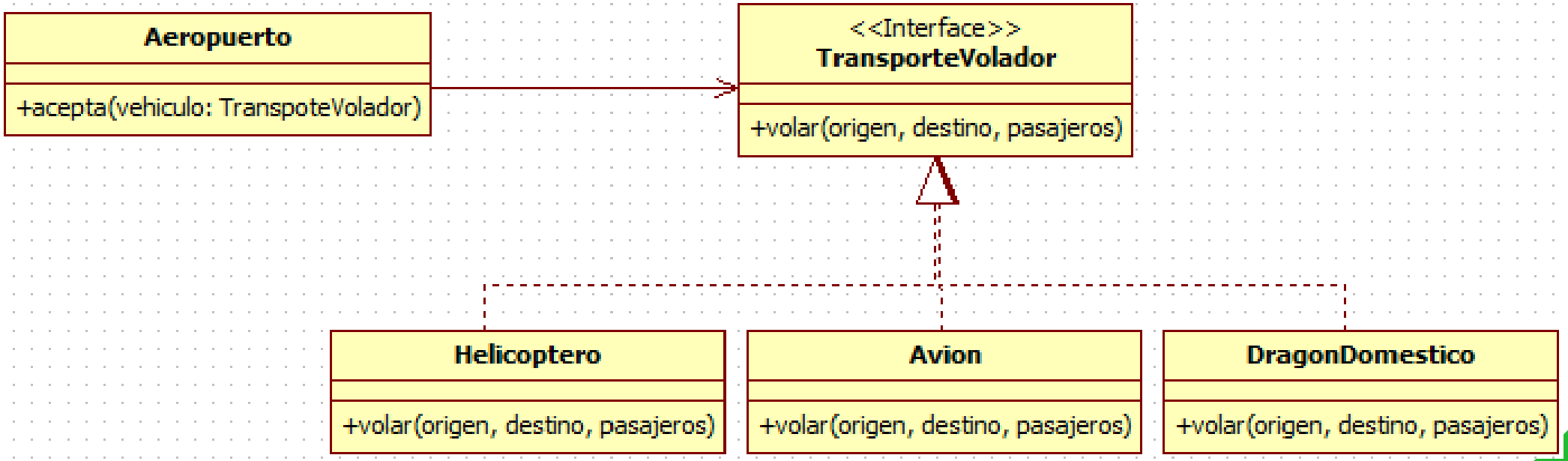
La Interfaz primero

La Interfaz primero



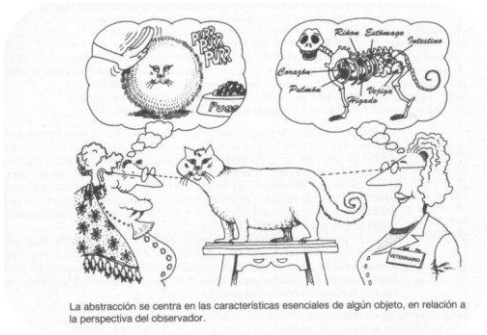
Contrato (obligaciones a las subclases) de implementar la versión específica de las funciones declaradas en la interfaz (volar en este caso).

Restringir que el Aeropuerto solo permite TransporteVolador



La implementación de volar puede cambiar las veces que quieras

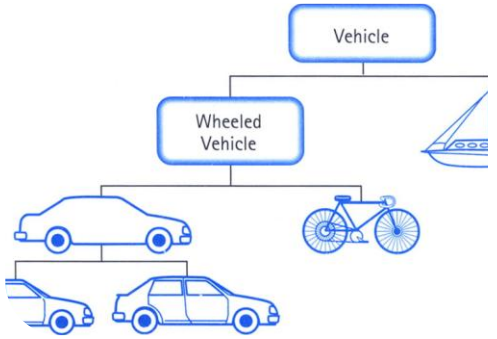
El paradigma orientado a objetos se basa en cuatro pilares



Abstracción

Polimorfismo

Encapsulación



Herencia

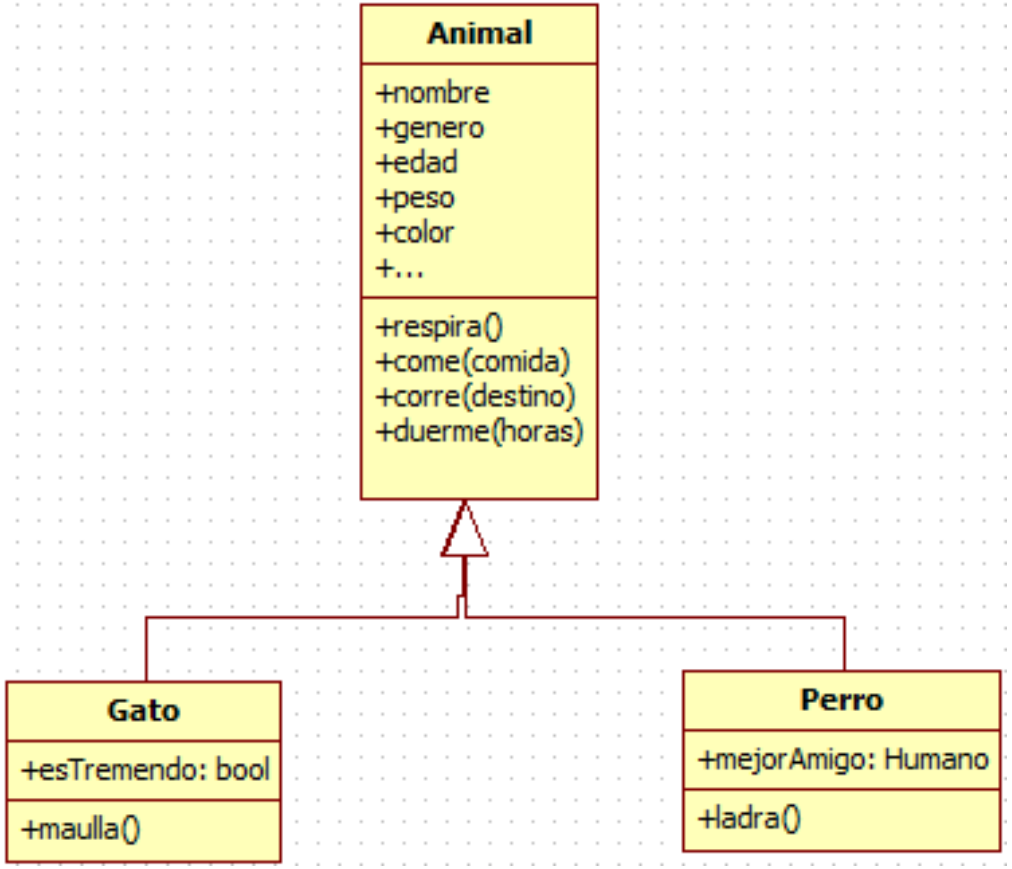
Herencia

- La herencia es la capacidad de construir nuevas clases sobre las existentes. unos.
- El principal beneficio de la herencia es la reutilización del código.

  Si deseas crear una clase que sea ligeramente diferente de una existente otra, no hay necesidad de duplicar el código.

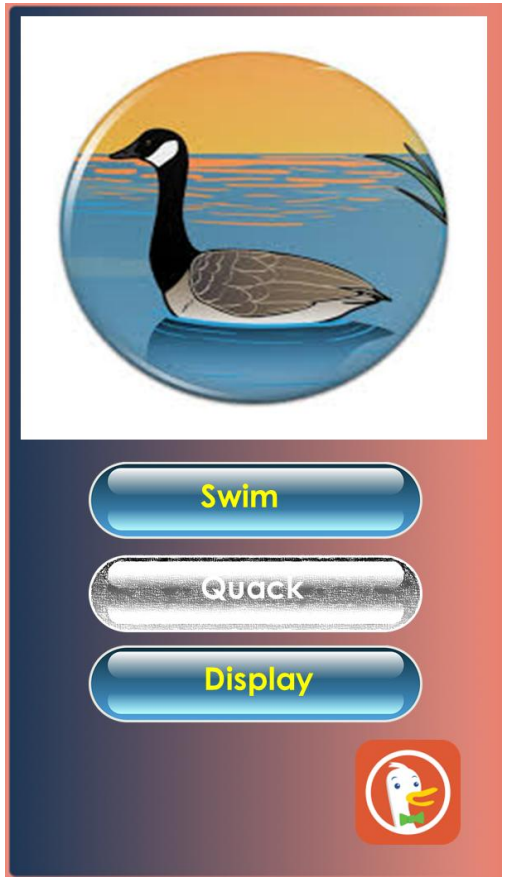
  Se extiende la clase existente y poner la funcionalidad adicional en un resultante

  subclase, que hereda campos y métodos de la superclase.



Patrones de Diseño de Software

- Simulador de Patos
  - Una app para mostrar los tipos de patos
    - Nadadores (swim())
    - Quackeadores (quack())
    - Presentación (display ())

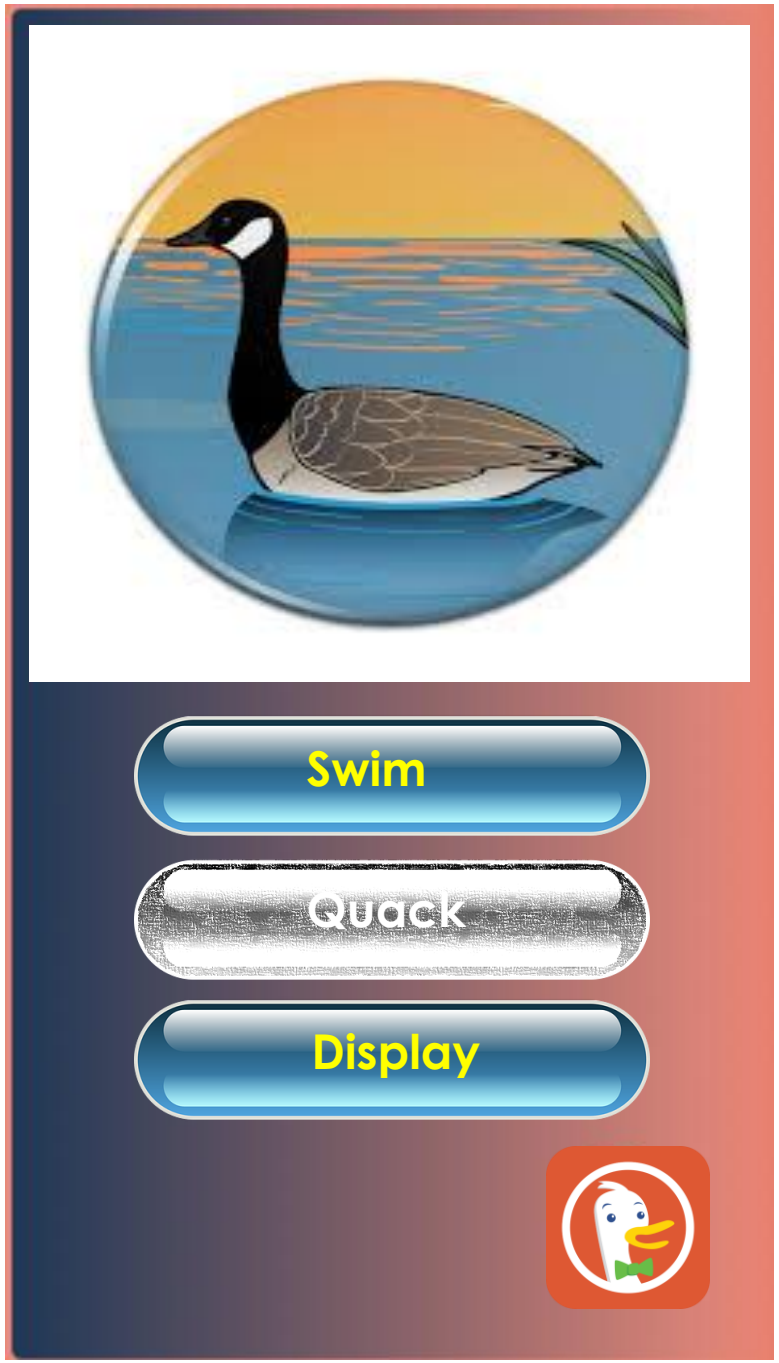


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

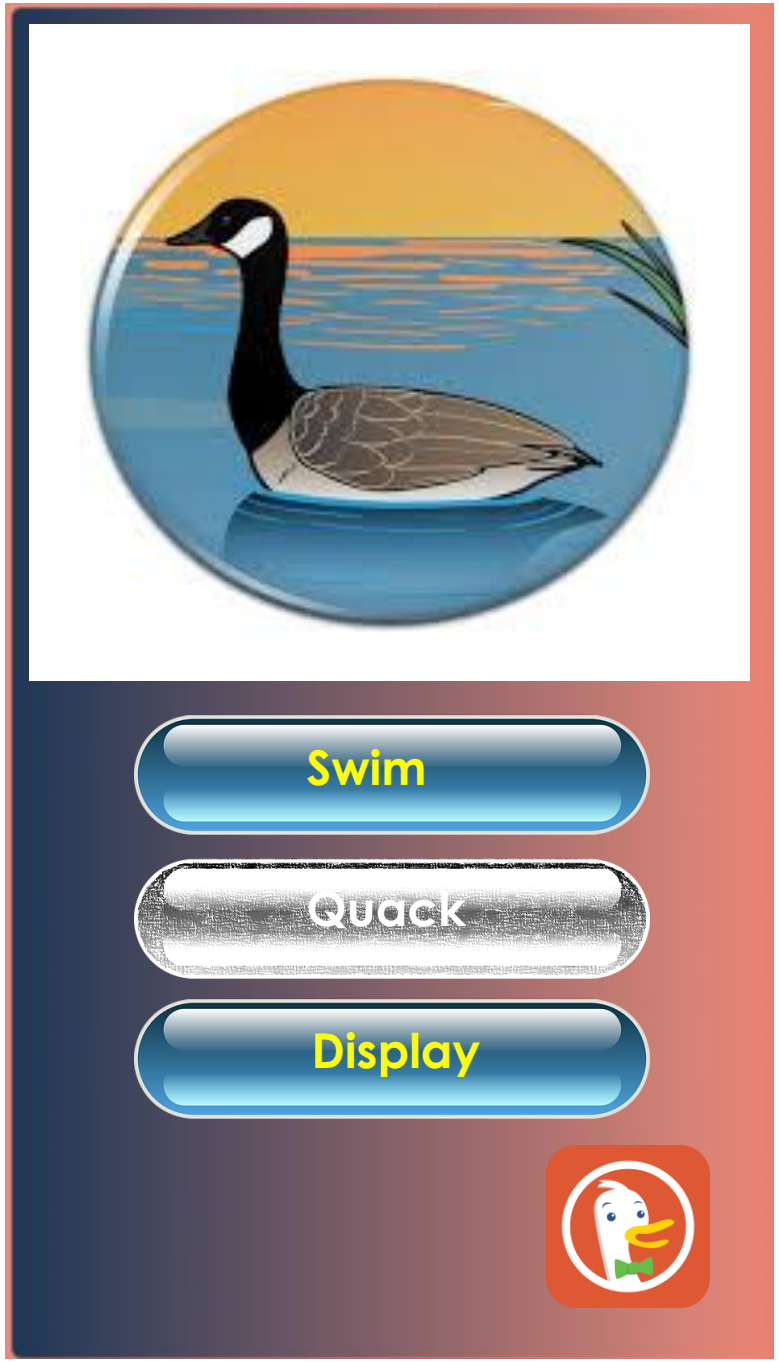
Patrones de Diseño de Software

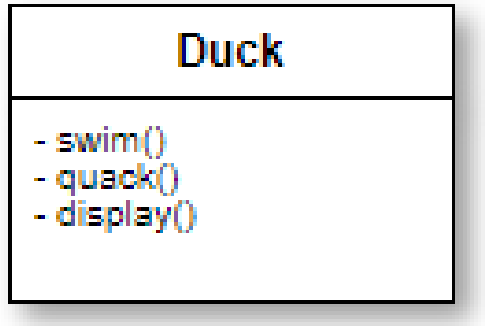
- Simulador de Patos
  - Una app para mostrar los tipos de patos
    - Nadadores (swim())
    - Quackeadores (quack())
    - Presentación (display ())
  - ¿Cómo queda su clase Duck?
    - Dejen de lado los atributos por el momento.

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly



Duck
- swim() - quack() - display()



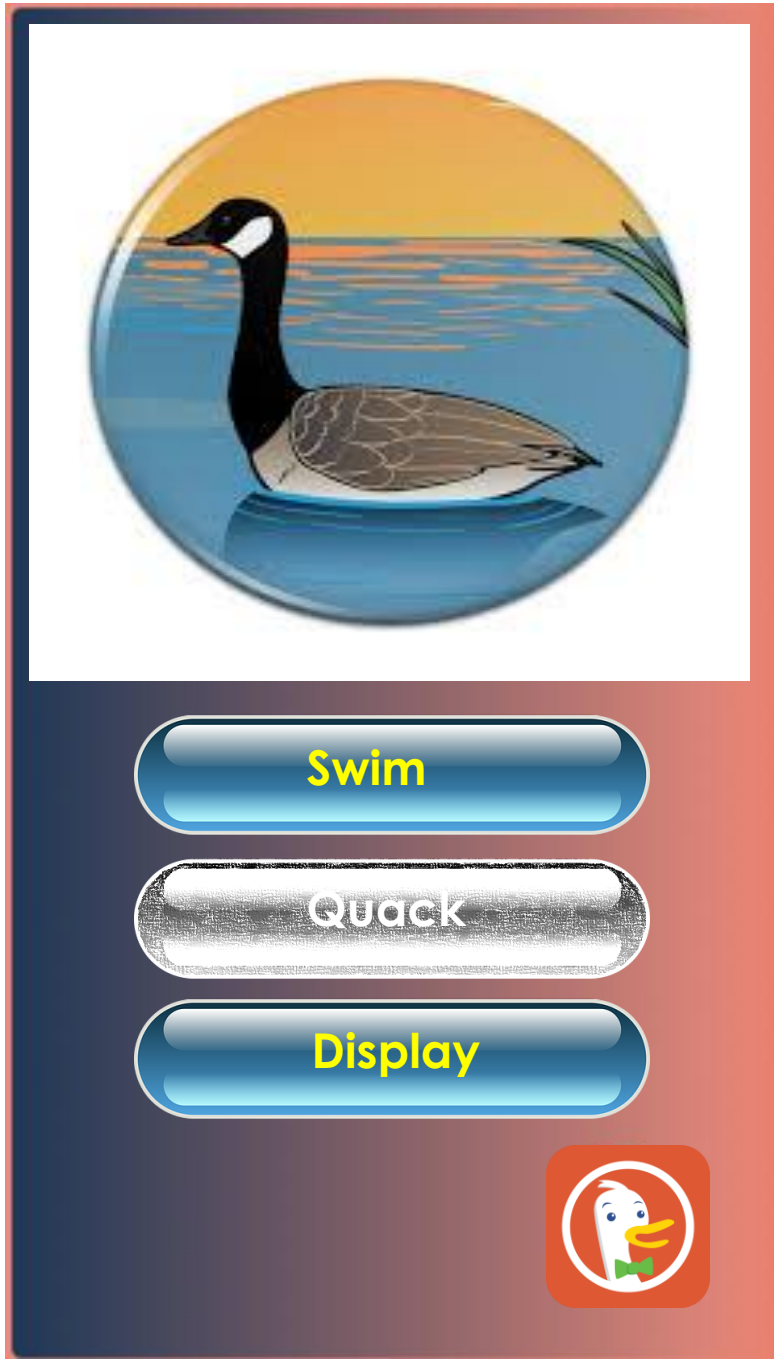


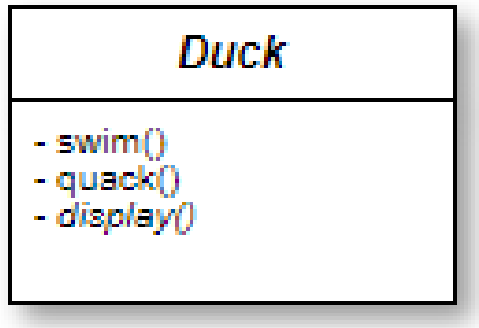
¿Qué métodos son abstractos y se deben sobrescribir en las subclases?

- ¿Qué cambia?

¿Qué código puedo escribir una vez y sirve para todos?

- ¿Qué es igual para todos los patos?



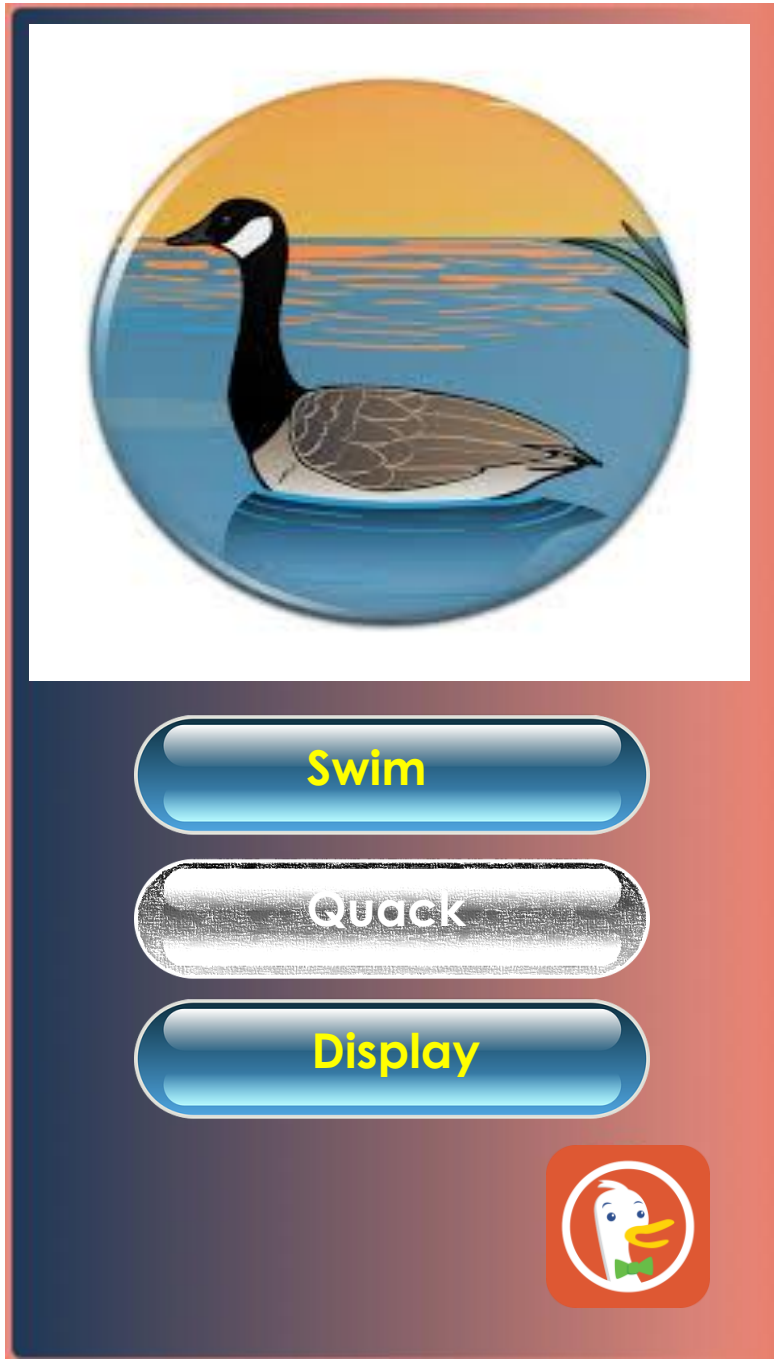


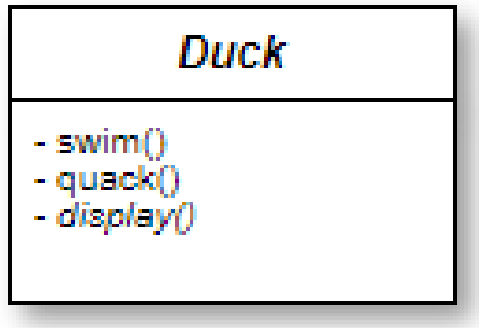
¿Qué métodos son abstractos y se deben sobrescribir en las subclases?

- ¿Qué cambia?
  - ***display()***

¿Qué código puedo escribir una vez y sirve para todos?

- ¿Qué es igual para todos los patos?
  - **swim()**
  - **quack()**





¿Qué métodos son abstractos y se deben sobrescribir en las subclases?

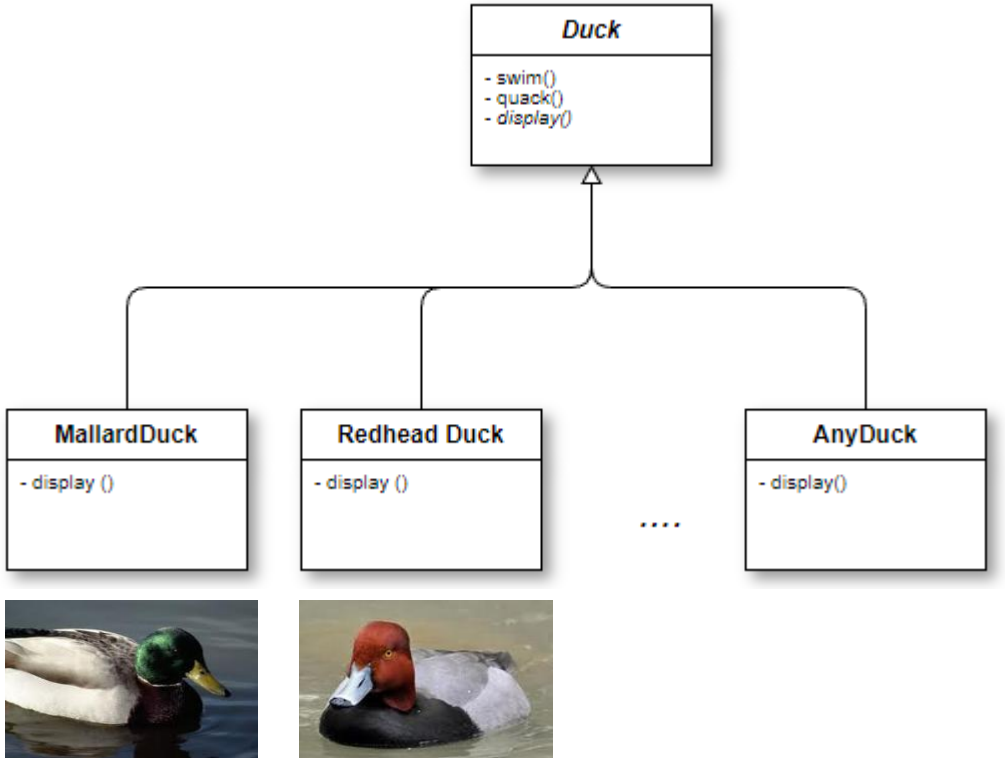
- ¿Qué cambia?
  - ***display()***

¿Qué código puedo escribir una vez y sirve para todos?

- ¿Qué es igual para todos los patos?
  - **swim()**
  - **quack()**

Simulador de Patos

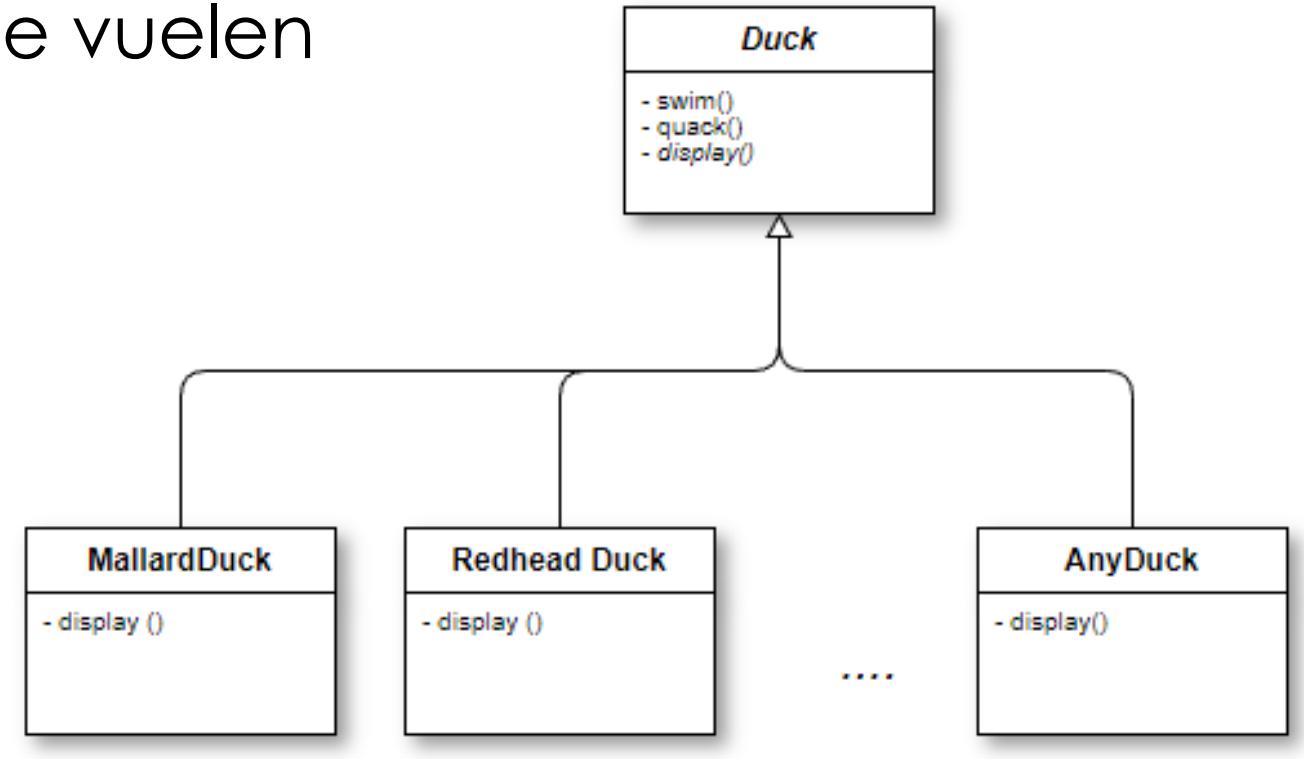
- El método **display()** se define en cada subclase pues es diferente para cada pato



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

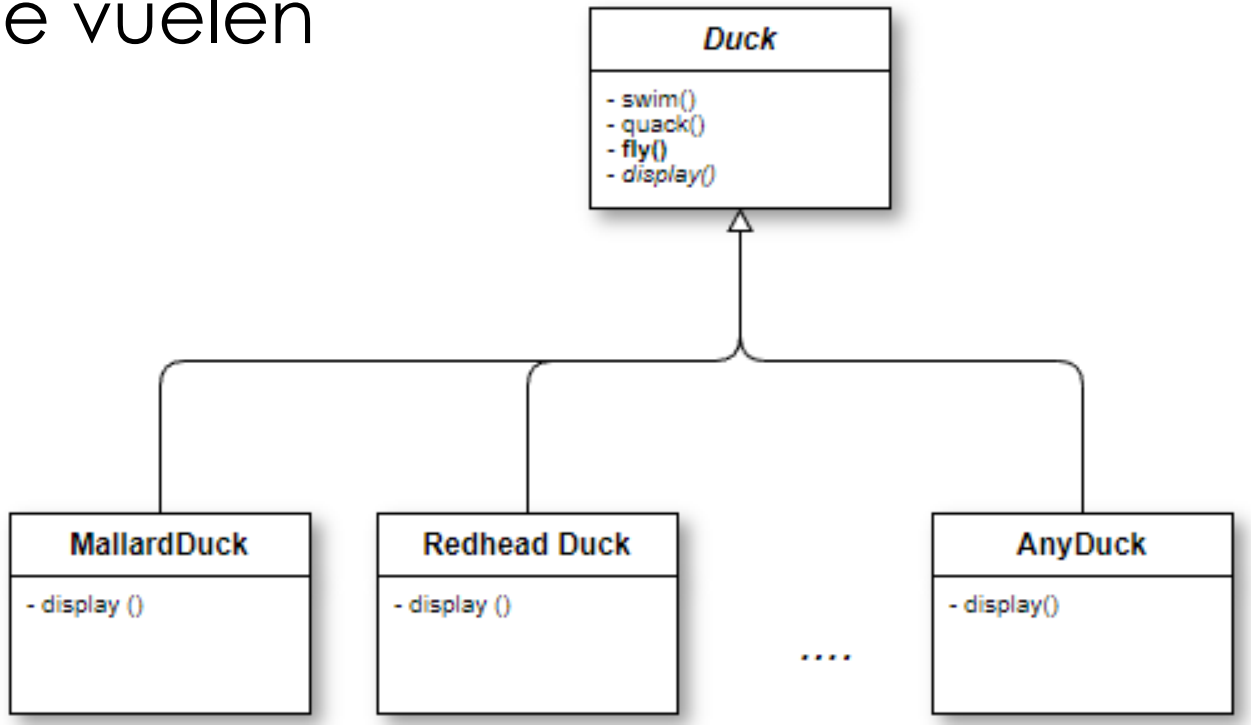
- Los patos queremos que vuelen
- ¿Qué cambia?



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

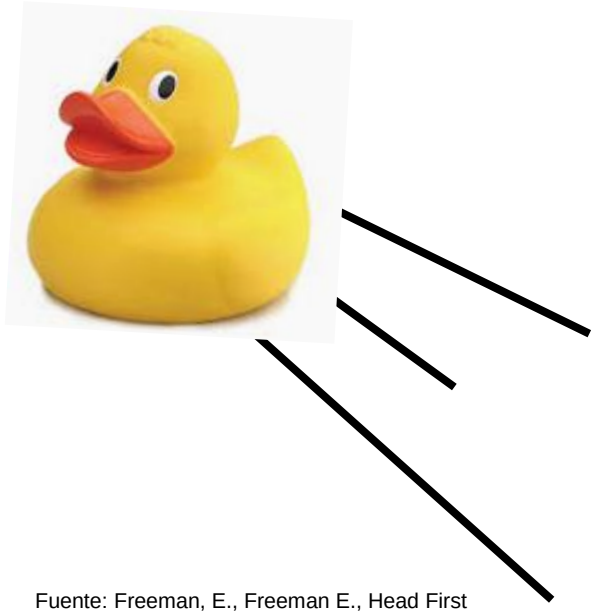
- Los patos queremos que vuelen
- ¿Qué cambia?
  - **Método fly()**

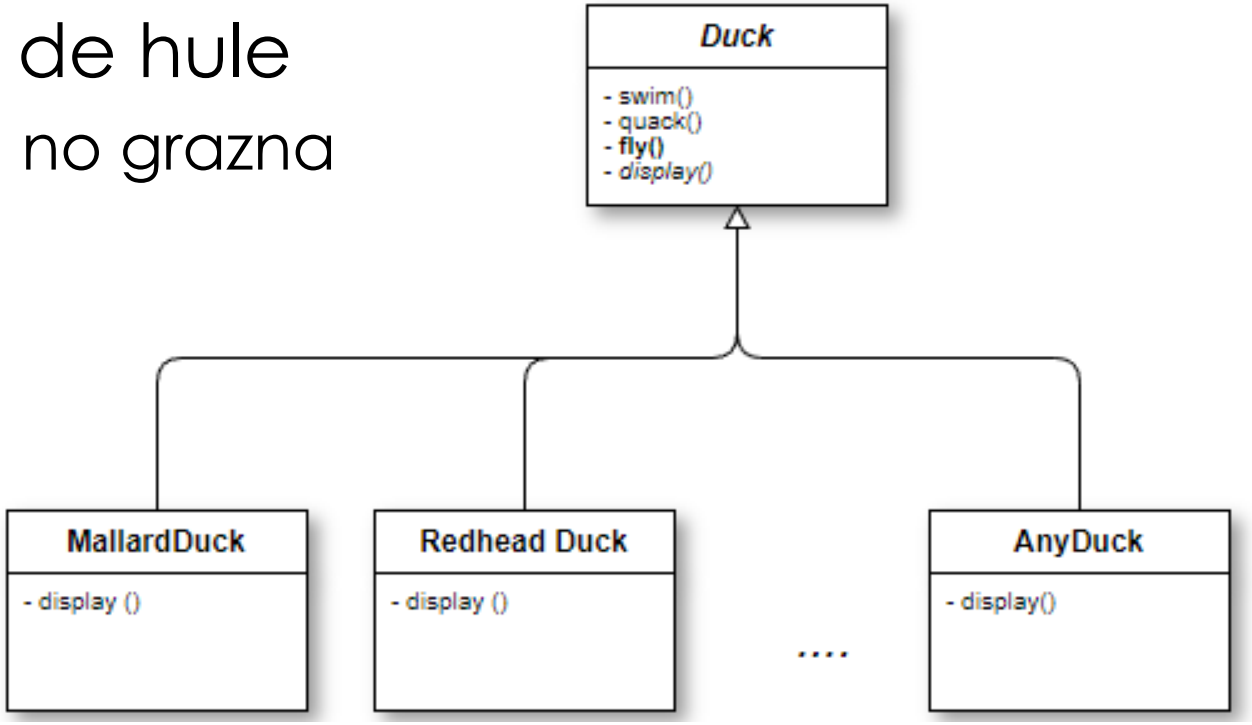


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

- Nos piden agregar un patito de hule
  - No vuela, ah!, incluso no chilla no grazna
- ¿Qué cambia?

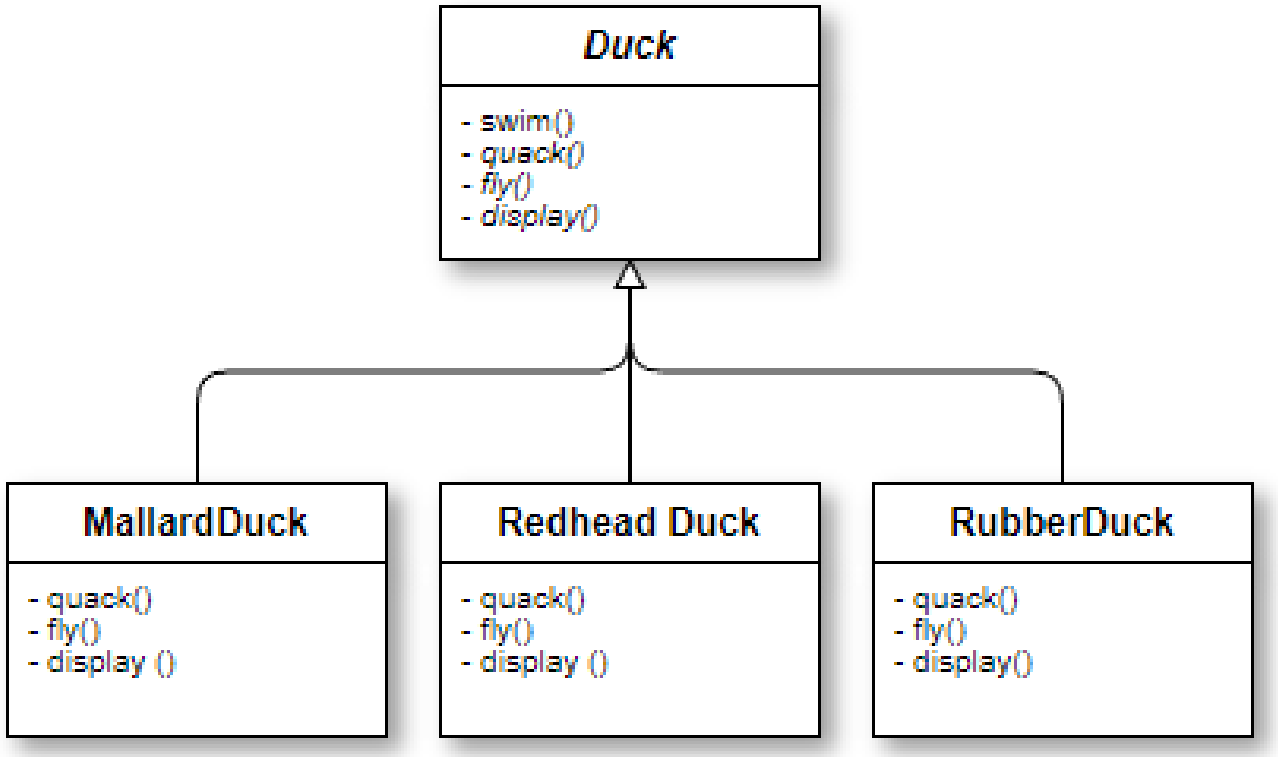


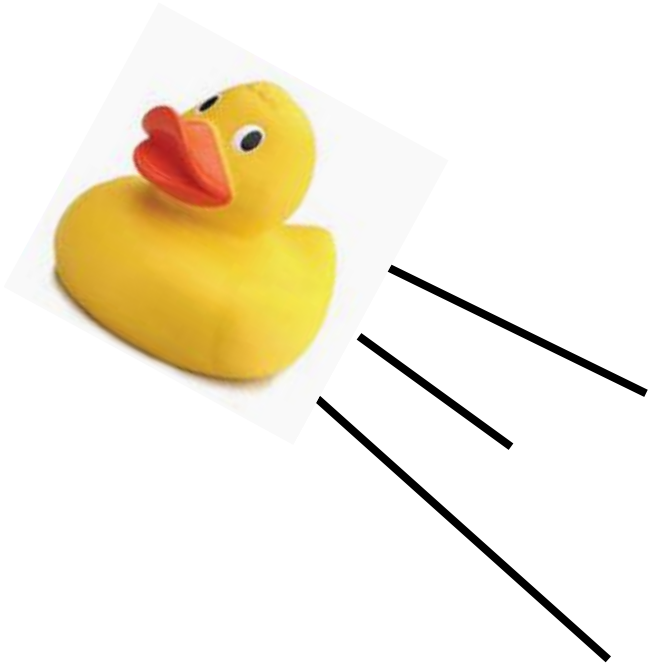


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

- ¿Qué dicen si sobrescribimos?

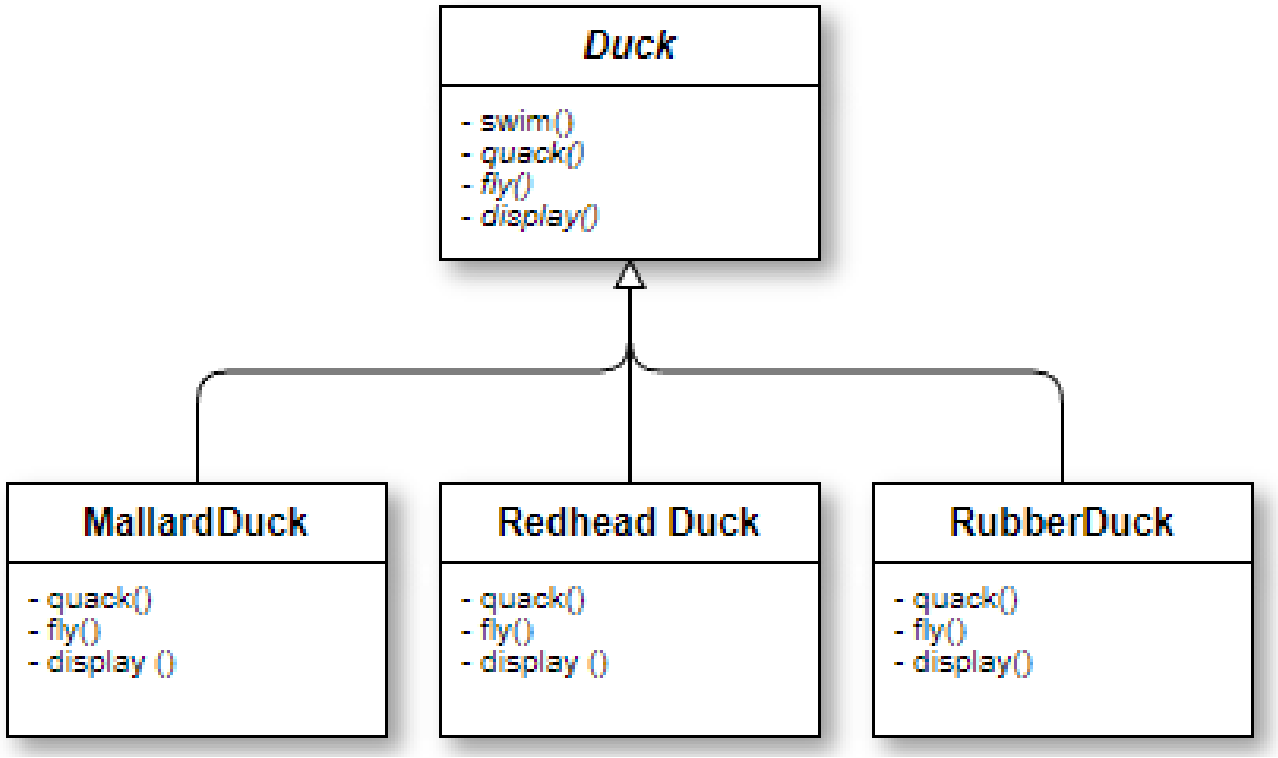




Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

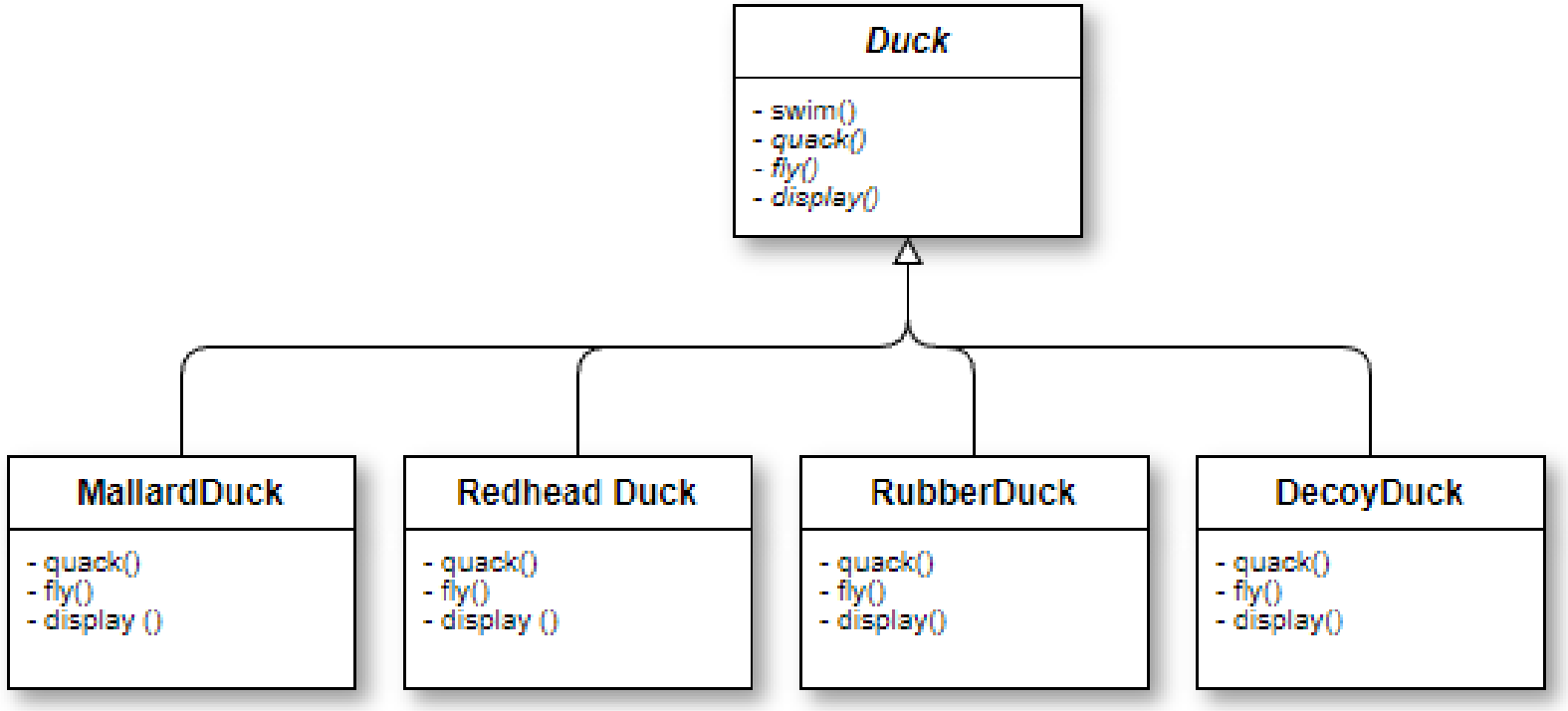
- Y si queremos un patito de madera ¿Hacemos lo mismo?



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

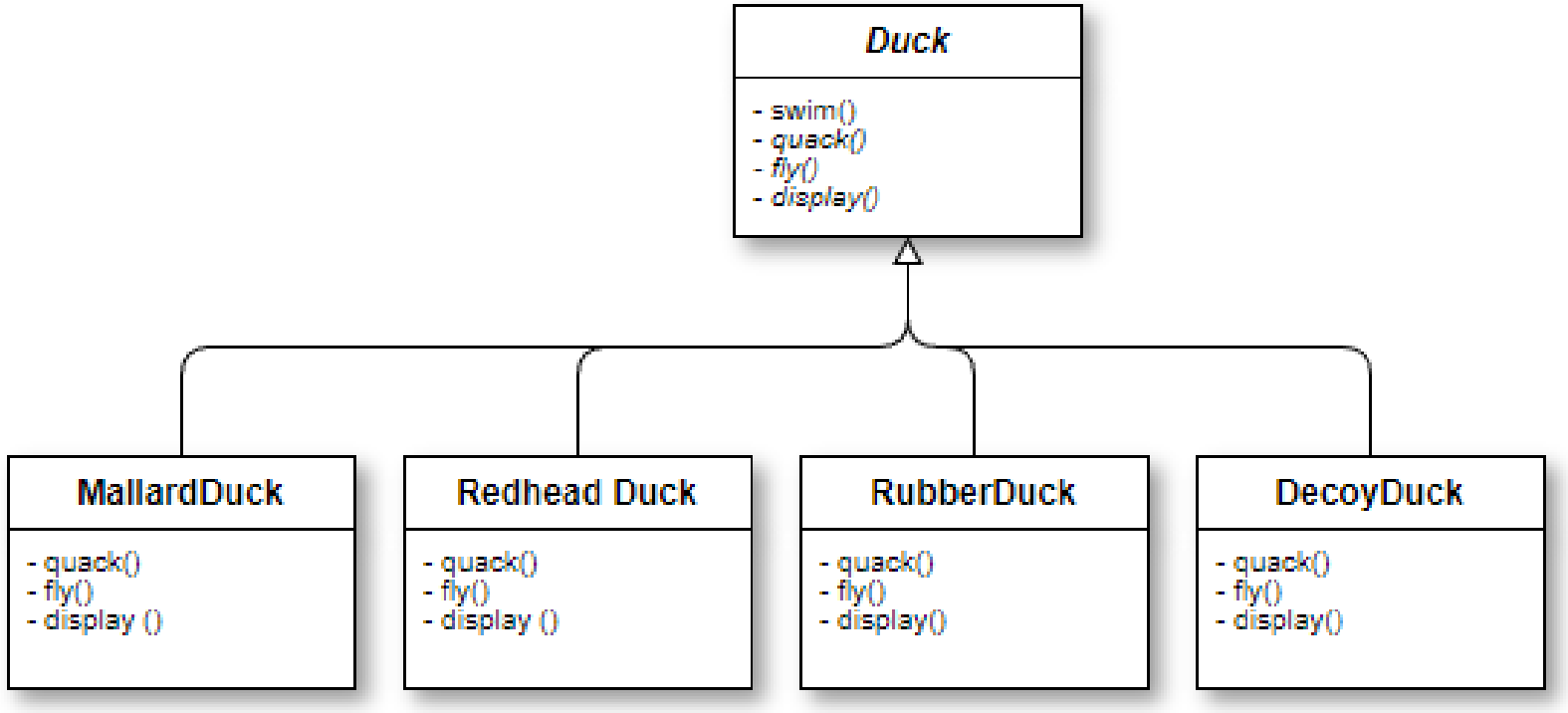
- Y si queremos un patito de madera ¿Hacemos lo mismo?



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

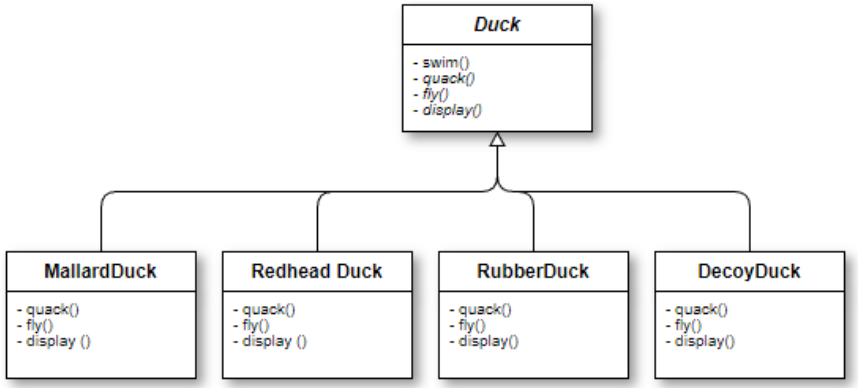
- ¿Cuáles son las desventajas de usar subclases para agregar comportamiento especifico a cada pato?



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

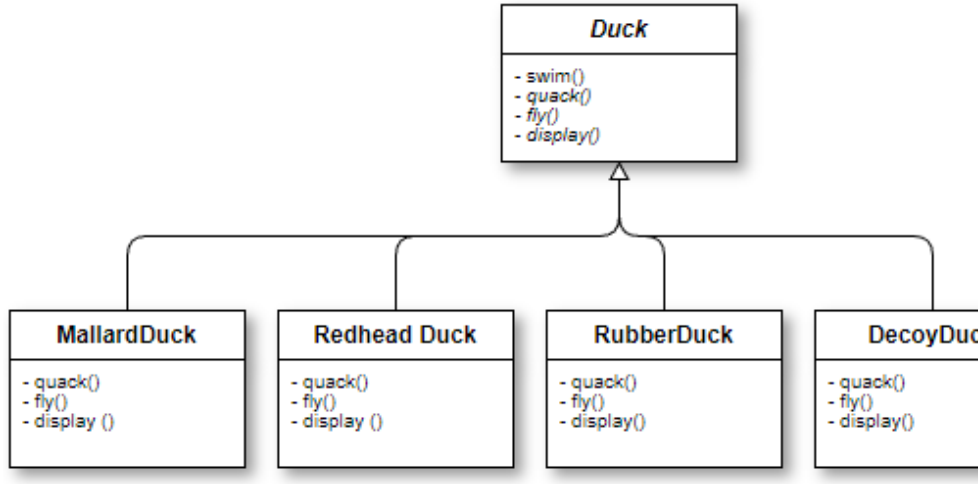
- ¿Cuáles son las desventajas de usar subclases para agregar comportamiento especifico a cada pato?
  - **El código es duplicado en las subclases**



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

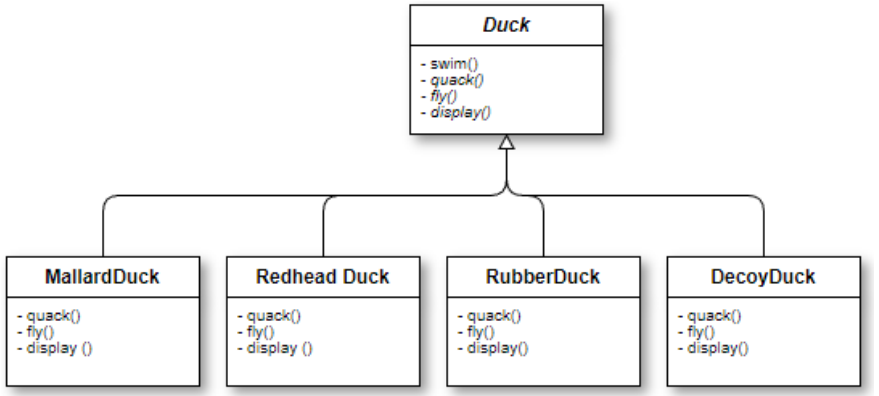
- ¿Cuáles son las desventajas de usar subclases para agregar comportamiento especifico a cada pato?
  - El código es duplicado en las subclases
  - **Cambios de funcionalidad son difíciles**



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

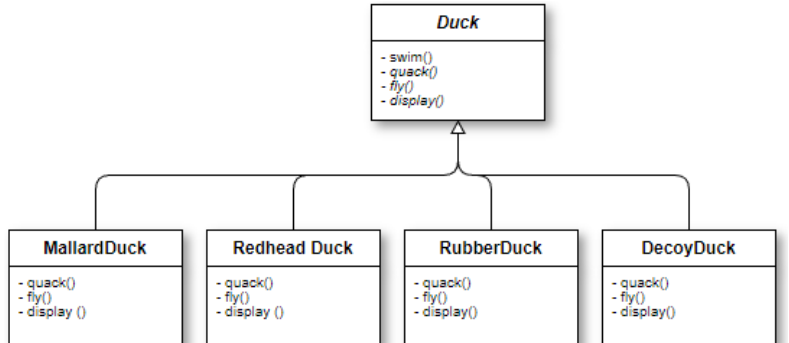
- ¿Cuáles son las desventajas de usar subclases para agregar comportamiento especifico a cada pato?
  - El código es duplicado en las subclases
  - Cambios de funcionalidad son difíciles
  - **Difícil obtener conocimiento del comportamiento de los patos, cada quien puede hacer lo que quiere**



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

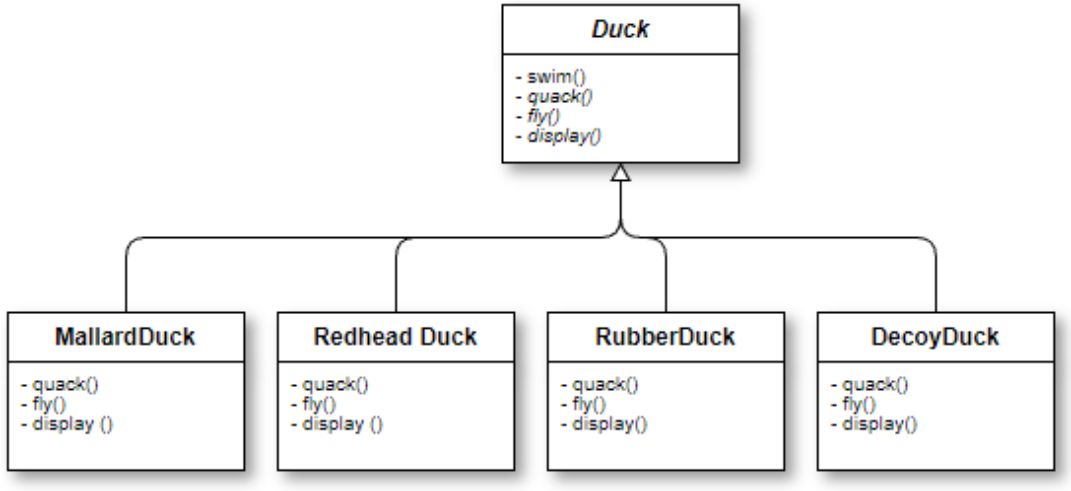
Simulador de Patos

- ¿Cuáles son las desventajas de usar subclases para agregar comportamiento especifico 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**

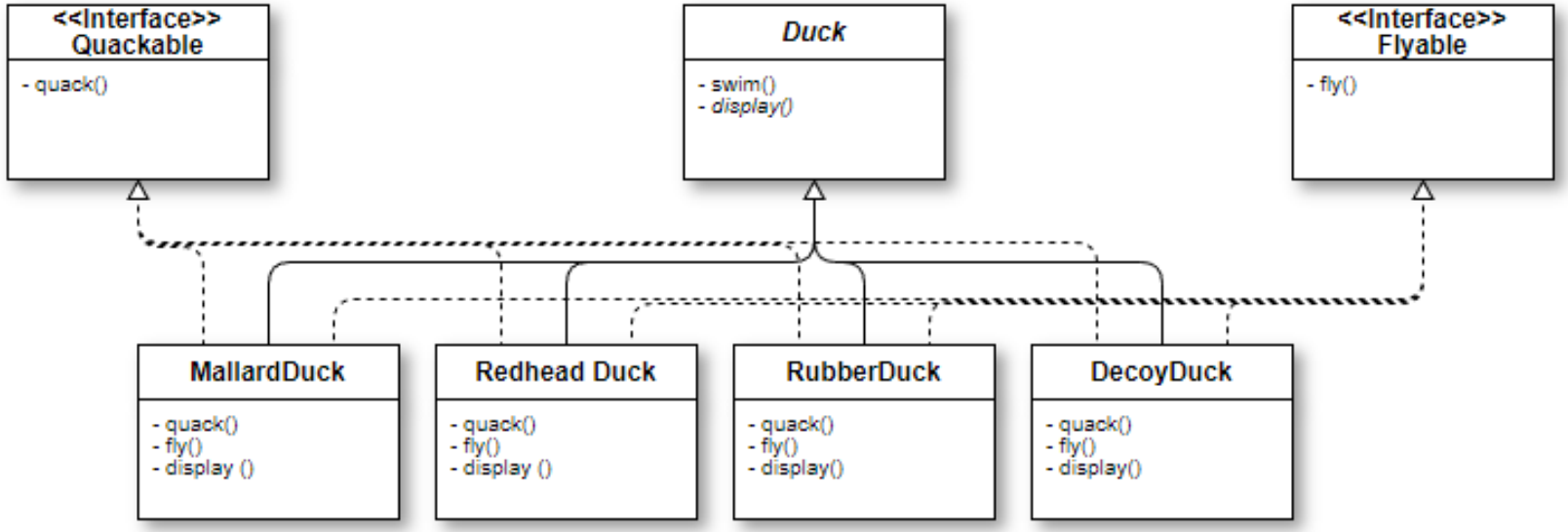


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos



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



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

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**

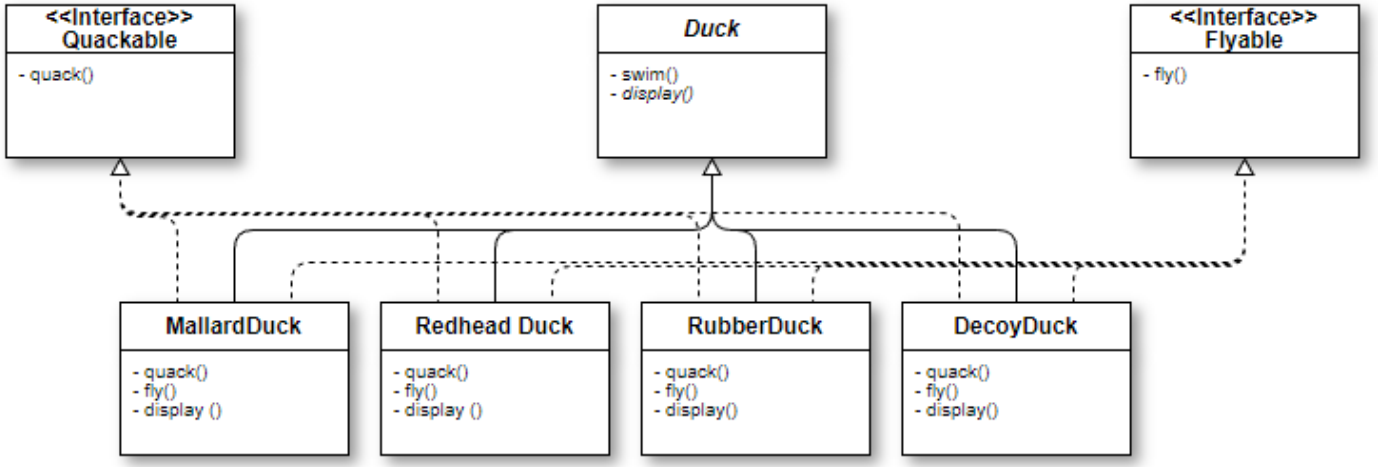
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Recuerda la constante del software el cambio

FUENTE: FREEMAN, E., FREEMAN E., HEAD FIRST DESIGN PATTERNS, O'REILLY

Simulador de Patos

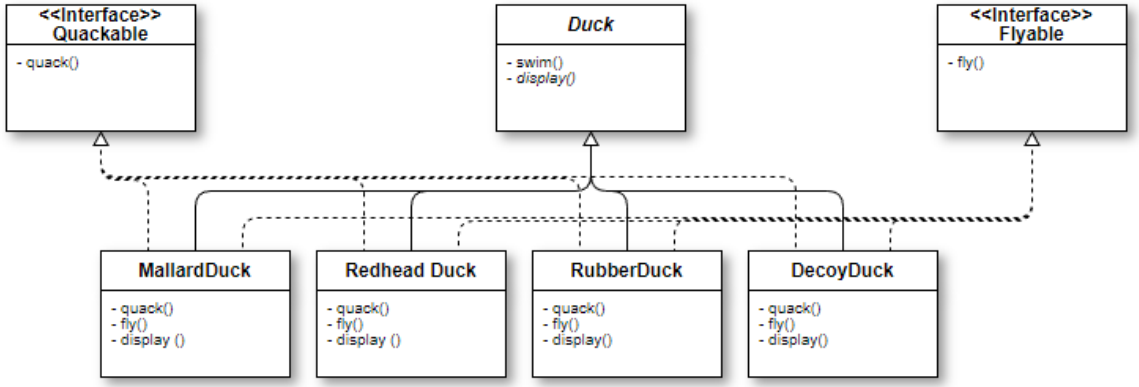
- El problema de la herencia múltiple:
  - Cambios constantes de clases
  - No todos vuelan y hacen quack
  - Modificar clases que se vean afectadas no es fácil
  - Efecto esperado → muchos errores



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

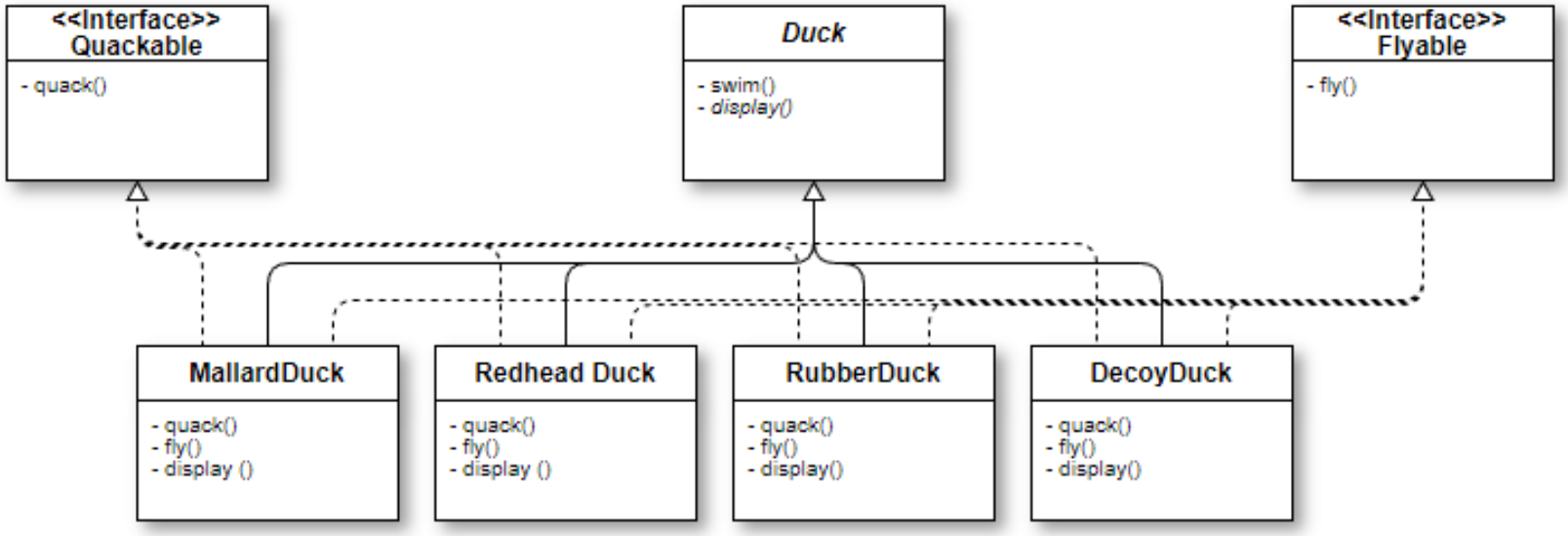
- **Principio de diseño:** Estrategia, encapsula los eventos que cambian y sepáralos del resto de la aplicación.
  - Asegurar que no nos afecten
  - Menor número de consecuencias por error en código
    - mayor flexibilidad



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

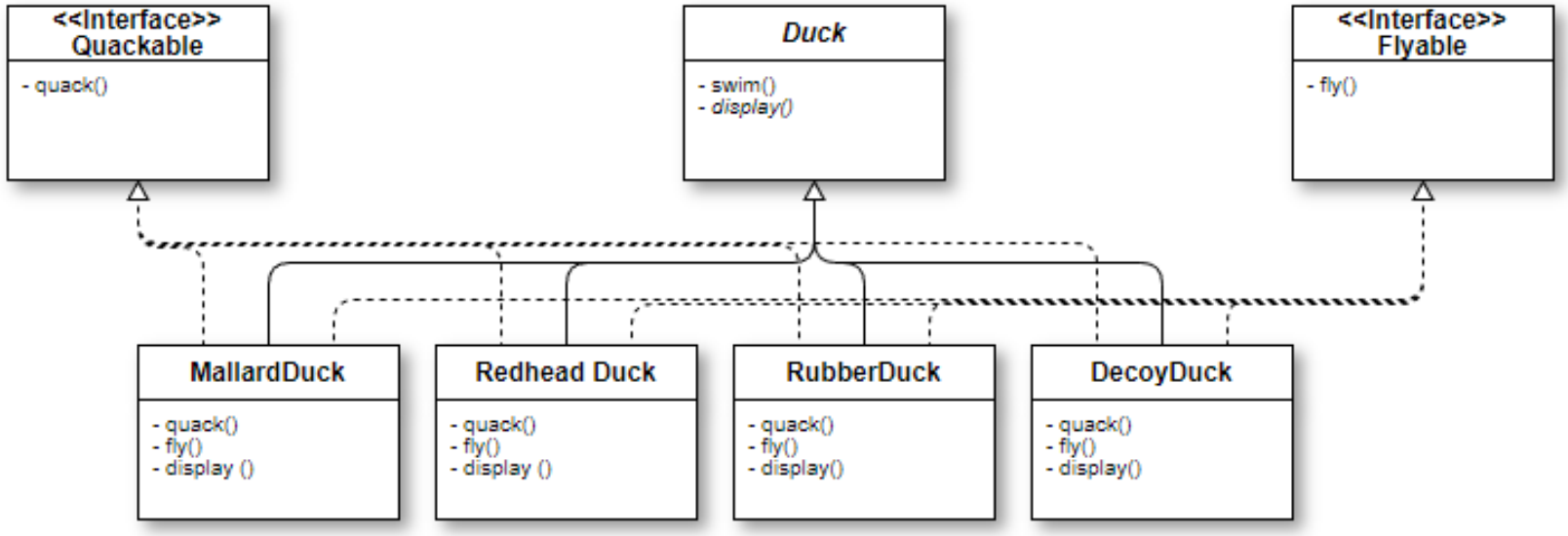
- ¿Qué eventos nos están afectando?



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

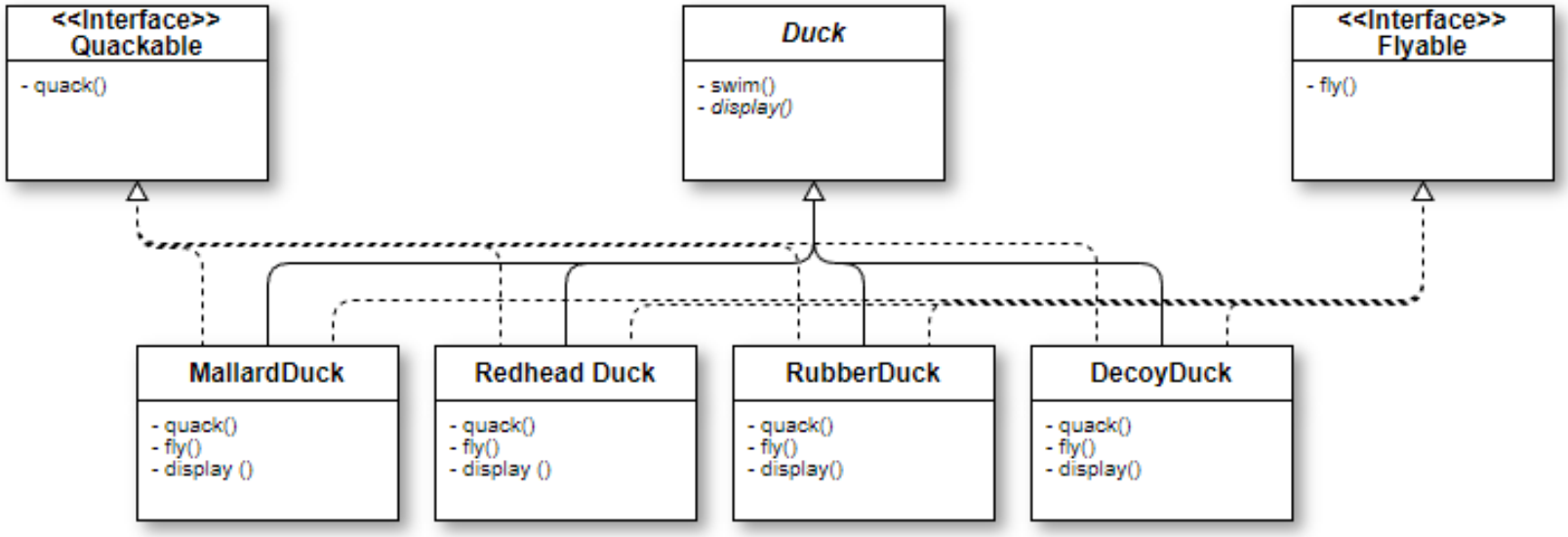
- ¿Qué eventos nos están afectando?
  - fly()
  - Quack()



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Simulador de Patos

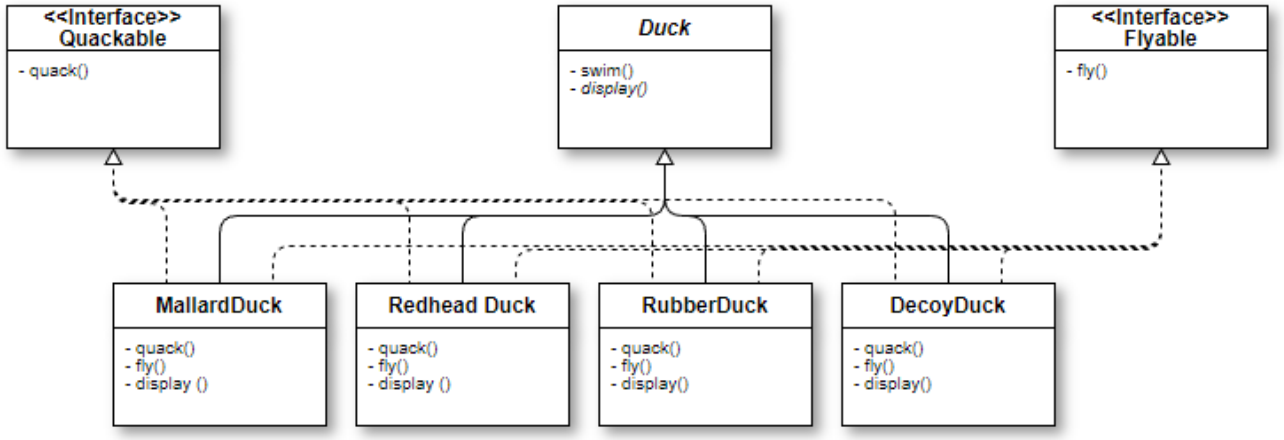
- Dejemos la clase Duck intacta y **separemos estos comportamientos**
  - **¿Cómo?**

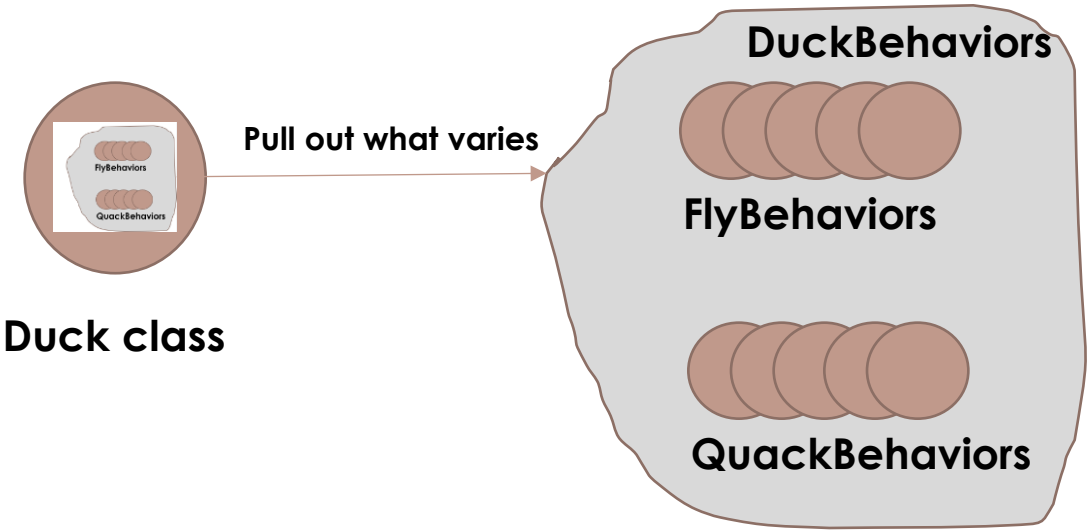


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

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 defina cada comportamiento.**
    - **Todos los comportamientos de vuelo (FlyBehaviors)**
    - **Todos los comportamientos de graznido (QuackBehaviors)**



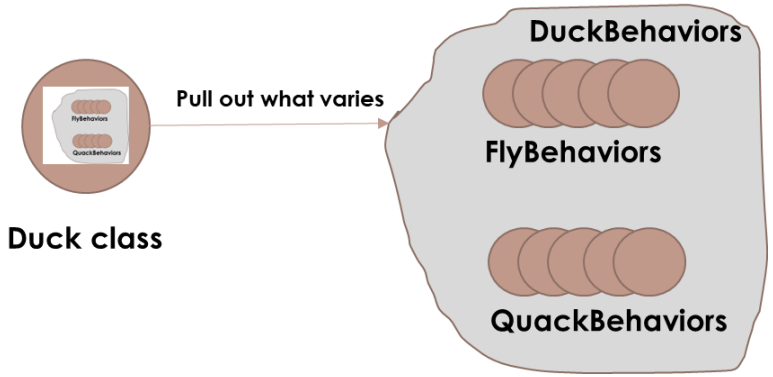


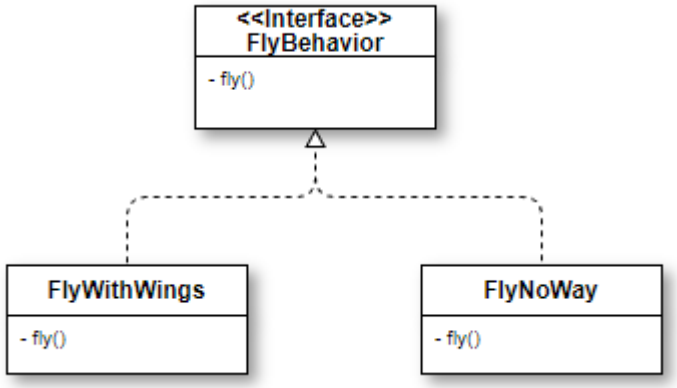
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Las Interfaces al rescate

MODELANDO EL COMPORTAMIENTO DE PATO

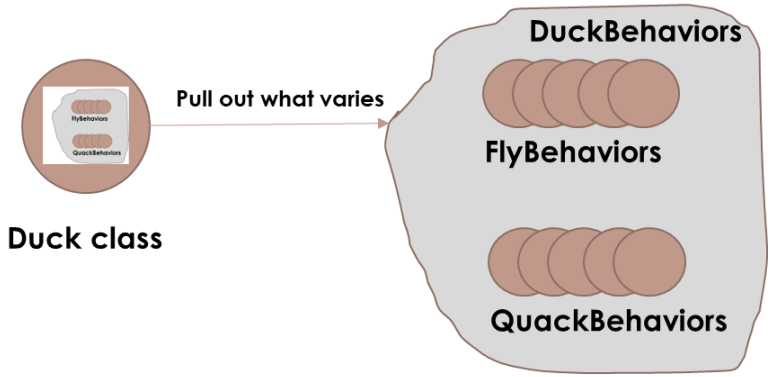
Diseñar el comportamiento del Pato

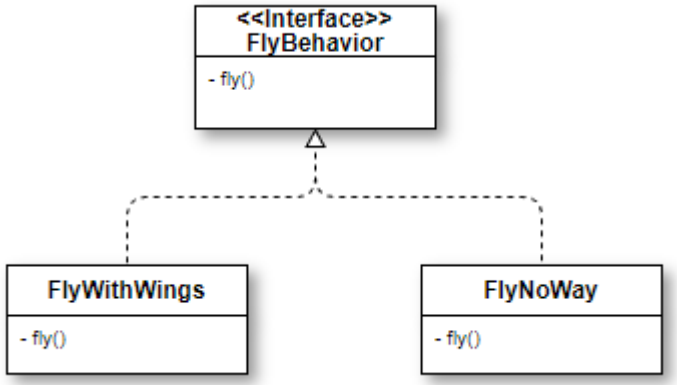




- El comportamiento del pato lo definimos en una clase separada
  - Una clase que implementa cada comportamiento
  - FlyBehavior
    - Vuela
    - No vuela
    - ....

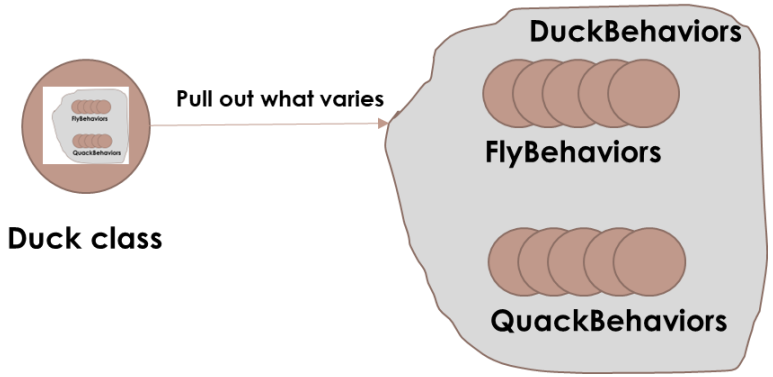
Diseñar el comportamiento del Pato

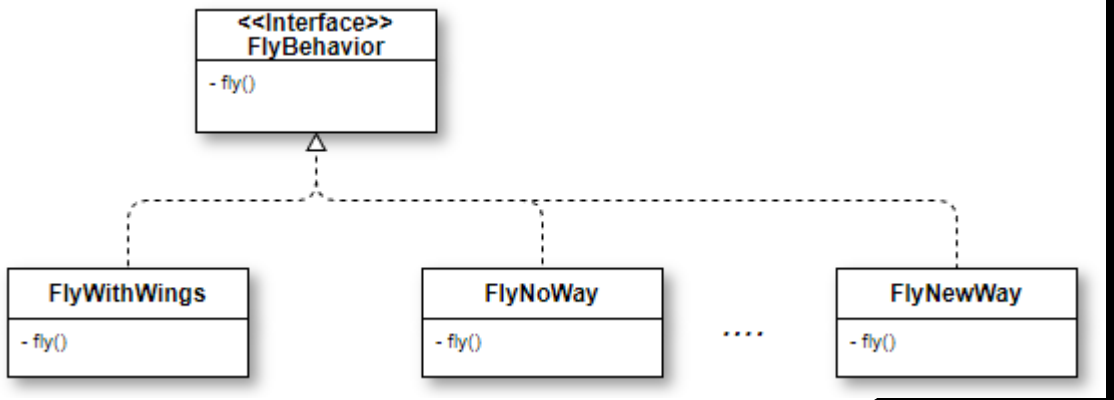




- El comportamiento del pato reside en una clase separada
  - Una clase que implementa el comportamiento de una forma particular
    - Volar
    - No hacer nada
- El pato no sabe nada de la implementación de sus comportamientos eso queda implementado en estas clases

Diseñar el comportamiento del Pato



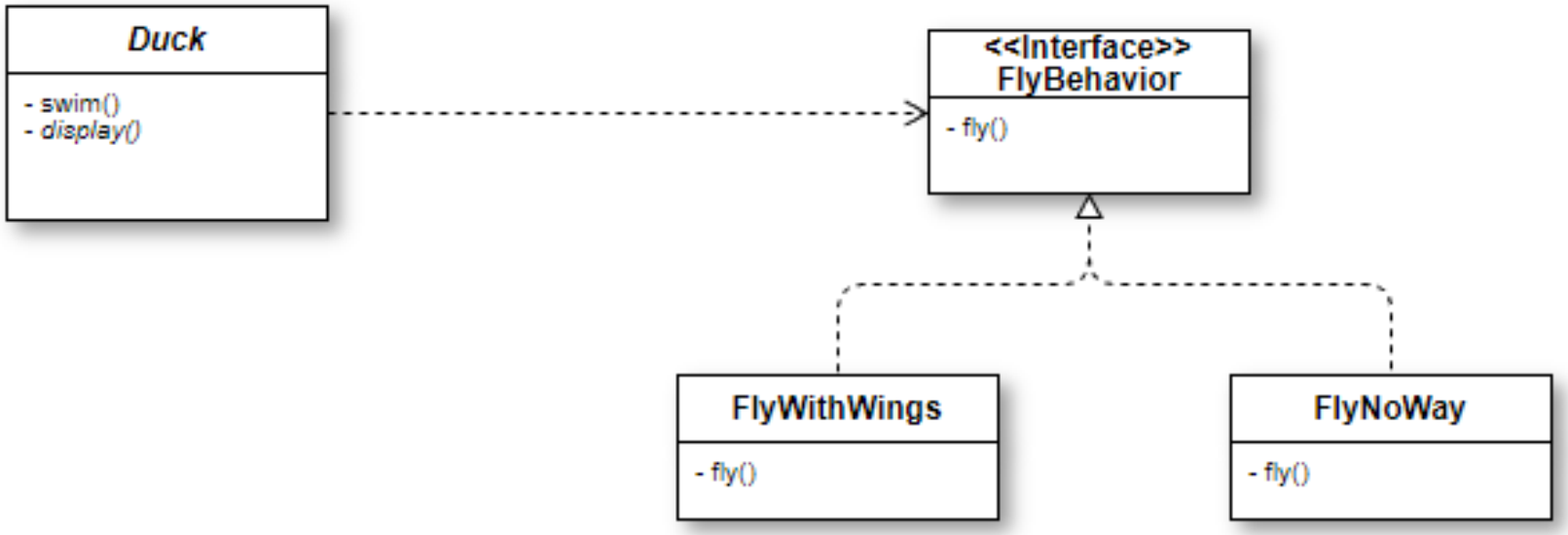


- El comportamiento del pato reside en una clase separada
  - Una clase que implementa el comportamiento de una forma particular
    - Volar
    - No hacer nada
- El pato no sabe nada de la implementación de sus comportamientos eso queda implementado en estas clases
- **Si el pato puede volar de otra forma la lista simplemente crece así como las clases que implementen los nuevos comportamientos**

Y entonces
¿Cómo sabe el pato
su comportamiento?

Diseñar el comportamiento del Pato

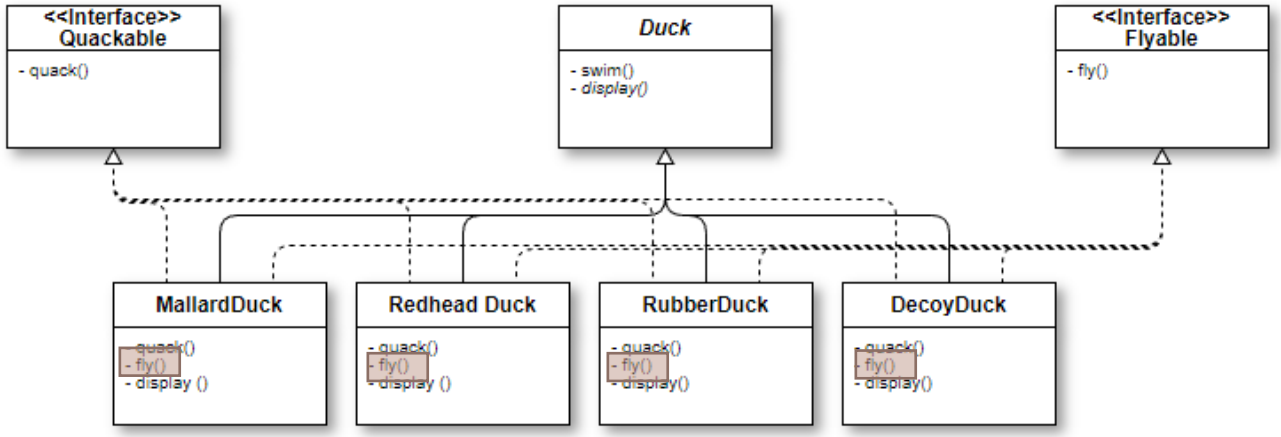
- **Principio de Diseño 2:** programa en las interfaces no en la implementación de una clase



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Un cambio con patrones no afecta mucho y corremos menos riesgos de equivocarnos

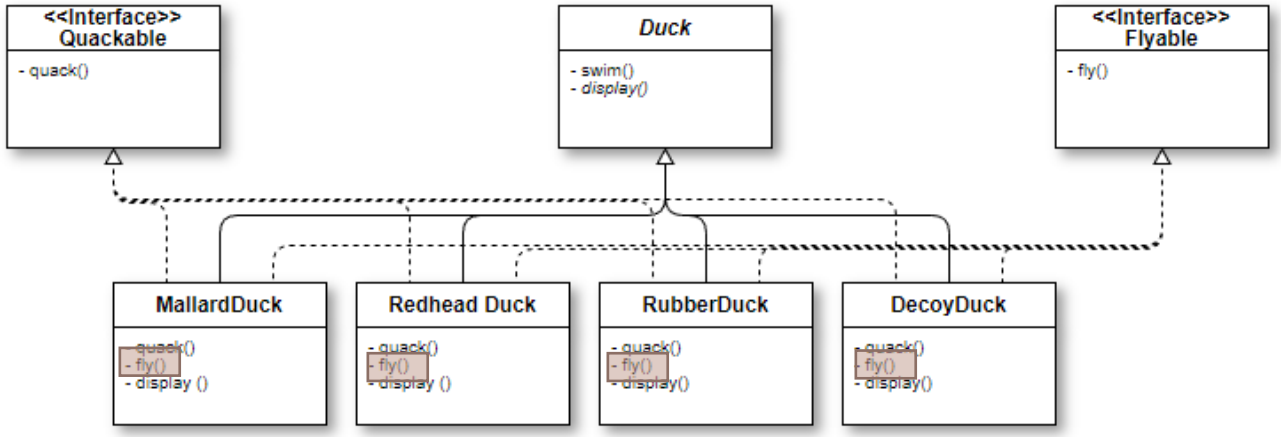
- Cambiar aquí es hacer más código y es difícil ubicarlo, al menos 4 lugares. Muchas más si más subclases se agregan.



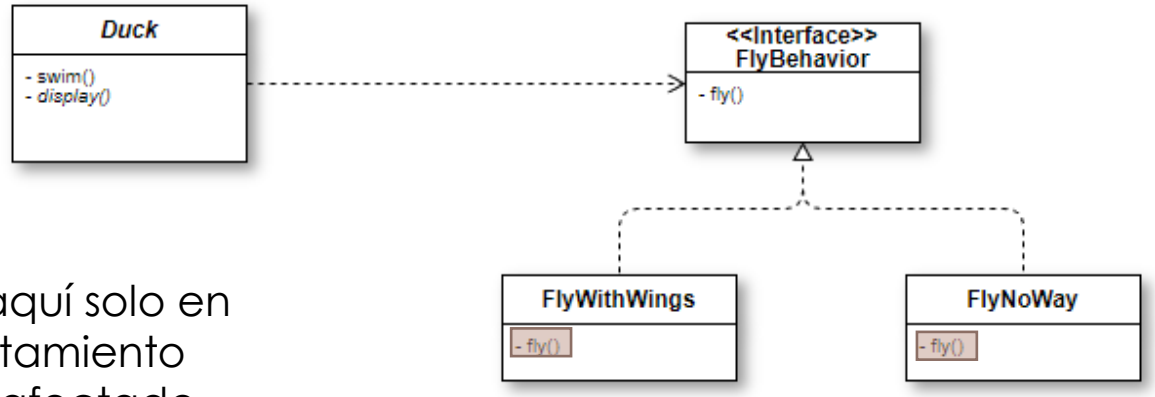
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Un cambio con patrones no afecta mucho y corremos menos riesgos de equivocarnos

- Cambiar aquí es hacer más código y es difícil ubicarlo, al menos 4 lugares. Muchas más si más subclases se agregan.



Ahora es más fácil de rastrear donde debo hacer cambios

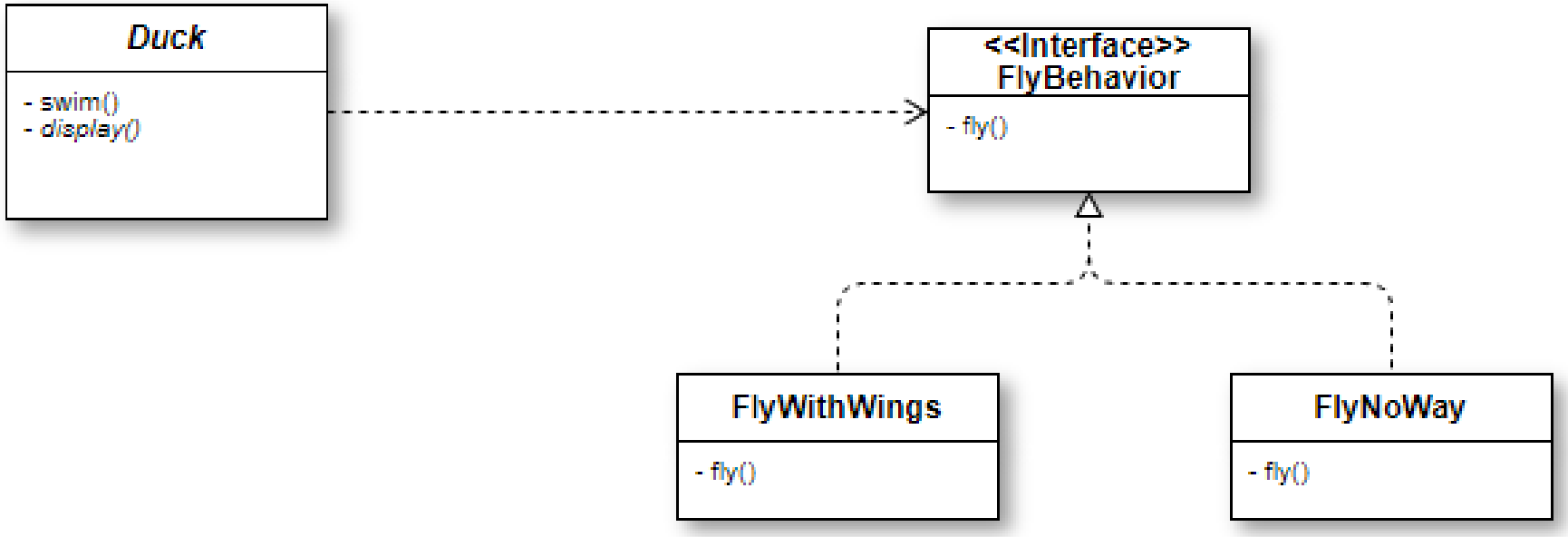


- Cambiar aquí solo en el comportamiento que se ve afectado.

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Diseñar el comportamiento del Pato

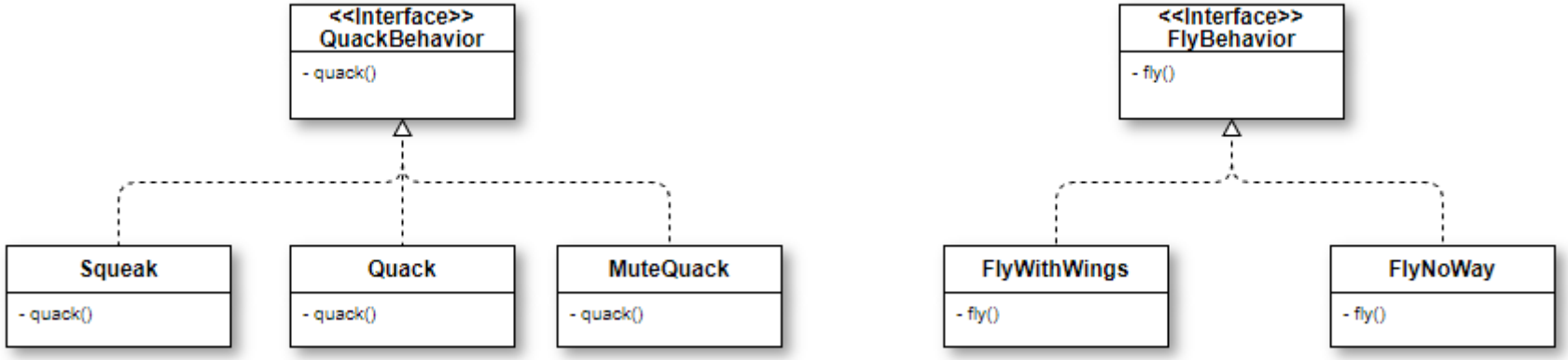
- El objetivo es encapsular el comportamiento,
- Explotar el polimorfismo programando supertipos (FlyBehaviour) en lugar del objeto usado en tiempo de ejecución (Duck)



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Implementación de cada comportamiento

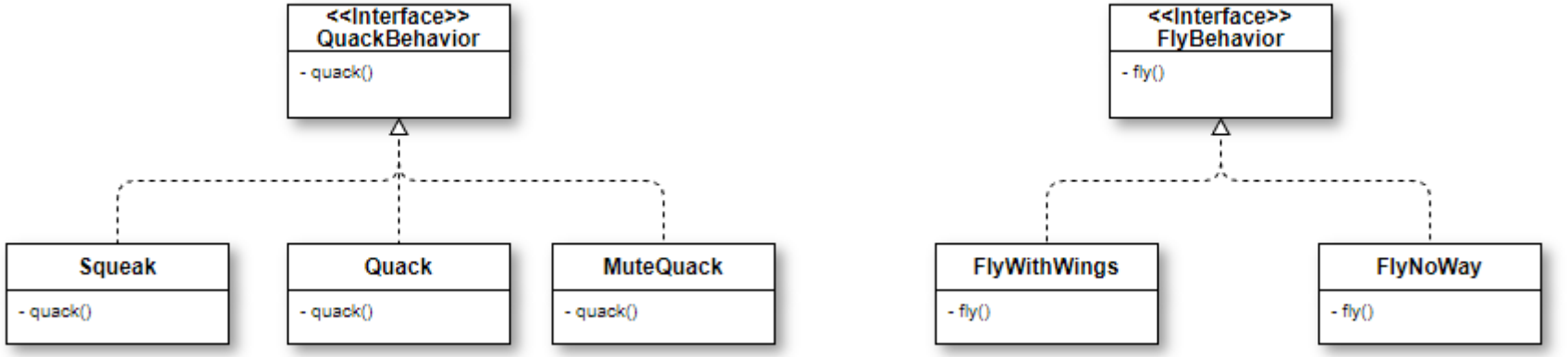
Interfaces que todas las subclases deben implementar



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Implementación de cada comportamiento

Interfaces que todas las subclases deben implementar

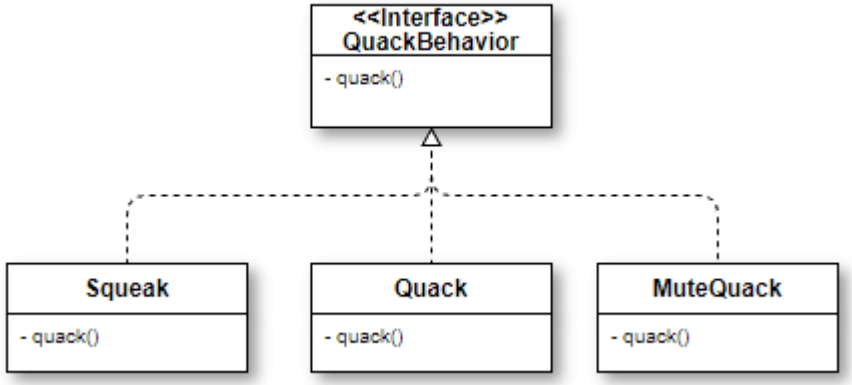


Patos con alas y sin alas

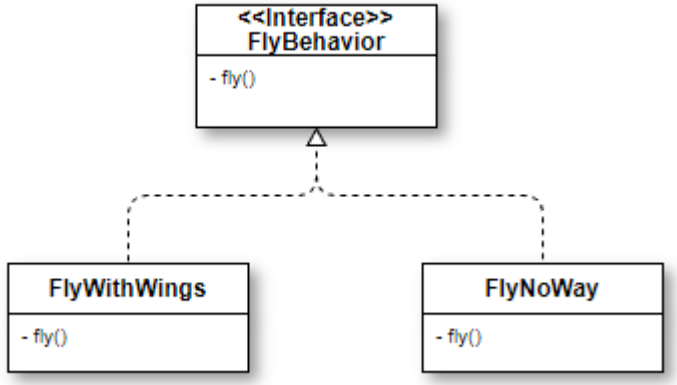
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Implementación de cada comportamiento

Interfaces que todas las subclases deben implementar



Los que hacen quack, chillan, son mudos y no hacen nada

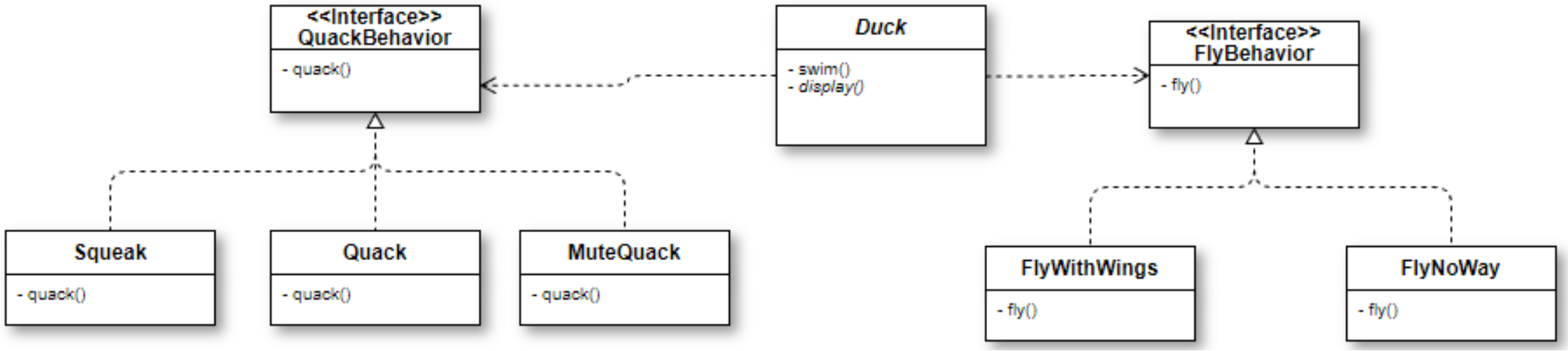


Patos con alas y sin alas

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Implementación de cada comportamiento

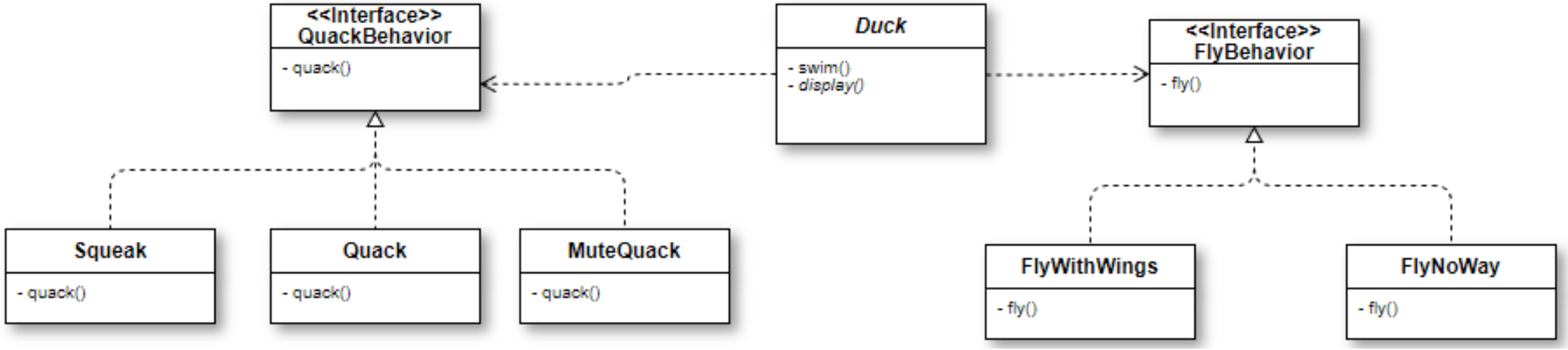
Ahora todos los patos podrán usar esto ya no se encuentra escondido en la clase particular de algún pato



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

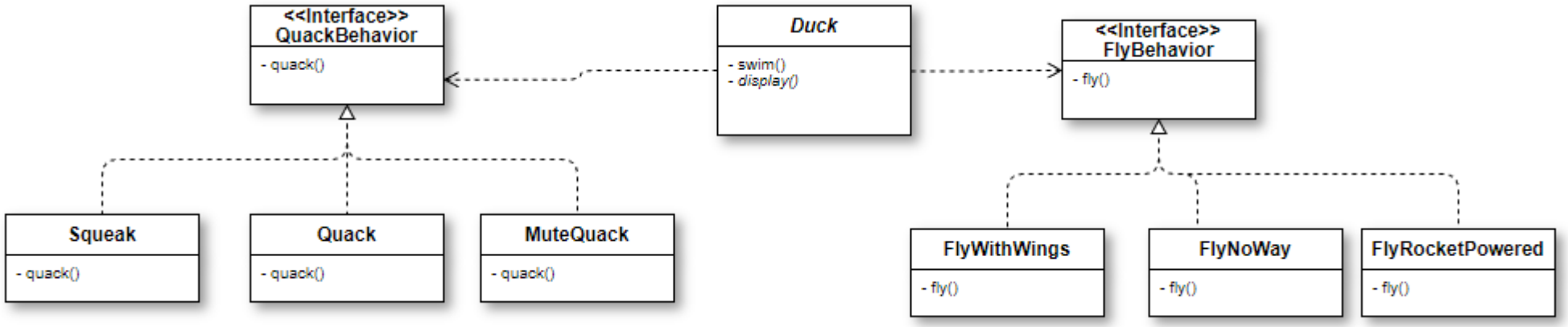
Preguntas Simples

- ¿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 nuestro simulador de patos?

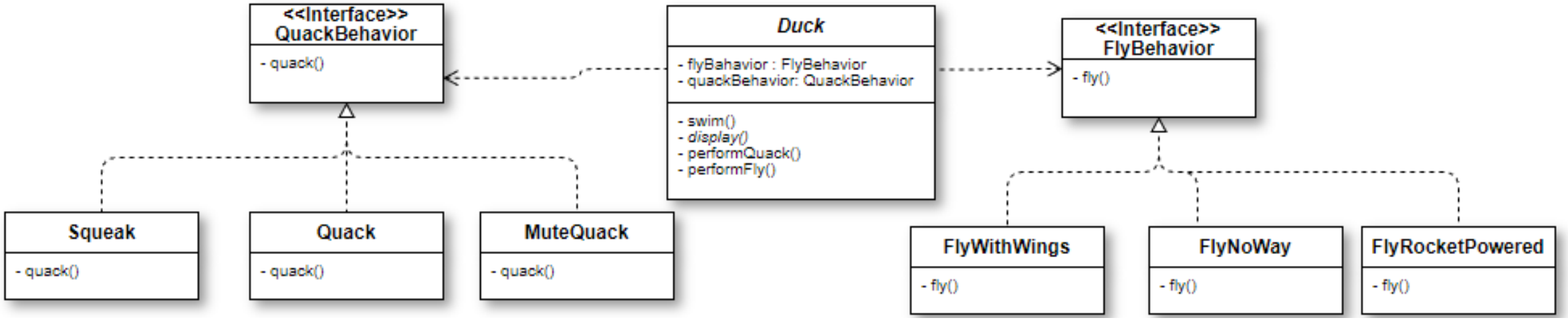
Preguntas Simples

- ¿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

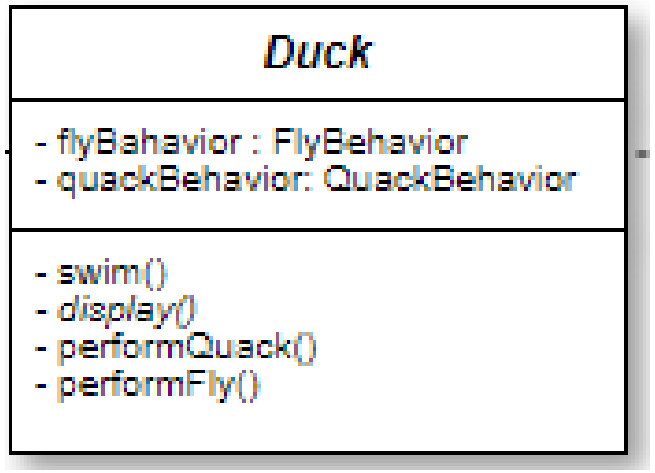
- Para que el pato haga quack y vuele debemos agregar variables que lo permitan a la clase Duck.



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Integrando el comportamiento al Pato

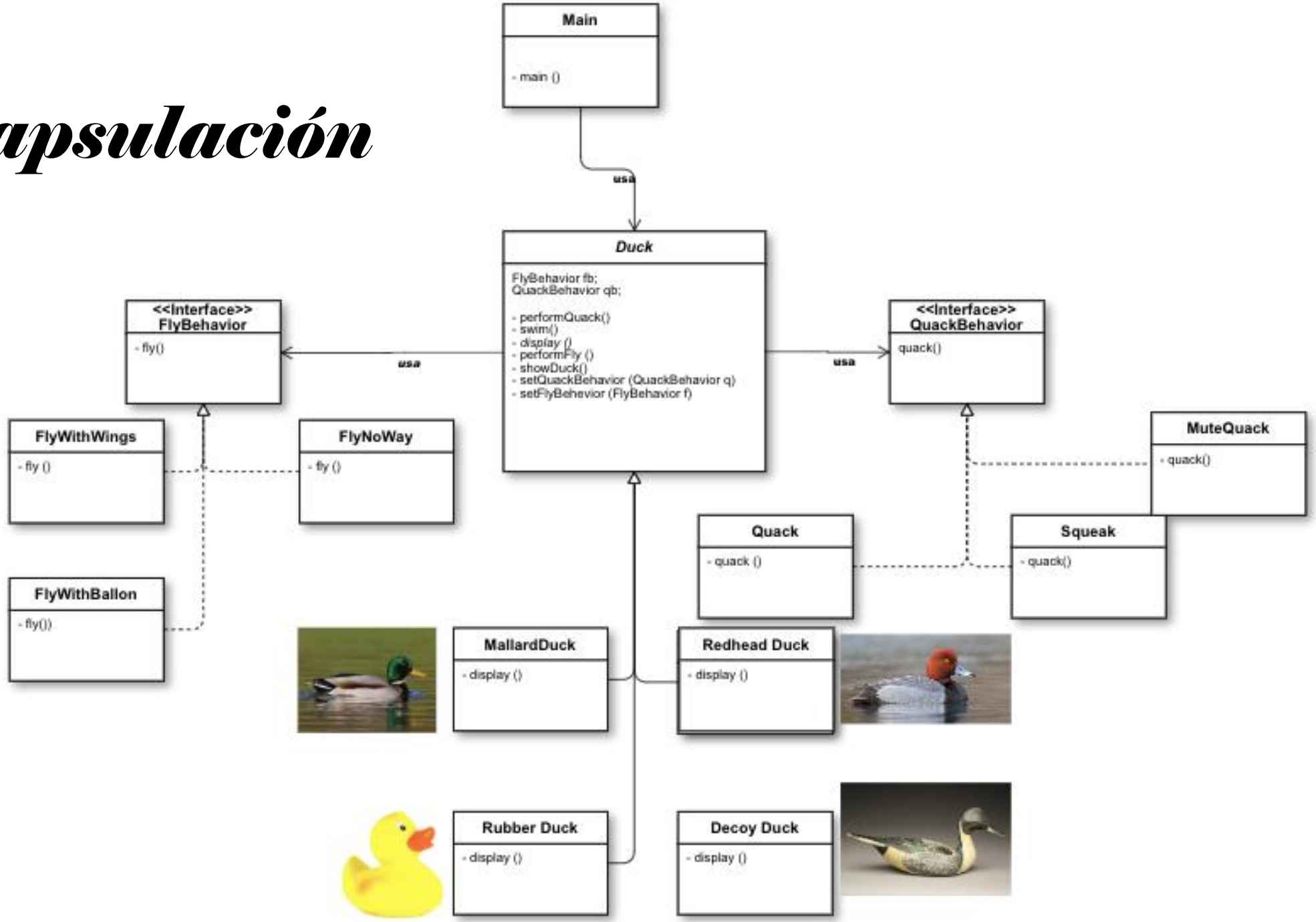
- Para que el pato haga quack y vuele debemos agregar variables que lo permitan




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

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

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

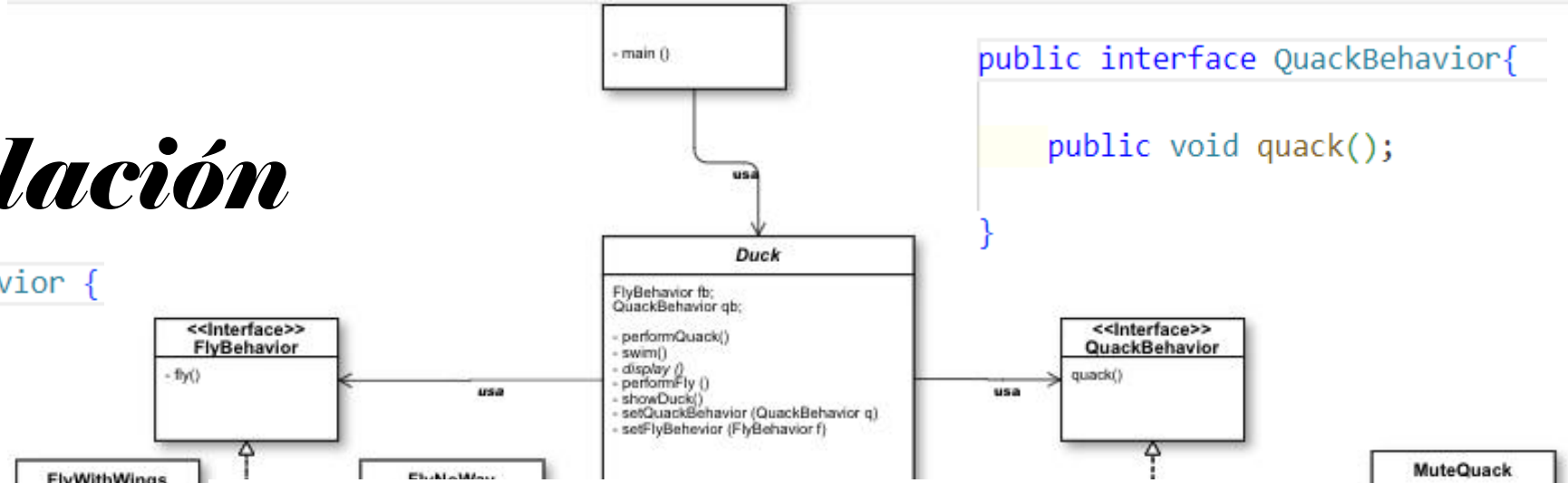
La encapsulación



Fuente: Freeman, E., I
Design Patterns, O'Reilly

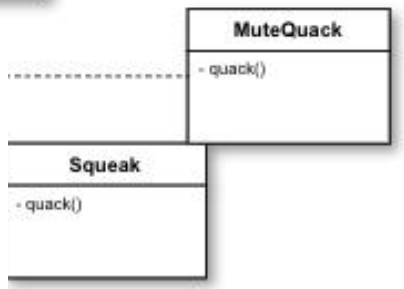
La encapsulación

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




```java
public interface QuackBehavior{

    public void quack();

}
```

```java
public abstract class Duck {

    QuackBehavior qb;
    FlyBehavior fb;

    public abstract void display();

    public void setQuackBehavior(QuackBehavior q){ ···

    public void setFlyBehavior(FlyBehavior f){ ···

    public void performQuack(){ ···

    public void performFly(){ ···

    public void swim(){ ···

    public void showDuck (){ ···
}
```




```java
public class FlyWithBallon implements FlyBehavior{      Juan Manuel Gonzalez Calleros,

    @Override
    public void fly() {
        System.out.println(x:"I'm flying with a ballon!");
    }
}
```

```java
public class Squeak implements QuackBehavior{

    @Override
    public void quack() {
        System.out.println(x:"Squeak");
    }

}
```

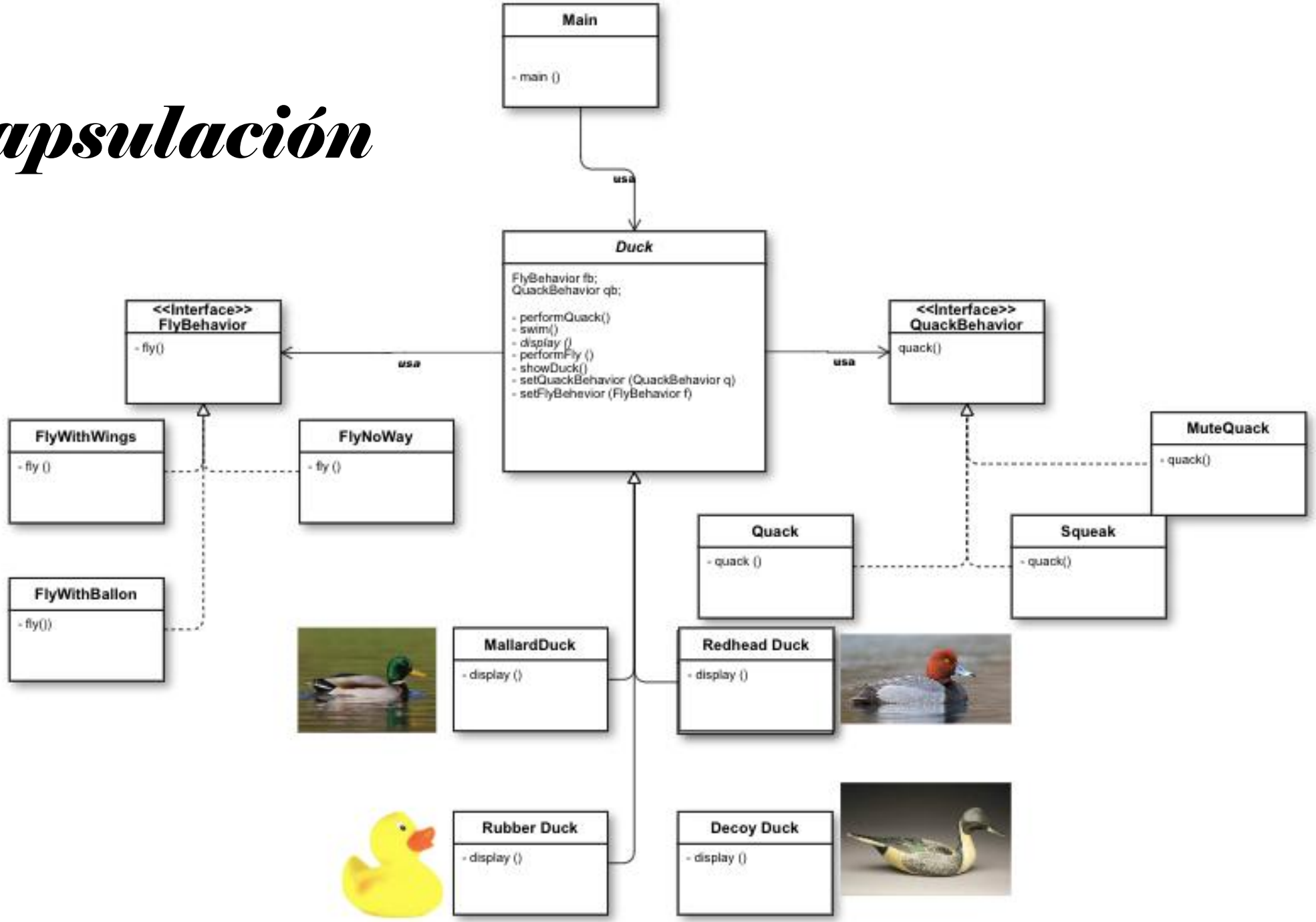
```java
c class Decoy extends Duck{

ublic Decoy() {
    qb = new MuteQuack();
    fb = new FlyNoWay();

Override
ublic void display() {
    System.out.println(x:"Decoy 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"
    - "squeze"
    - "no hago ruido"

- Hacer esto para los 4 DISPLAY de los patos del ejemplo
  - "Soy un Decoy"
  - "Soy un Rubber"
  - "Soy un Redhead"
  - "Soy un Mallard"

La encapsulación



Fuente: Freeman, E., I
Design Patterns, O'Reilly

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"
    - "squeze"
    - "no hago ruido"

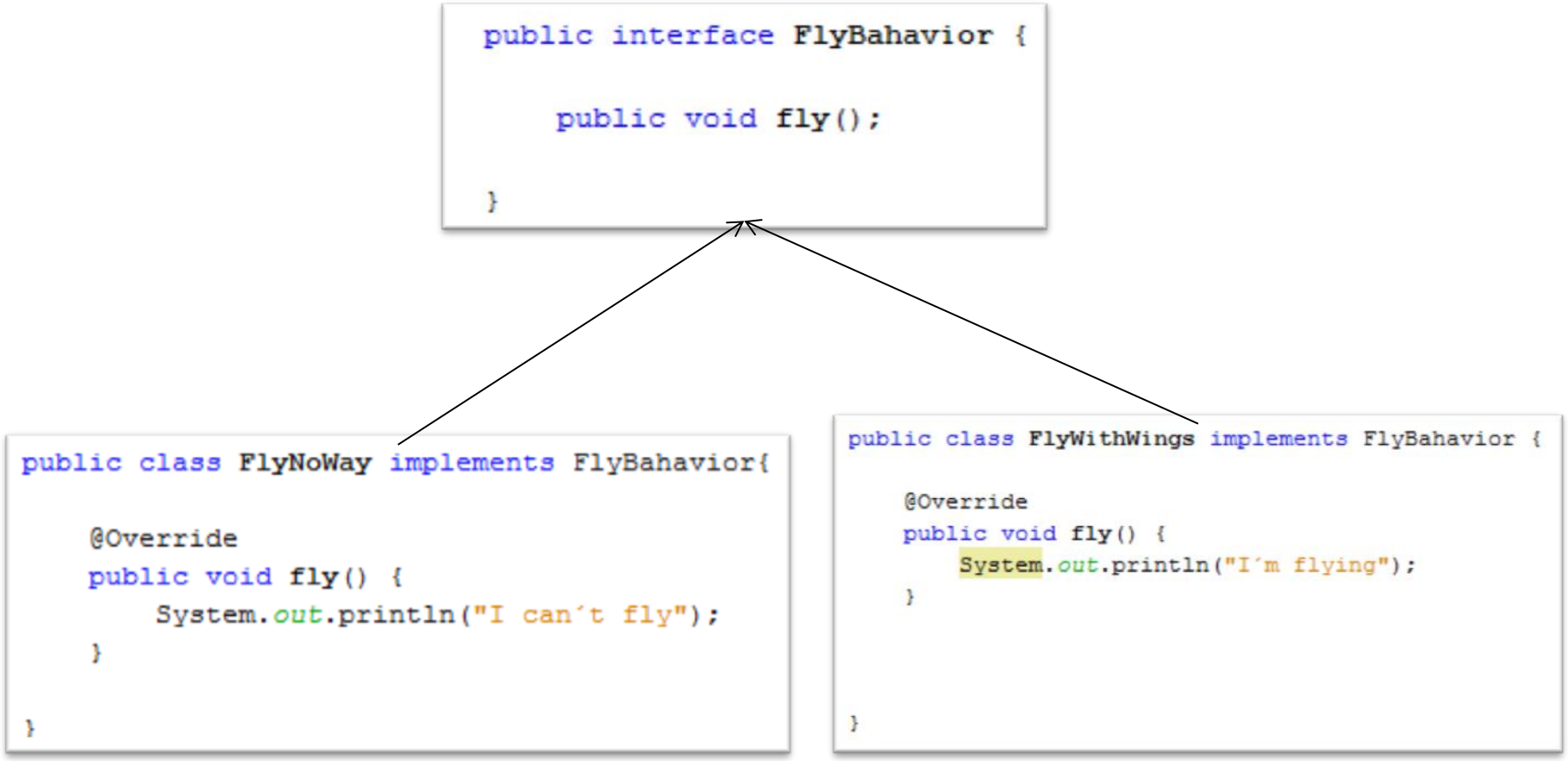
- Hacer esto para los 4 DISPLAY de los patos del ejemplo
  - "Soy un Decoy"
  - "Soy un Rubber"
  - "Soy un Redhead"
  - "Soy un Mallard"

Clase Duck

```
public abstract class Duck {
    FlyBahavior 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 ()
    {
        flyBehavior.fly();
    }

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

    }

    //Este método siempre es el mismo para todos
   public void swim (){
       System.out.println("All ducks float, even decoys!");
   }
   public void showDuck (){

        display();
        performFly();
        performQuack();
        swim();
   }

}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Interfaz de Vuelo



```java
public interface FlyBahavior {

    public void fly();

}
```

```java
public class FlyNoWay implements FlyBahavior{

    @Override
    public void fly() {
        System.out.println("I can´t fly");
    }

}
```

```java
public class FlyWithWings implements FlyBahavior {

    @Override
    public void fly() {
        System.out.println("I´m flying");
    }


}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Interfaz Graznido

```java
public interface QuackBehavior {

    public void quack();

}
```

```java
public class Quack implements QuackBehavior{

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


}
```

```java
public class MuteQuack implements QuackBehavior{

    @Override
    public void quack() {

        System.out.println("<<Silence>>");
    }

}
```

```java
public class Squeak implements QuackBehavior{

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


}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Prueba

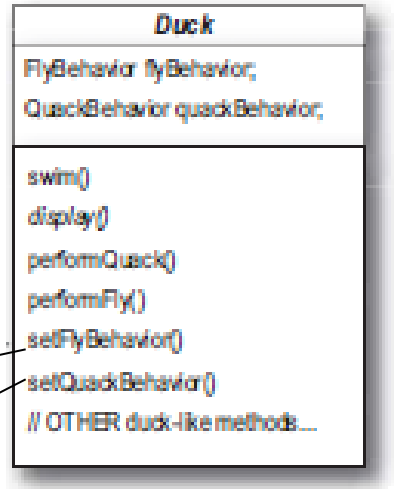
```
public class DuckTest {

    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!
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definamos el comportamiento de forma automática

```
public abstract class Duck {
    FlyBahavior 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(FlyBahavior fb){

    flyBehavior =fb;
}

public void setQuackBehavior(QuackBehavior qb){

    quackBehavior =qb;
}

}
```



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Vamos a crear un pato que no vuela

- Llámenle ModelDuck

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

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");
    }

}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

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");
    }

}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definamos ahora un comportamiento de vuelo con cohete

- Llámenle FlyRocketPowered

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definamos ahora un comportamiento de vuelo con cohete

- Llámenle FlyRocketPowered

```
public class FlyRocketPowered implements FlyBahavior{

    public void fly (){

        System.out.println("I´m flying with a rocket!");
    }
}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

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!
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

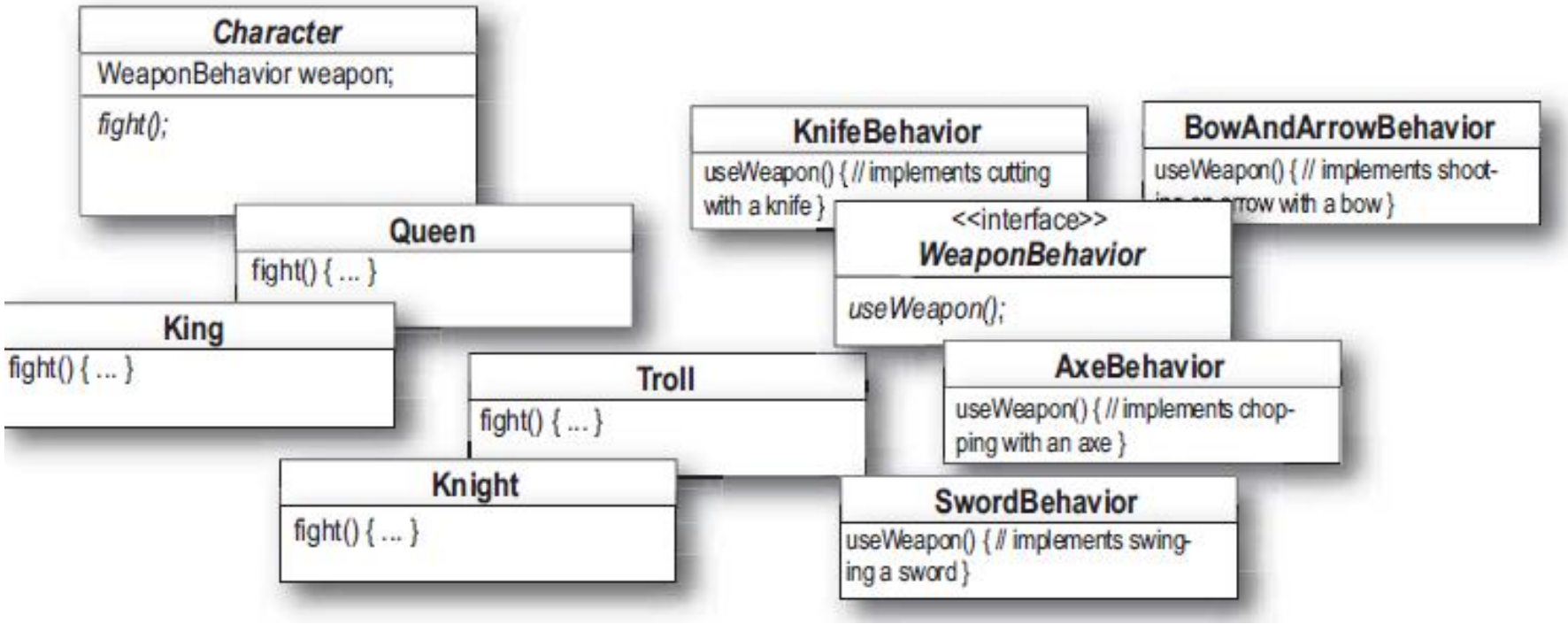
Ejercicio – Organiza el desorden

- 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

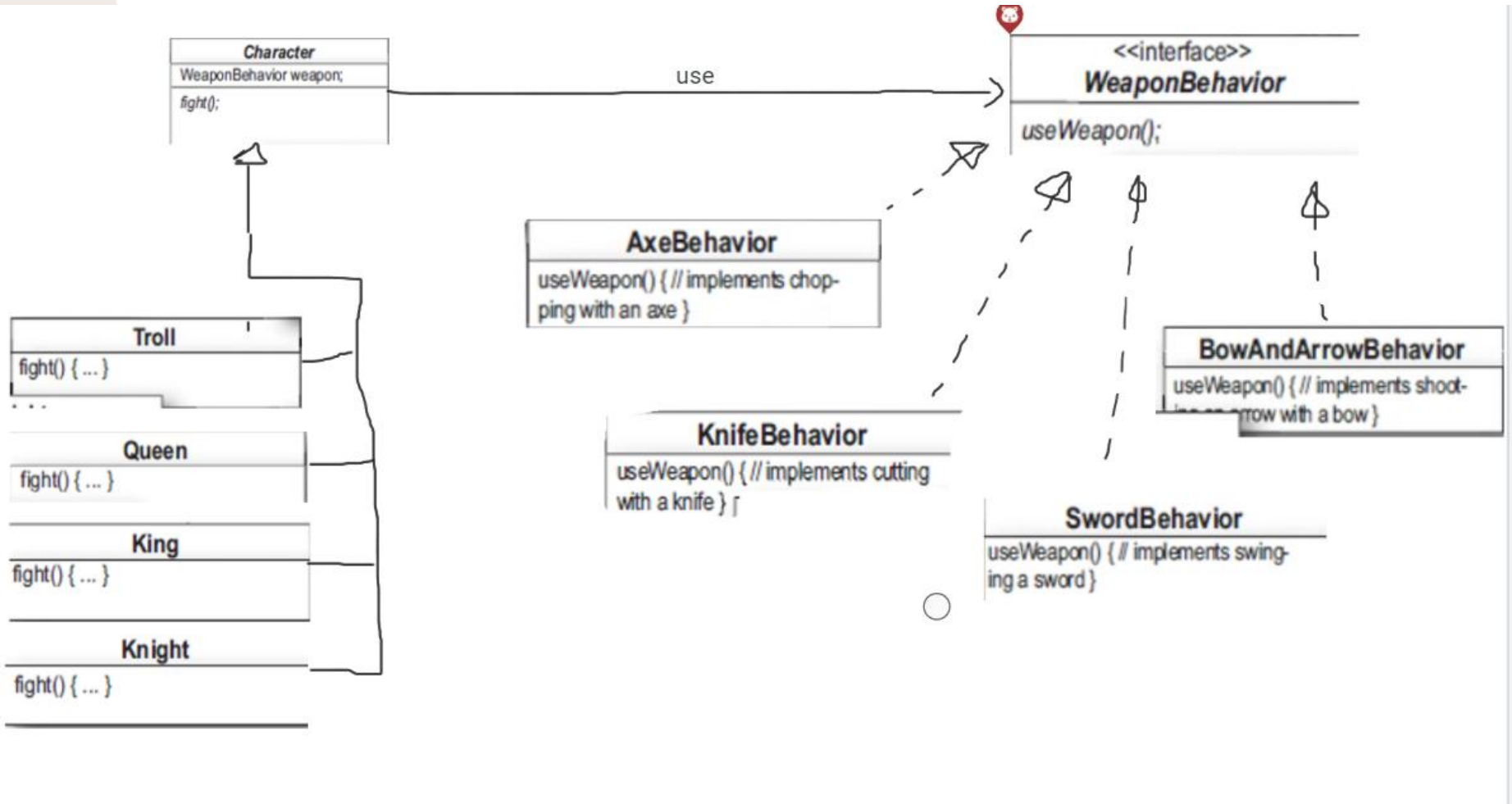
```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

Ejercicio – Organiza el desorden




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

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly


Character
WeaponBehavior weapon;
fight();
use
<<interface>>
WeaponBehavior
useWeapon();
AxeBehavior
useWeapon() { // implements chop-
ping with an axe }
Troll
fight() { ... }
Queen
fight() { ... }
King
fight() { ... }
Knight
fight() { ... }
KnifeBehavior
useWeapon() { // implements cutting
with a knife }
BowAndArrowBehavior
useWeapon() { // implements shoot-
row with a bow }
SwordBehavior
useWeapon() { // implements swing-
ing a sword }

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

https://github.com/jumagoca78/arquitectura-de-Software-/tree/main/DuckSimulator

Patrones de Diseño - Observador

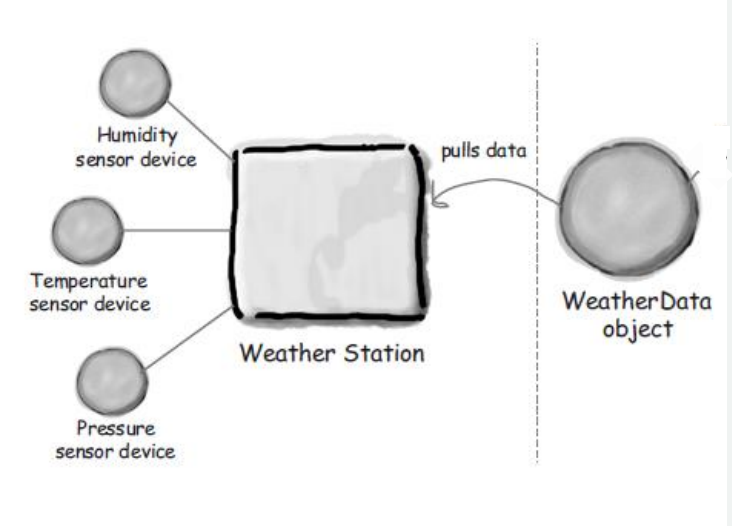
+ Dra. Josefina Guerrero García
+ Dr. Juan Manuel Gonzalez Calleros

Patrón Observador

Patrón comportamental

Escenario de Uso

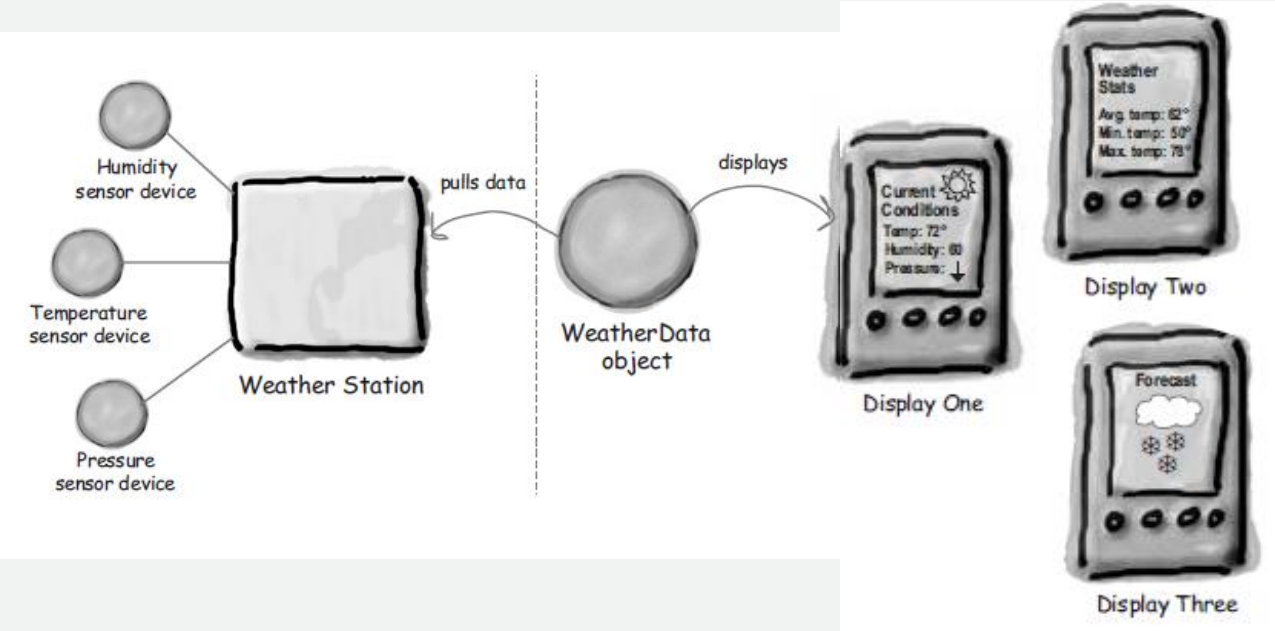
- En una estación en la sierra medimos tres variables asociadas con el clima:
  - Humedad (Humidity)
  - Temperatura (Temperature)
  - Presión (Pressure)



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Escenario de Uso

+ En una estación en la sierra medimos tres variables asociadas con el clima:
  + Humedad (Humidity)
  + Temperatura (Temperature)
  + Presión (Pressure)



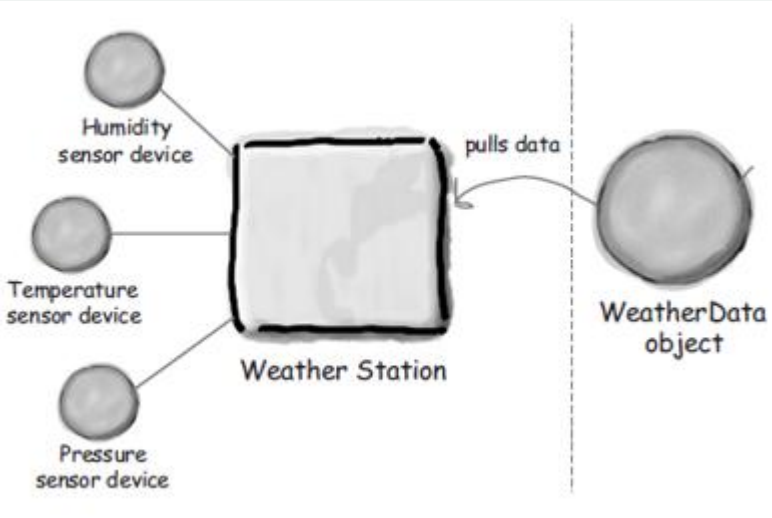
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

+ **Necesitamos hacer unas aplicaciones para comunicar al usuario**:
  + Condiciones actuales (current conditions), lectura actual de las tres variable
  + Estadísticas de clima (Statistics),, mínimo, máximo, promedio de la temperatura
  + Predicción (Forecast), cambios en la presión comunican mejor clima o mal clima

La clase Clima

+El cómo se manipulan los sensores no es nuestro problema, solo nos interesa manipular los datos

+¿Cómo quedaría la clase?

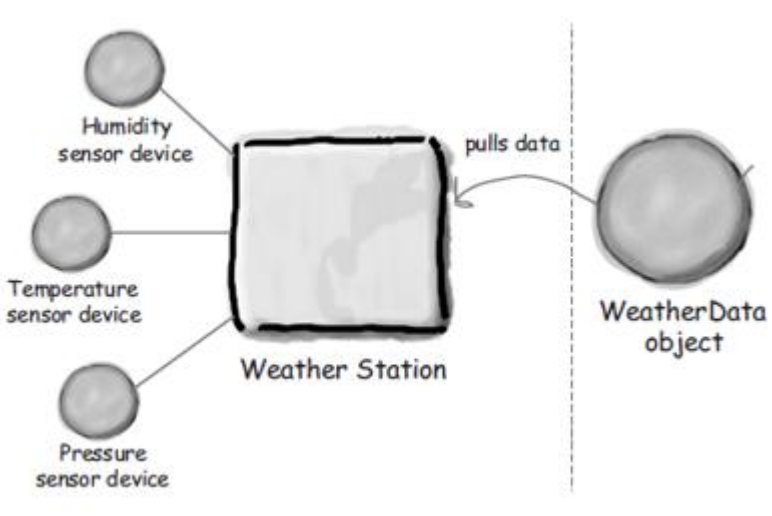


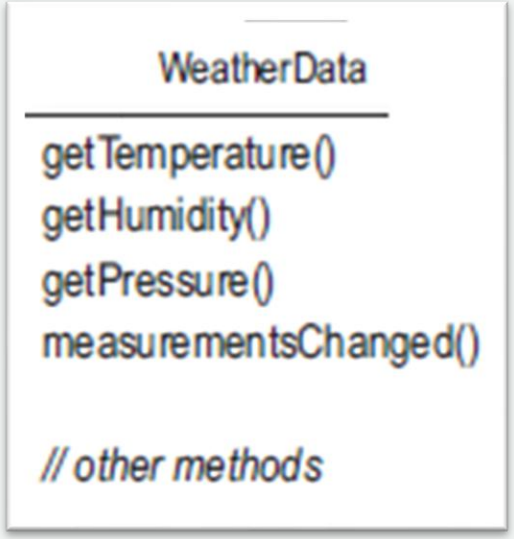
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

La clase Clima

+El cómo se manipulan los sensores no es nuestro problema, solo nos interesa manipular los datos

+¿Cómo quedaría la clase?



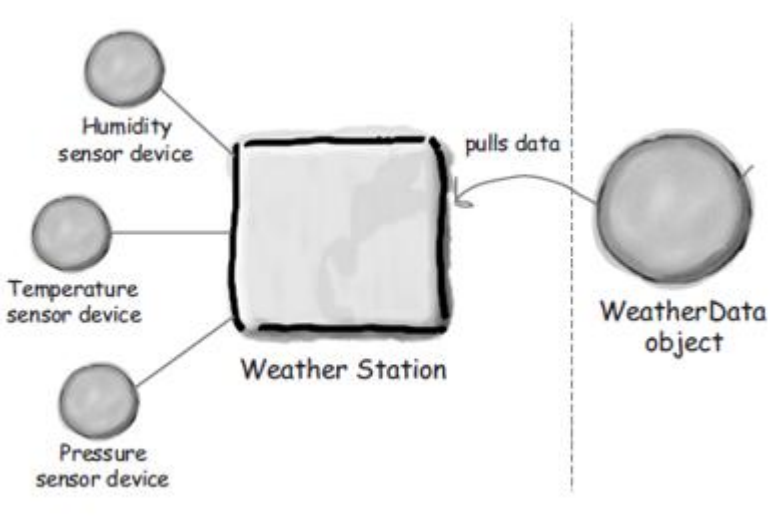


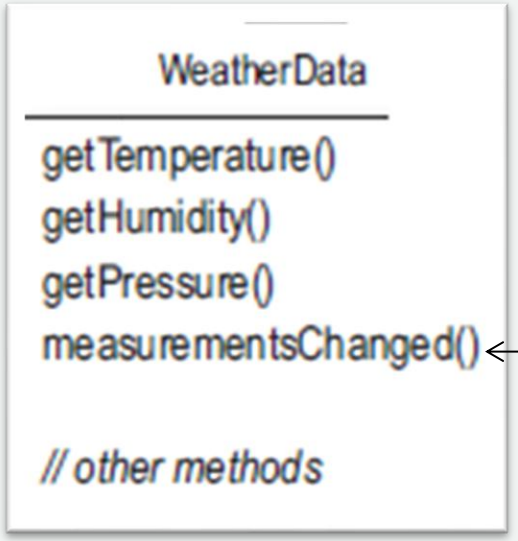
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

La clase Clima

+El cómo se manipulan los sensores no es nuestro problema, solo nos interesa manipular los datos

+¿Cómo quedaría la clase?






```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

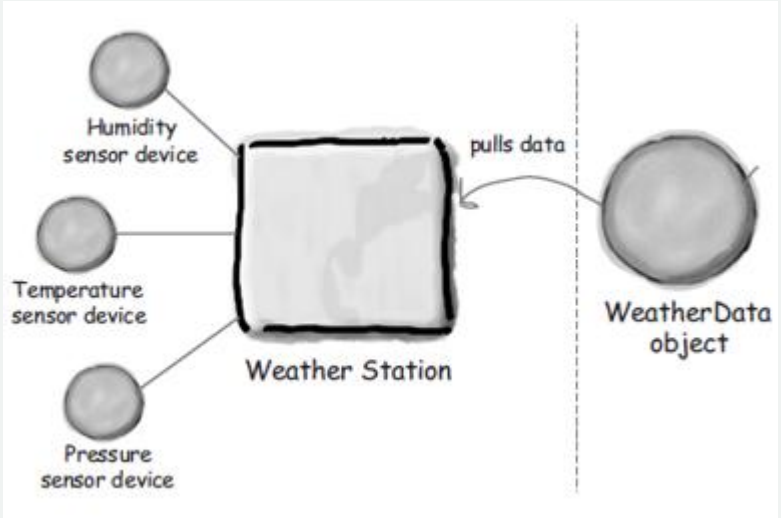
Cada que hay un cambio en una variable tenemos que avisar que las medidas cambiaron

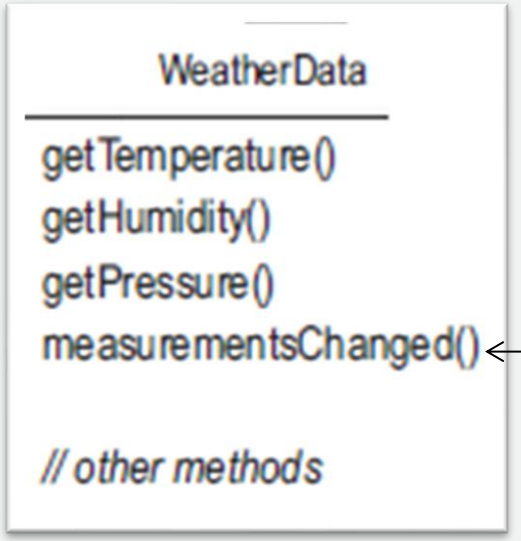
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

La clase Clima

+El cómo se manipulan los sensores no es nuestro problema, solo nos interesa manipular los datos

+¿Cómo quedaría la clase?





Actualización impacta los displays

- Condiciones actuales
- Estadísticas de clima
- Previsión

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Qué sabemos al momento

+Métodos para obtener el valor de tres variables

```
getTemperature()
getHumidity()
getPressure()
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Qué sabemos al momento

+ Métodos para obtener el valor de tres variables

```
getTemperature()
getHumidity()
getPressure()
```

+ Método que se llama cada que hay un cambio en alguna variable, solo sabemos que se manda a llamar

```
measurementsChanged()
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Qué sabemos al momento

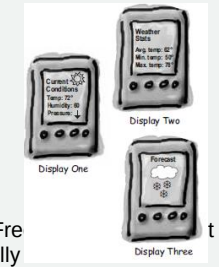
+ Métodos para obtener el valor de tres variables

```
getTemperature()
getHumidity()
getPressure()
```

+ Método que se llama cada que hay un cambio en alguna variable, solo sabemos que se manda a llamar

```
measurementsChanged()
```

+ Necesitamos implementar tres ventanas cada que se invoca el método



Fuente: Freeman, E., Fre t
Design Patterns, O'Reilly

Además

+El sistema debe ser expandible- otros desarrolladores podrían querer hacer sus propias ventanas y hacer tantas variantes como quieran, actualmente, conocemos sólo tres

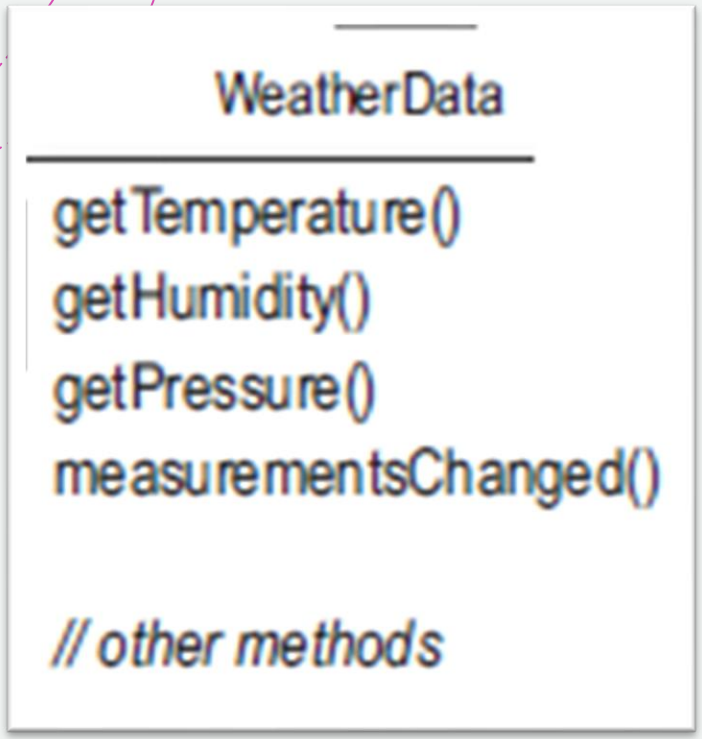
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Apps que podrían requerir del valor de nuestras tres variables

- Calendario con clima
- Agenda con aviso de ropa recomendada según tipo de clima

……

¿Funcionará el código actual?

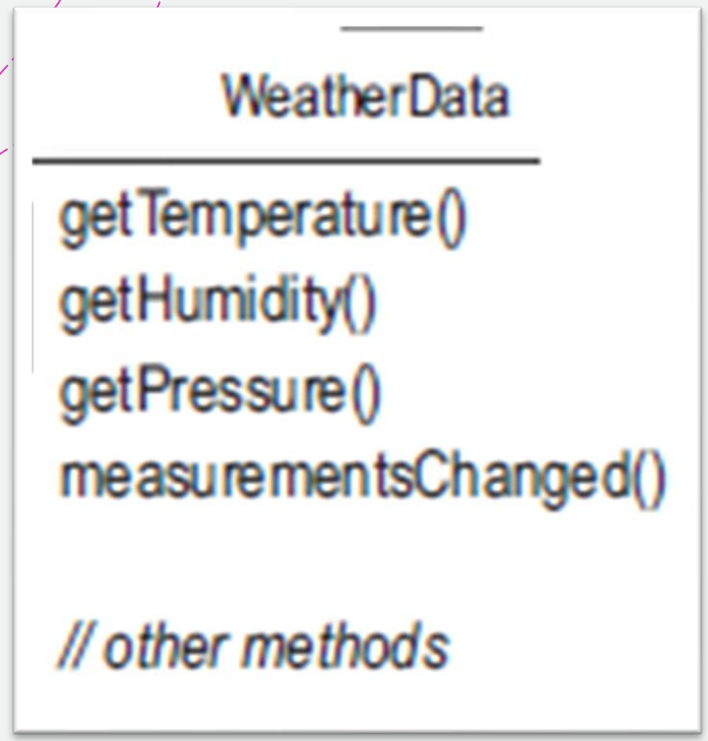


¿Programa la interfaz o la implementación?
Es decir, ¿Para cada Display tenemos que modificar el código?

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

¿Funcionará el código actual?



¿Programa la interfaz o la implementación?
Es decir, ¿Para cada Display tenemos que modificar el código?

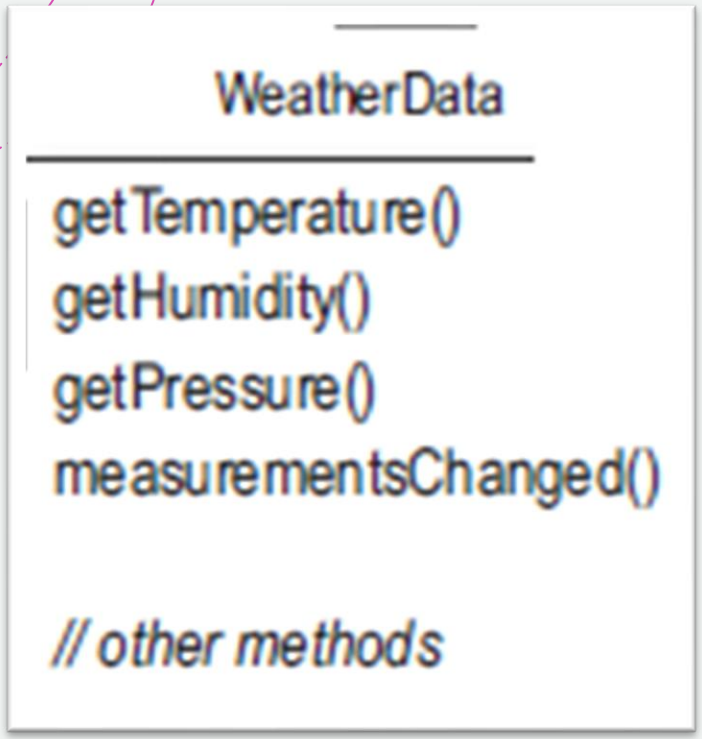
```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

En principio no pues cada display hará lo que su método update determine

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

¿Funcionará el código actual?

¿Programa la interfaz o la implementación?
Es decir, ¿Tenemos forma de agregar o quitar elementos de los display?

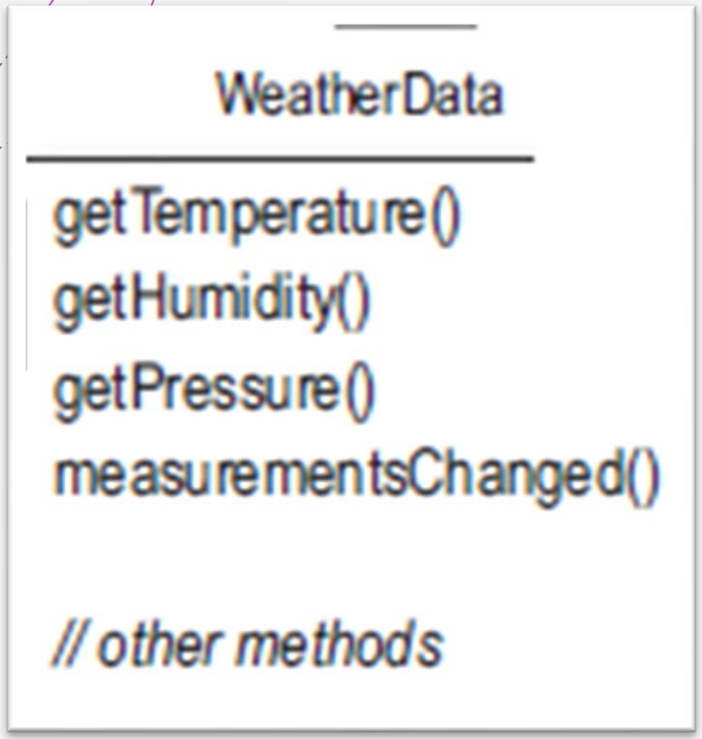



```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

¿Funcionará el código actual?

¿Programa la interfaz o la implementación?
Es decir, ¿Tenemos forma de agregar o quitar elementos de los display?

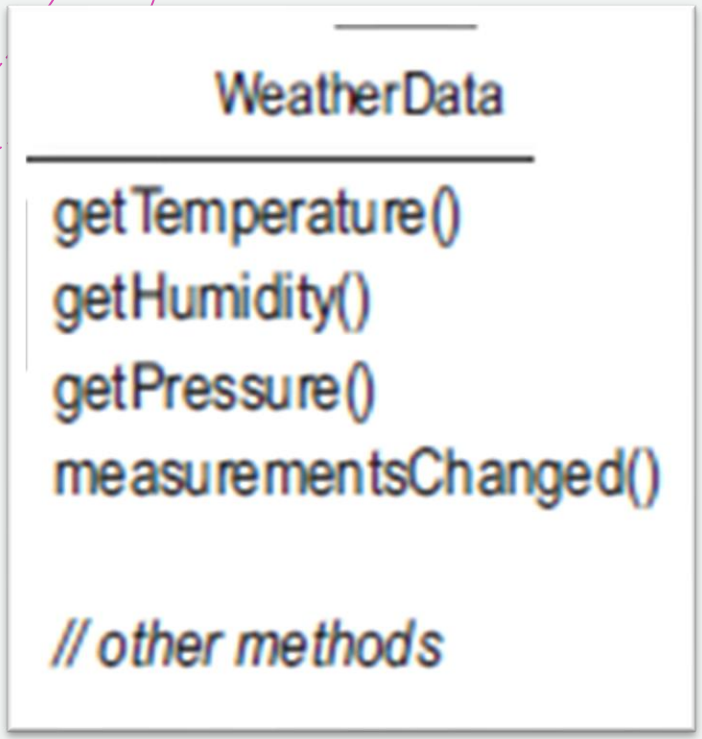



```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

En principio si y dependerá de lo que programemos en su método update

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

¿Funcionará el código actual?



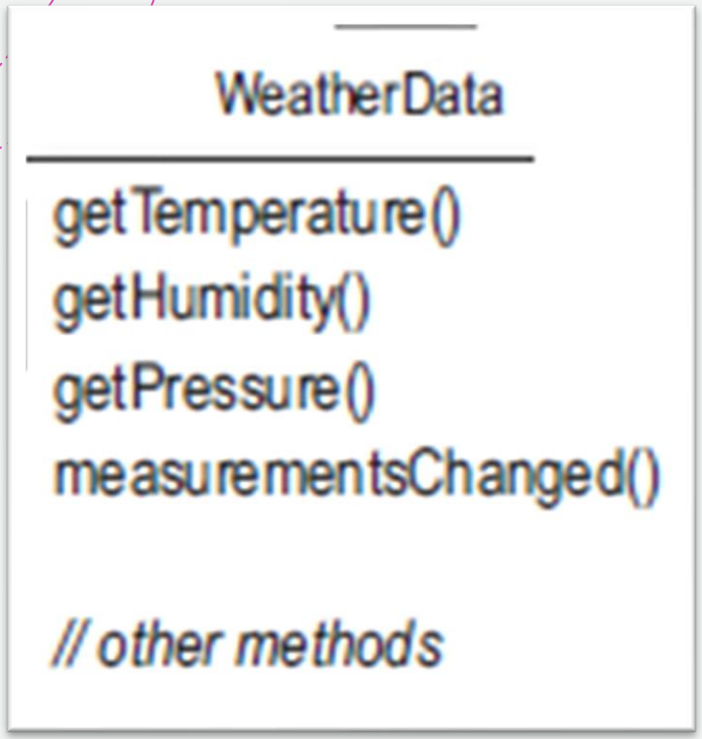
¿Programa la interfaz o la implementación?
Es decir, ¿Esta encapsulada la parte que cambia?

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

¿Funcionará el código actual?

¿Programa la interfaz o la implementación?
Es decir, ¿Esta encapsulada la parte que cambia?




```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Seguro si
Existe una interfaz que sabe actualizar y usa los mismos parámetros

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

¿Qué falla?

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

¿Qué falla?

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

- Si necesitamos nuevas apps o alguna de las tres ya no se usa, tenemos que venir a reescribir esto

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Necesitamos encapsular la lista de apps

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

¿Cómo?

Patrón Observador

+Usando de referente la subscripción a una revista o periódico

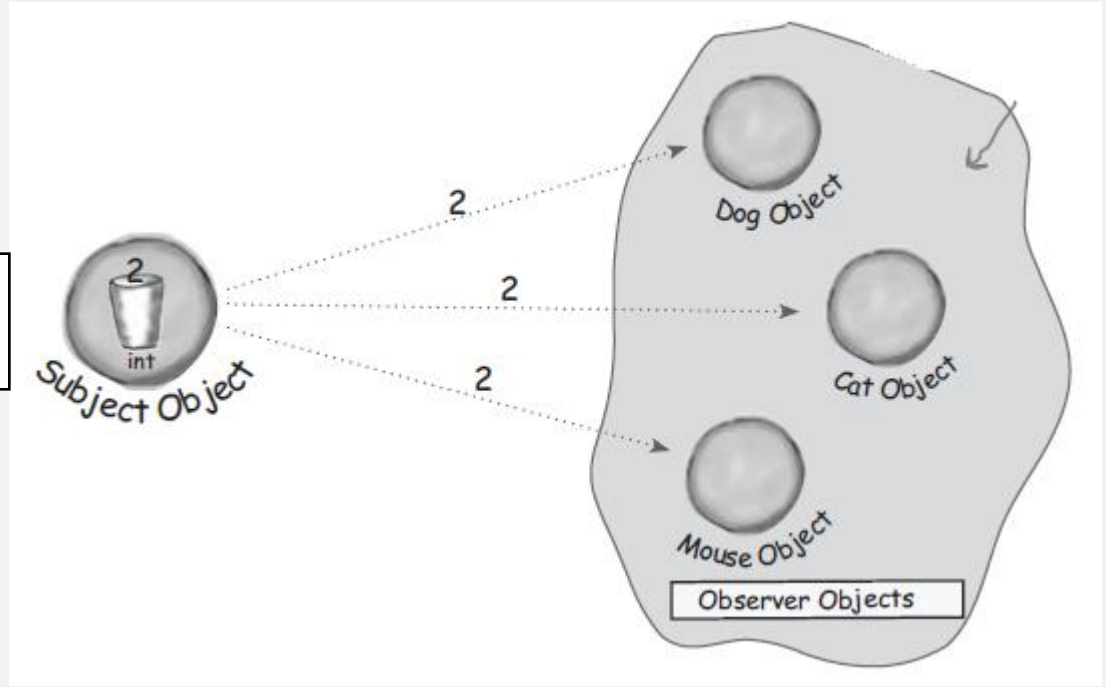
1. Una editorial entra al negocio y funda su publicación
2. Te suscribes a una editorial en particular, y cada que hay una nueva versión llega a ti, tanto como sigas registrado en la subscripción
3. Si te desafilias, la publicación ya no te llega
4. Mientras la editorial siga en el negocio, todo tipo de interesados se registran y dan de baja.

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Editorial + Suscriptores = Patrón Observador

+La editorial es el **sujeto** (subject) y los suscriptores los **observadores**

Maneja algún dato

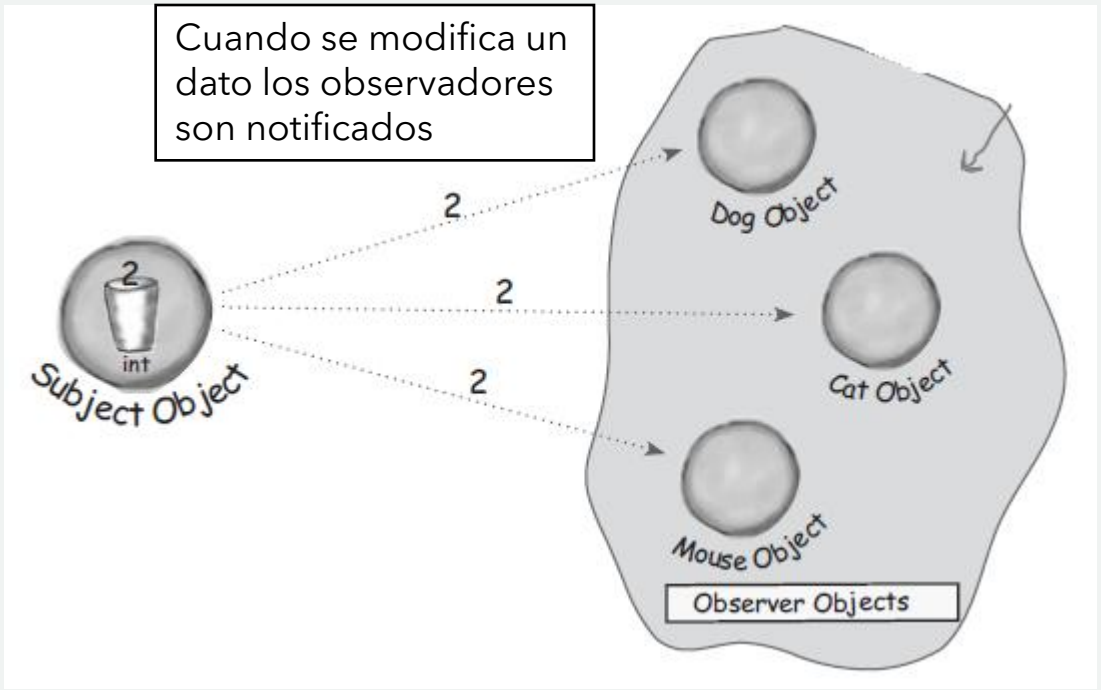


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Editorial + Suscriptores = Patrón Observador

+La editorial es el **sujeto** (subject) y los suscriptores los **observadores**

Maneja algún dato

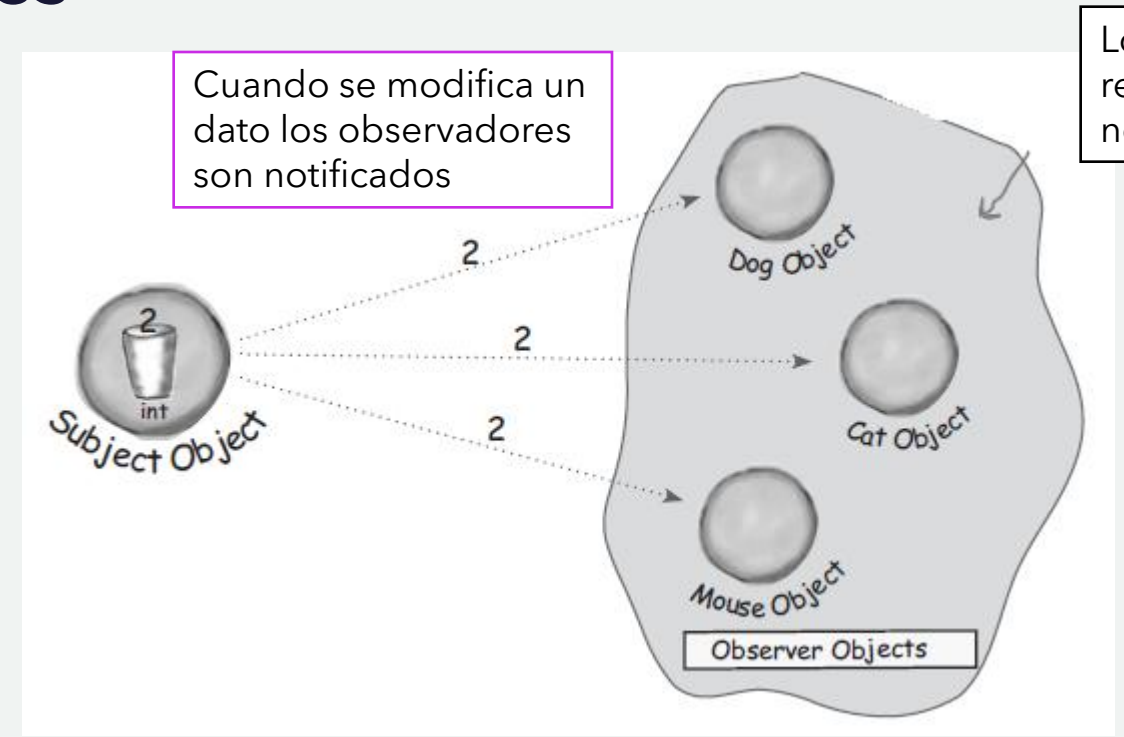


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Editorial + Suscriptores = Patrón Observador

+La editorial es el **sujeto** (subject) y los suscriptores los **observadores**

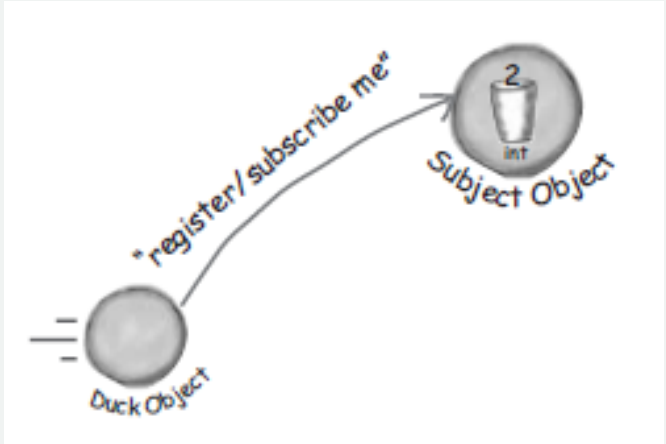
Maneja algún dato



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Un día en la vida del patrón Observador

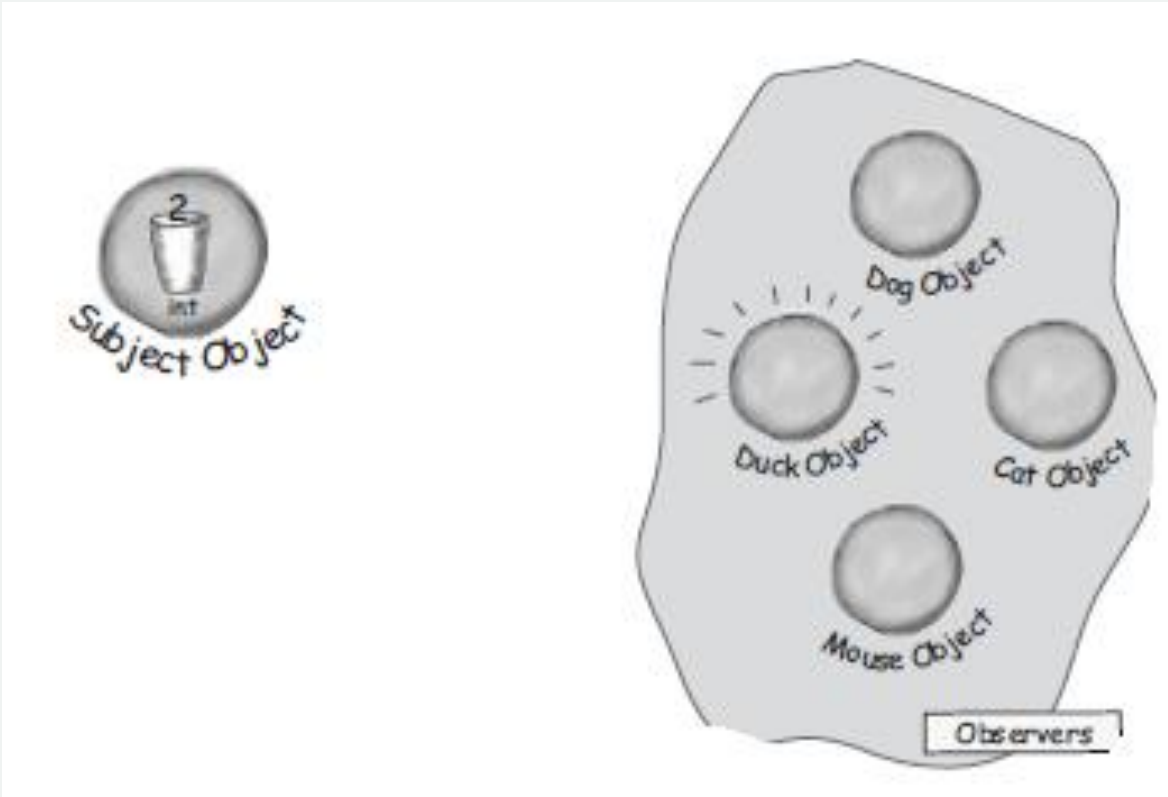
+Un día llega un objeto que quiere registrarse



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Un día en la vida del patrón Observador

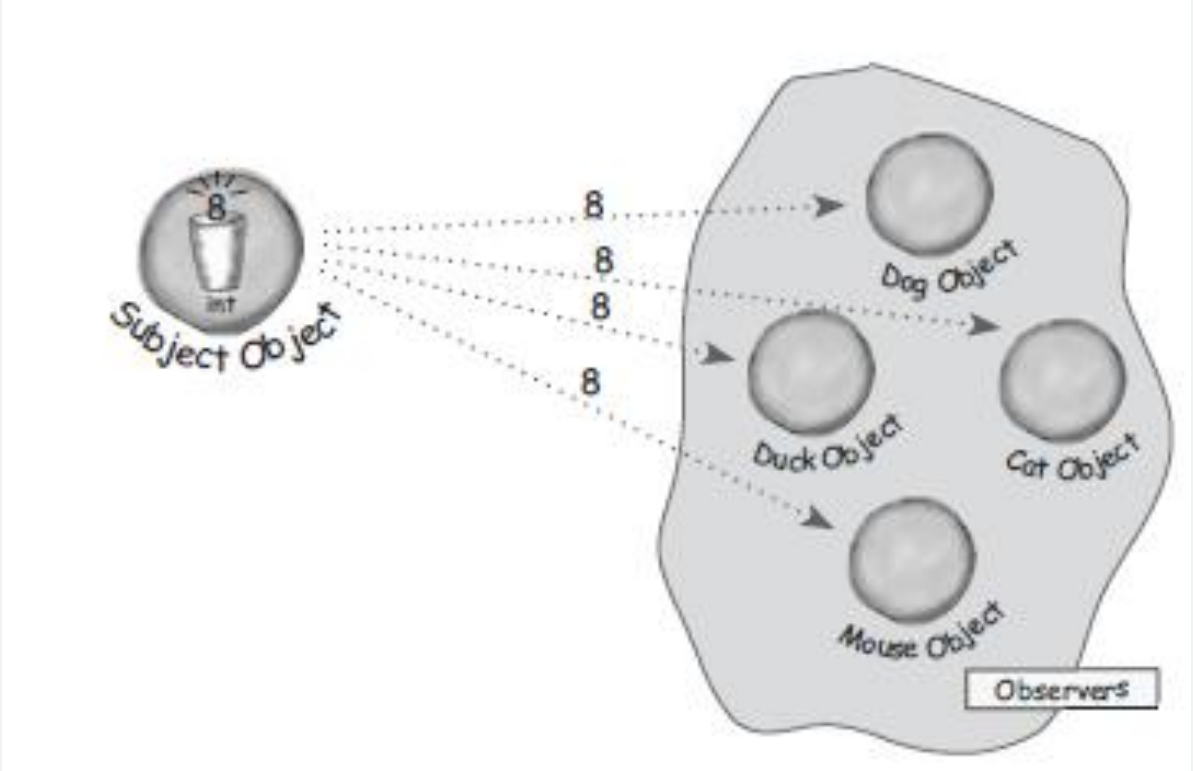
+En consecuencia el grupo de **observadores** crece



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Un día en la vida del patrón Observador

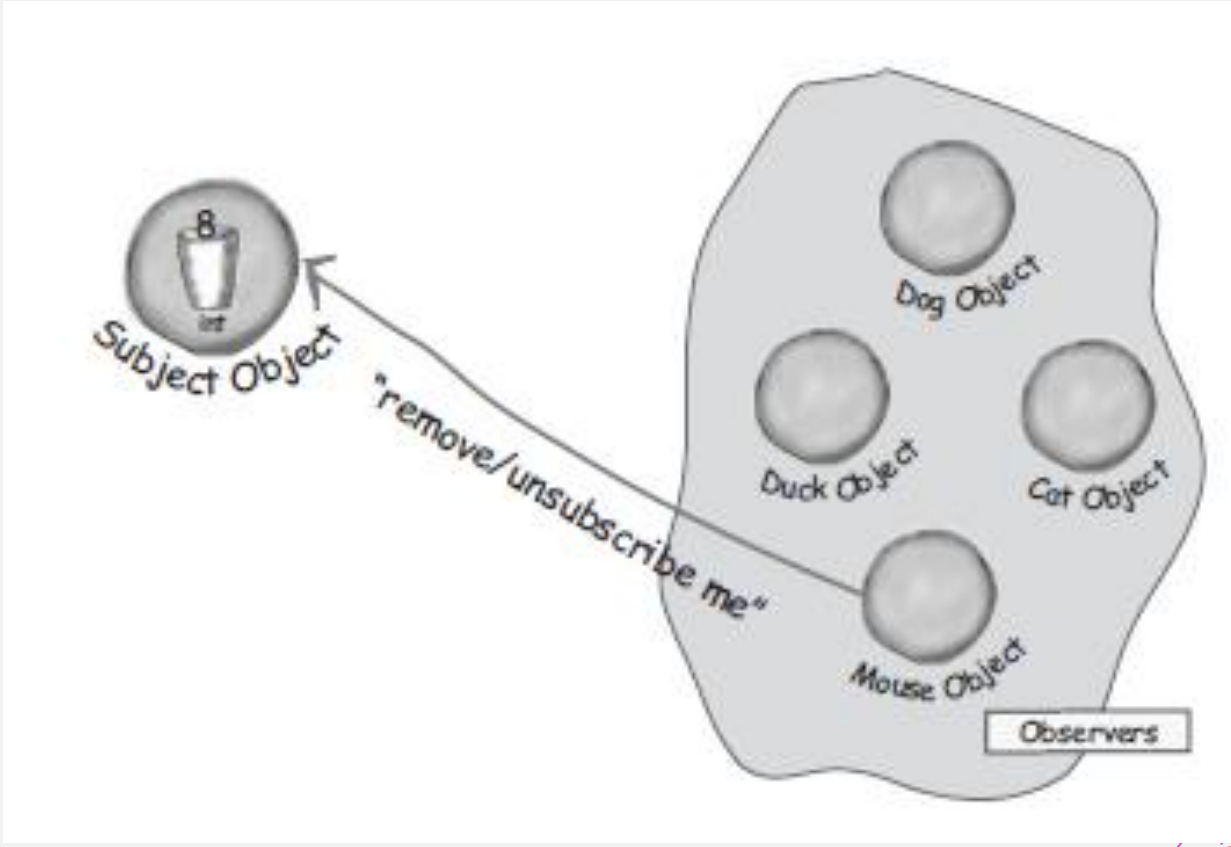
+Los cambios todos los reciben



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Un día en la vida del patrón Observador

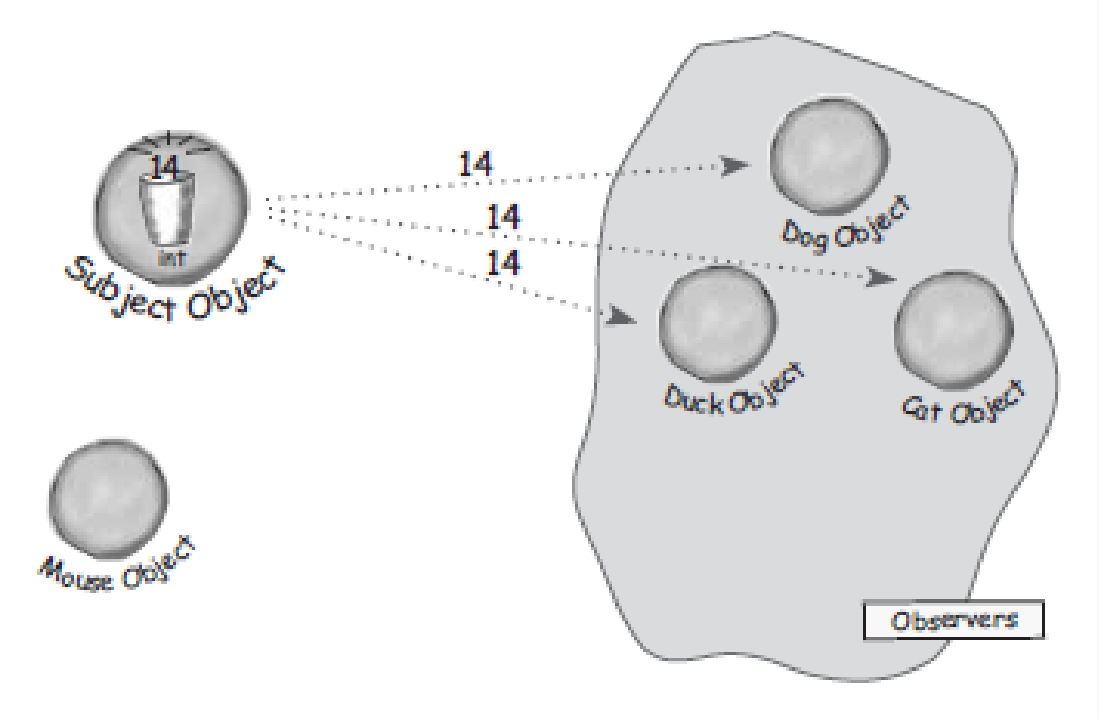
+Ahora el ratón ya no quiere ser observador



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Un día en la vida del patrón Observador

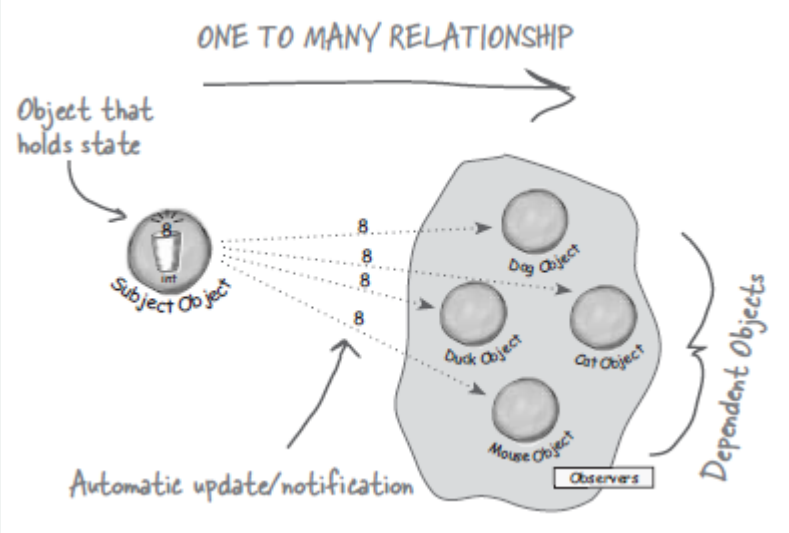
+El ratón se va y ya no recibe notificaciones



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

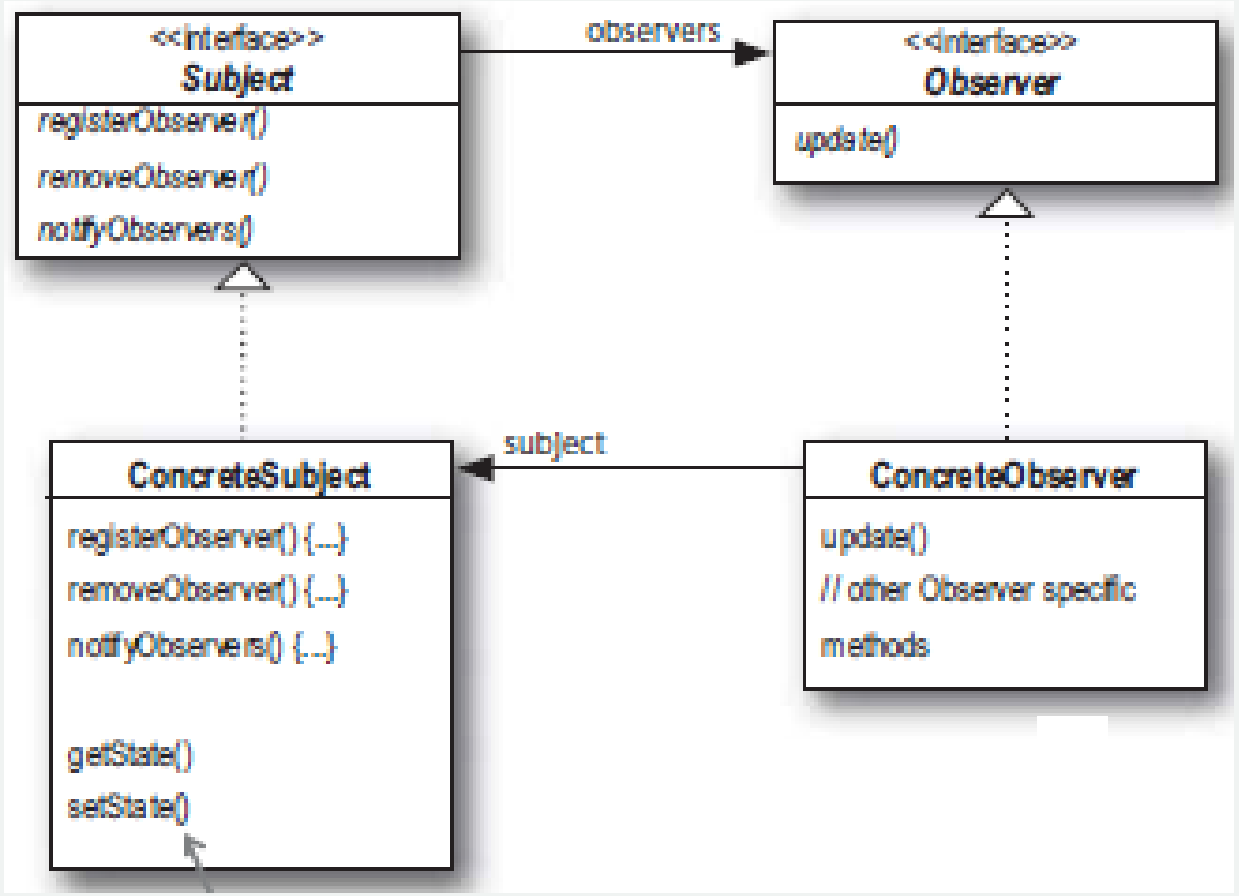
Definición de patrón Observador

+Define una dependencia de uno – a – muchos entre objetos de tal forma que, cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

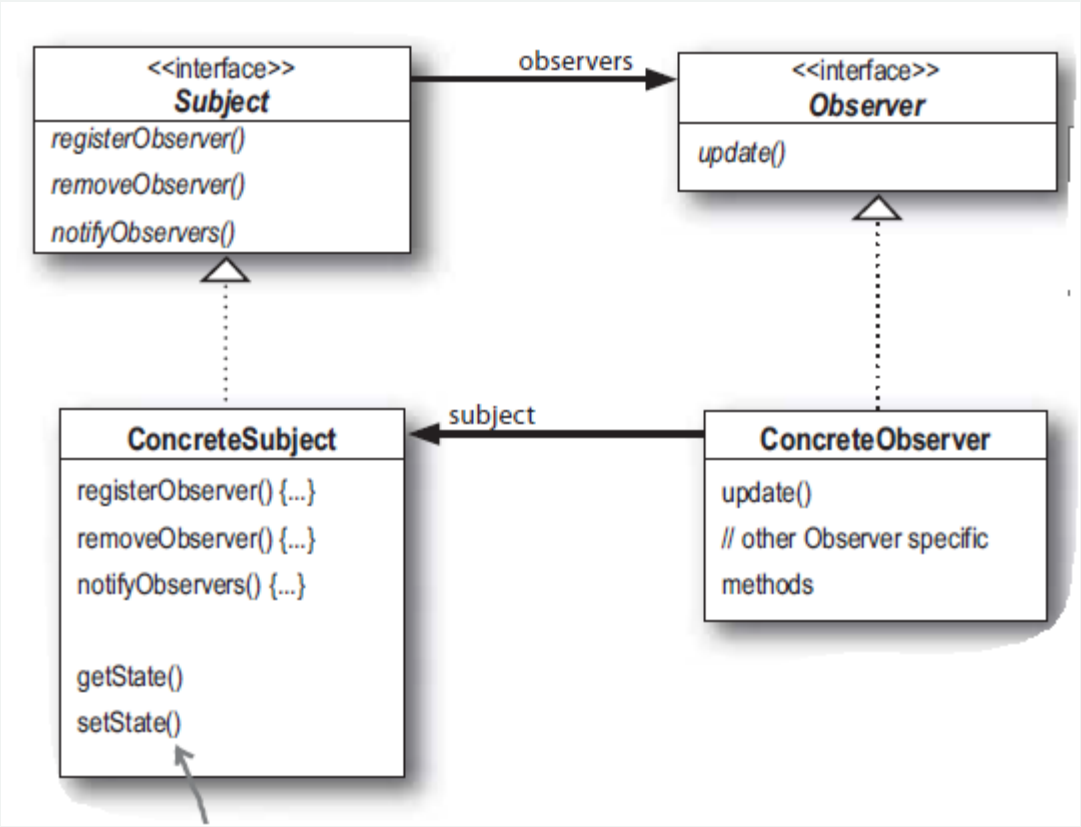
Definición de patrón Observador



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definición de patrón Observador

La interfaz del **sujeto.** Los objetos usan esta para registrarse o removerse como **observadores**

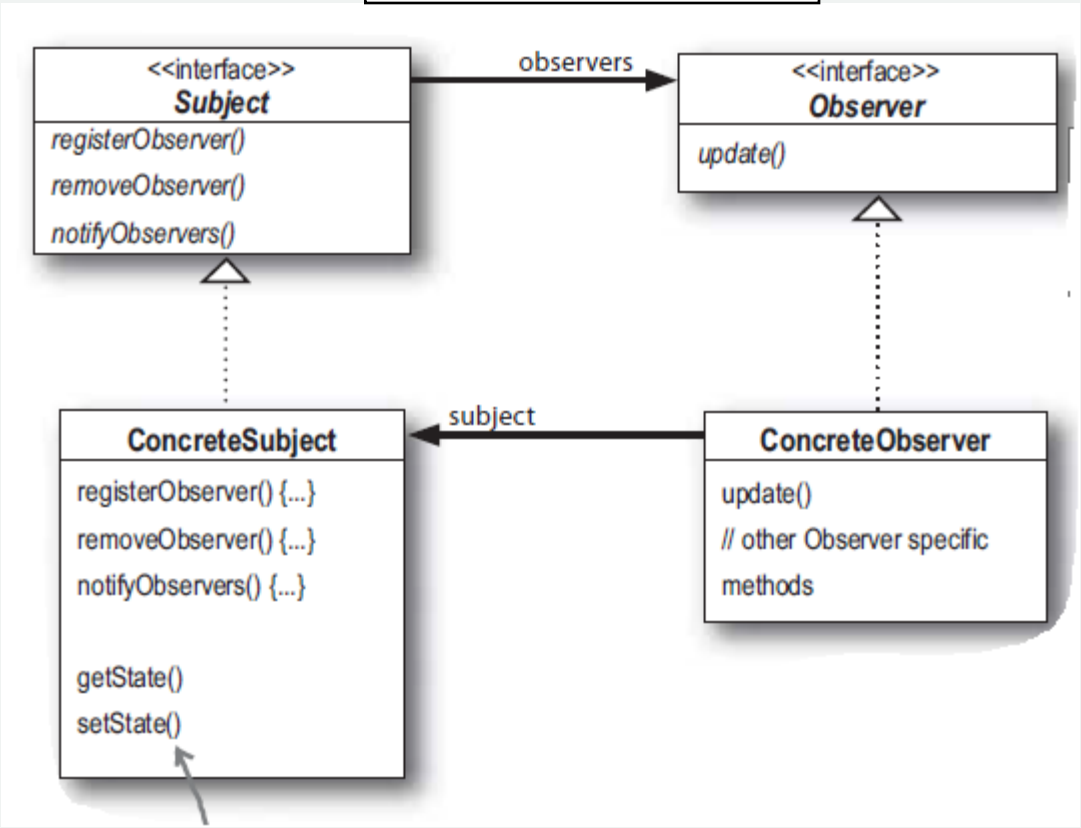


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definición de patrón Observador

La interfaz del **sujeto.** Los objetos usan esta para registrarse o removerse como **observadores**

Cada **sujeto** puede tener muchos **observadores**

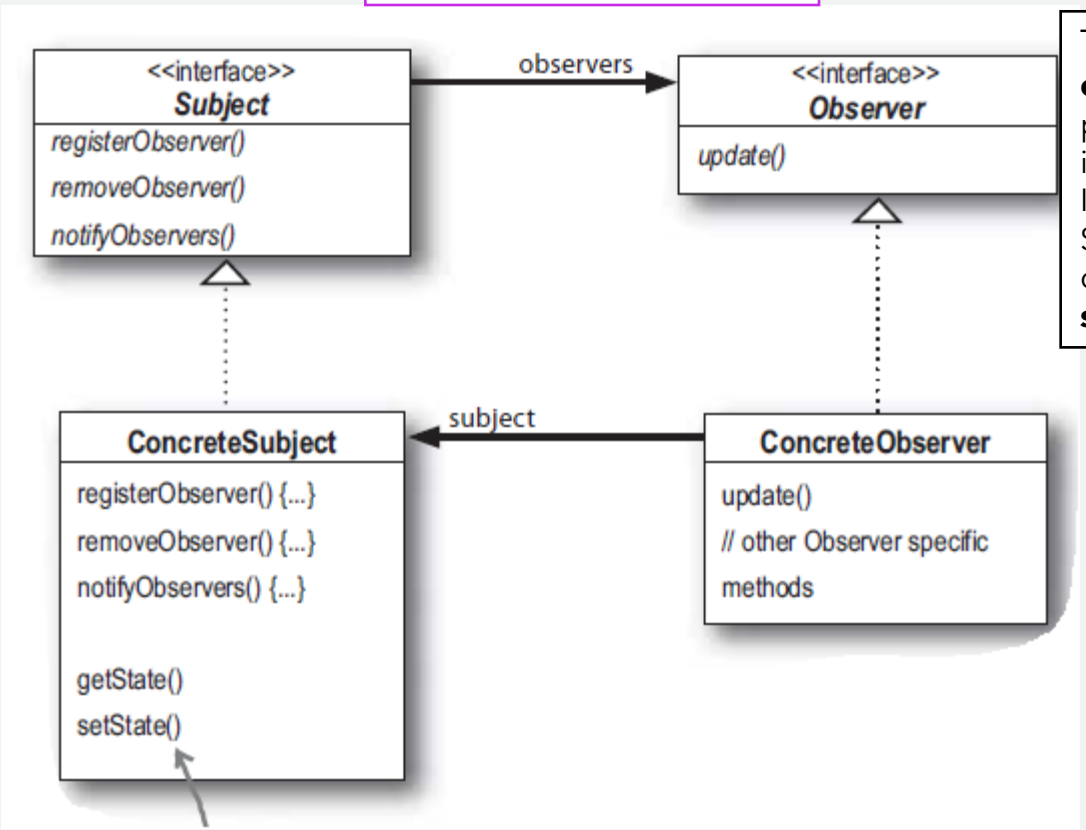


Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definición de patrón Observador

Cada **sujeto** puede tener muchos **observadores**

La interfaz del **sujeto.** Los objetos usan esta para registrarse o removerse como **observadores**



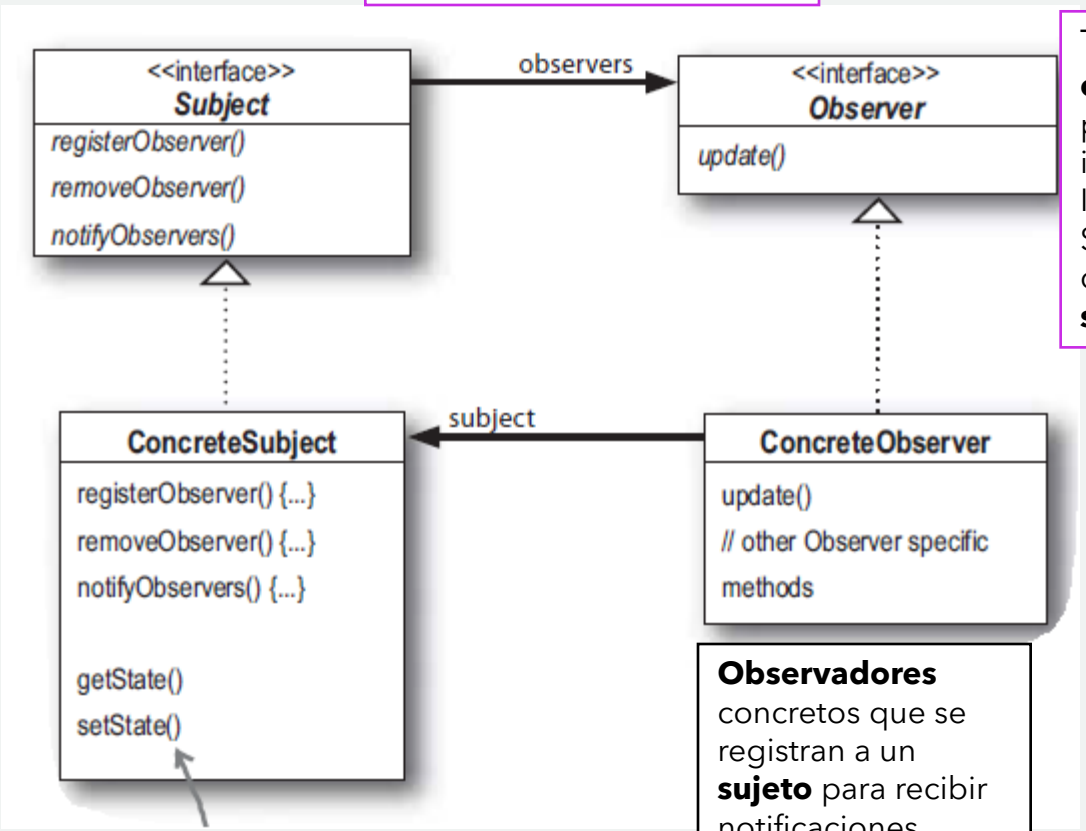
Todos los **observadores** potenciales deben implementar la Interfaz
Sólo un método que llamará el **sujeto**

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definición de patrón Observador

Cada **sujeto** puede tener muchos **observadores**

La interfaz del **sujeto.** Los objetos usan esta para registrarse o removerse como **observadores**



Todos los **observadores** potenciales deben implementar la Interfaz
Sólo un método que llamará el **sujeto**

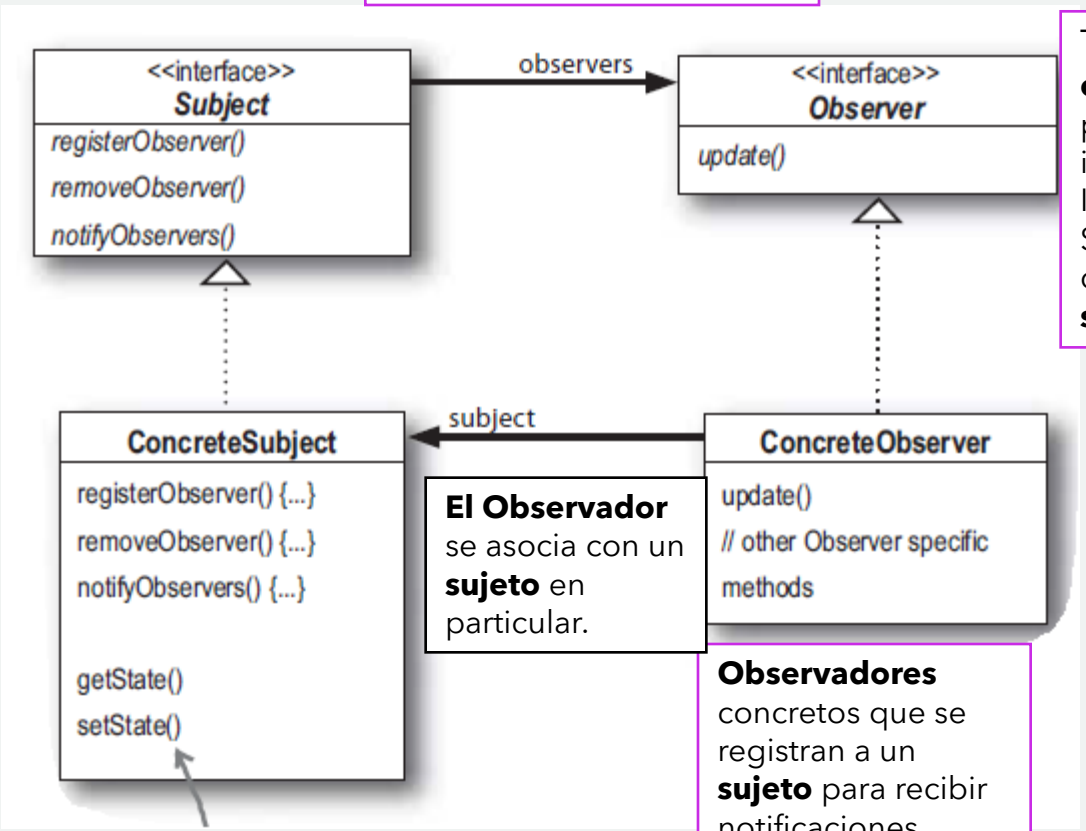
Observadores concretos que se registran a un **sujeto** para recibir notificaciones.

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definición de patrón Observador

Cada **sujeto** puede tener muchos **observadores**

La interfaz del **sujeto.** Los objetos usan esta para registrarse o removerse como **observadores**



Todos los **observadores** potenciales deben implementar la Interfaz
Sólo un método que llamará el **sujeto**

El Observador se asocia con un **sujeto** en particular.

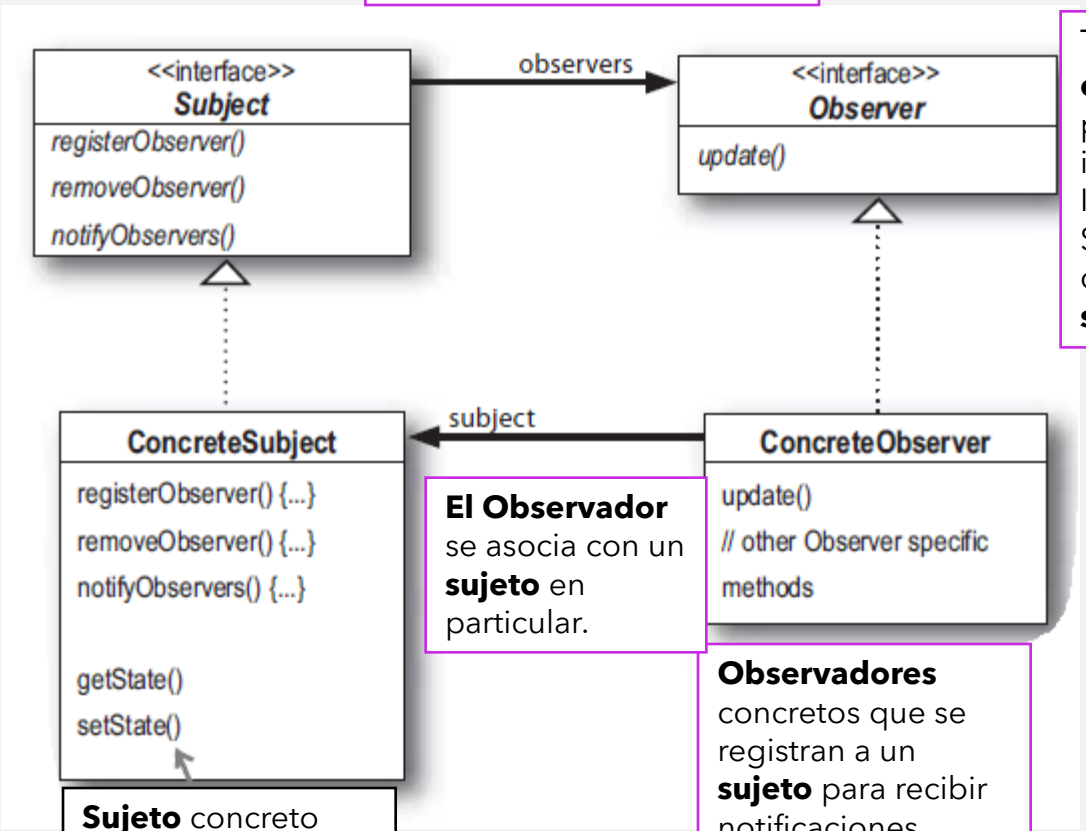
Observadores concretos que se registran a un **sujeto** para recibir notificaciones.

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definición de patrón Observador

Cada **sujeto** puede tener muchos **observadores**

La interfaz del **sujeto.** Los objetos usan esta para registrarse o removerse como **observadores**



Todos los **observadores** potenciales deben implementar la Interfaz
Sólo un método que llamará el **sujeto**

El Observador se asocia con un **sujeto** en particular.

Observadores concretos que se registran a un **sujeto** para recibir notificaciones.

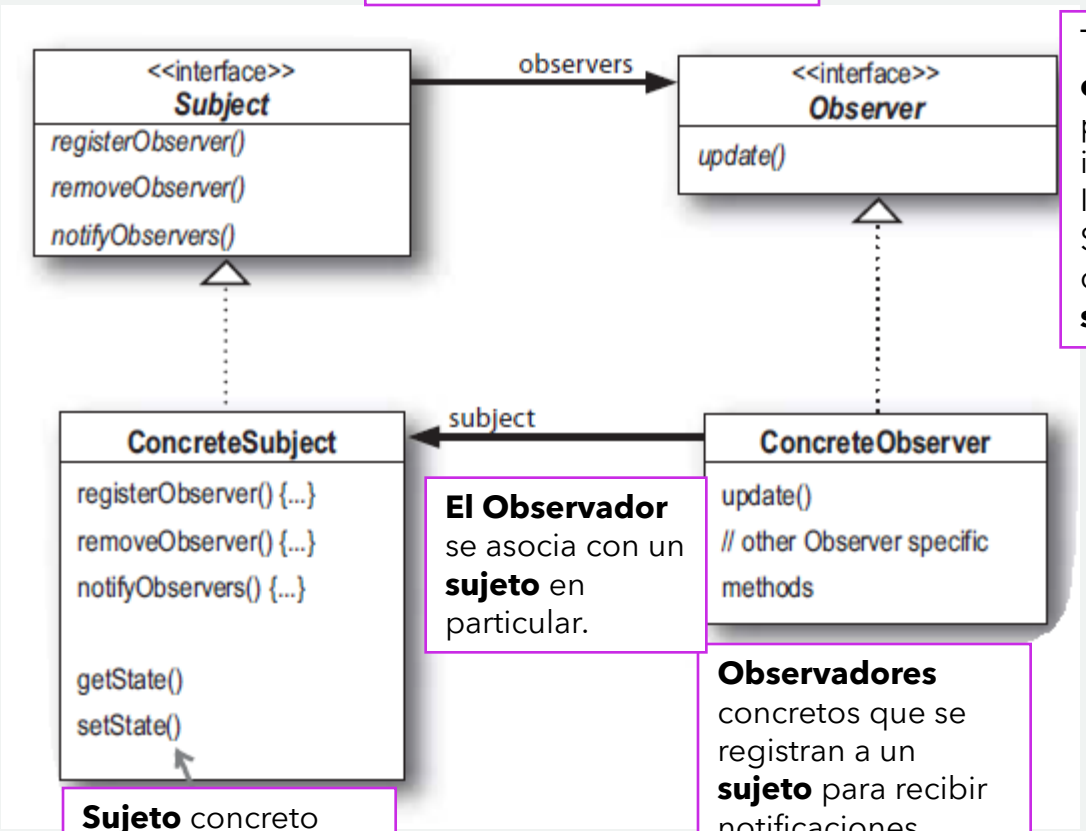
Sujeto concreto que usa métodos para modificar su estado

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Definición de patrón Observador

Cada **sujeto** puede tener muchos **observadores**

La interfaz del **sujeto.** Los objetos usan esta para registrarse o removerse como **observadores**



Todos los **observadores** potenciales deben implementar la Interfaz
Sólo un método que llamará el **sujeto**

Sujeto concreto
Que debe implementar los métodos

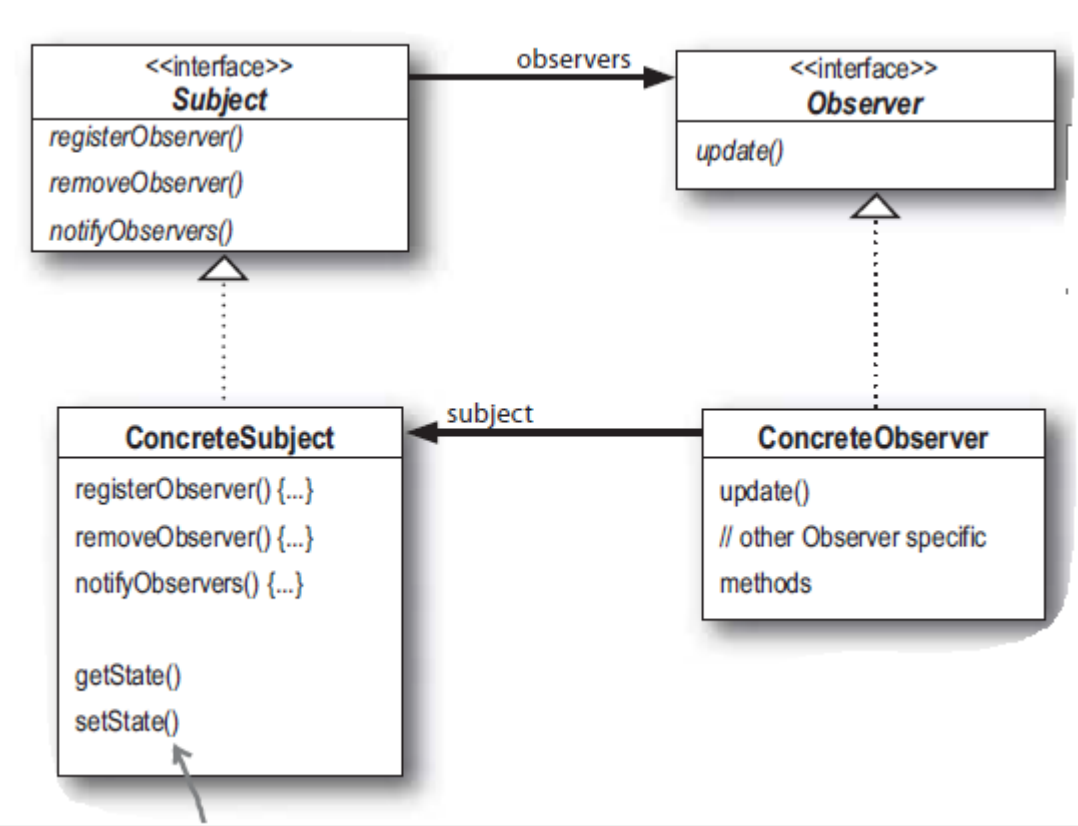
El Observador se asocia con un **sujeto** en particular.

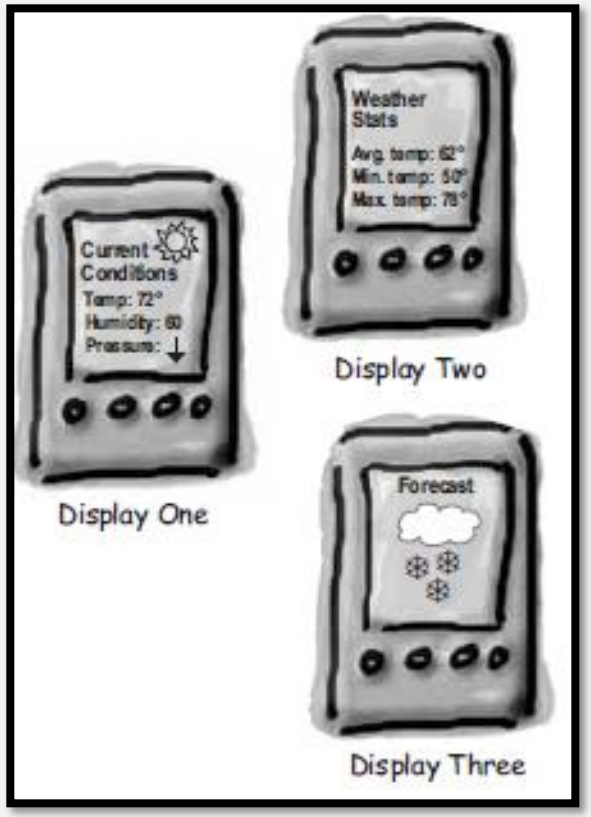
Observadores concretos que se registran a un **sujeto** para recibir notificaciones.

Sujeto concreto que usa métodos para modificar su estado

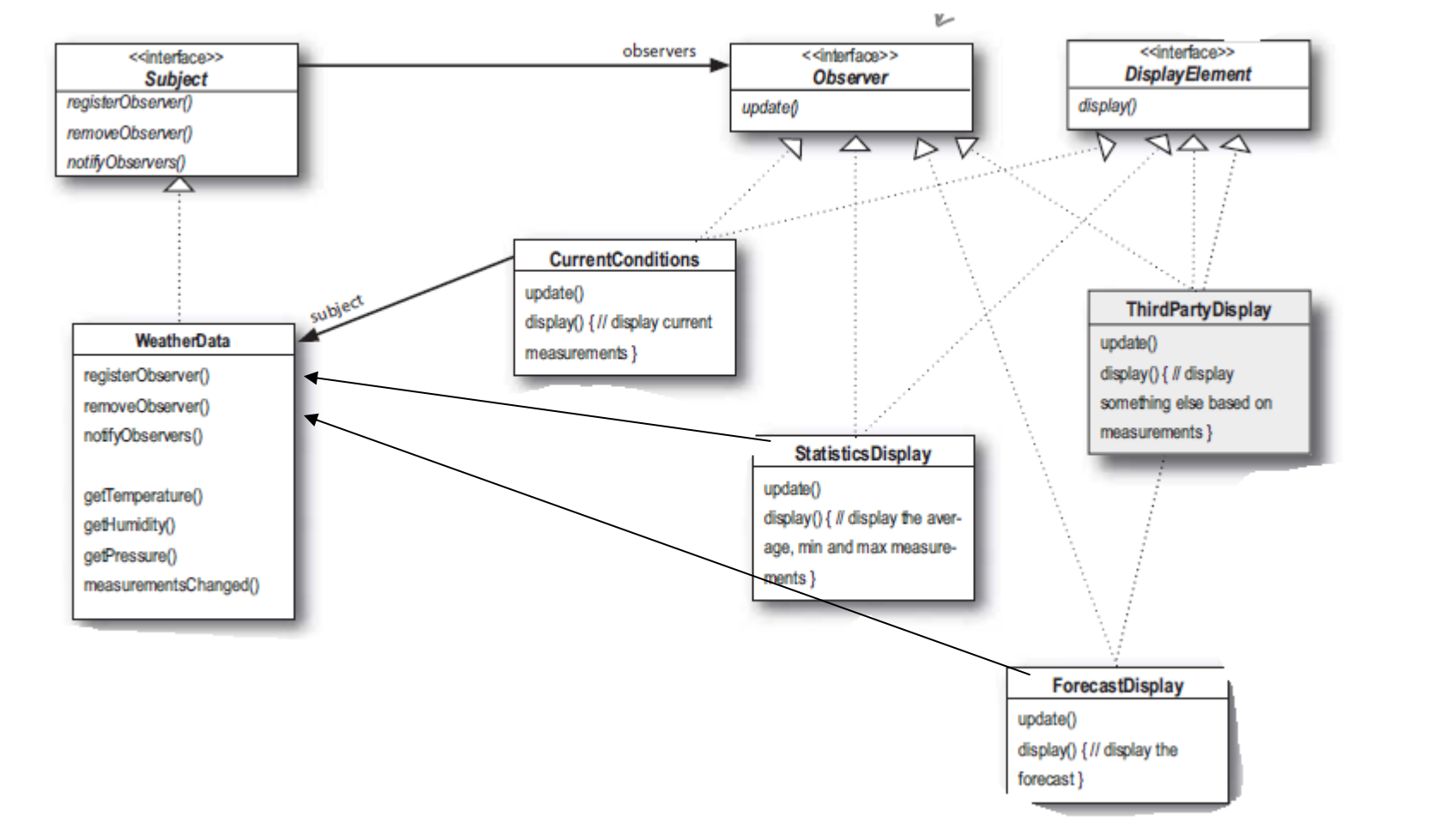
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Diseña las clases a usar para este problema

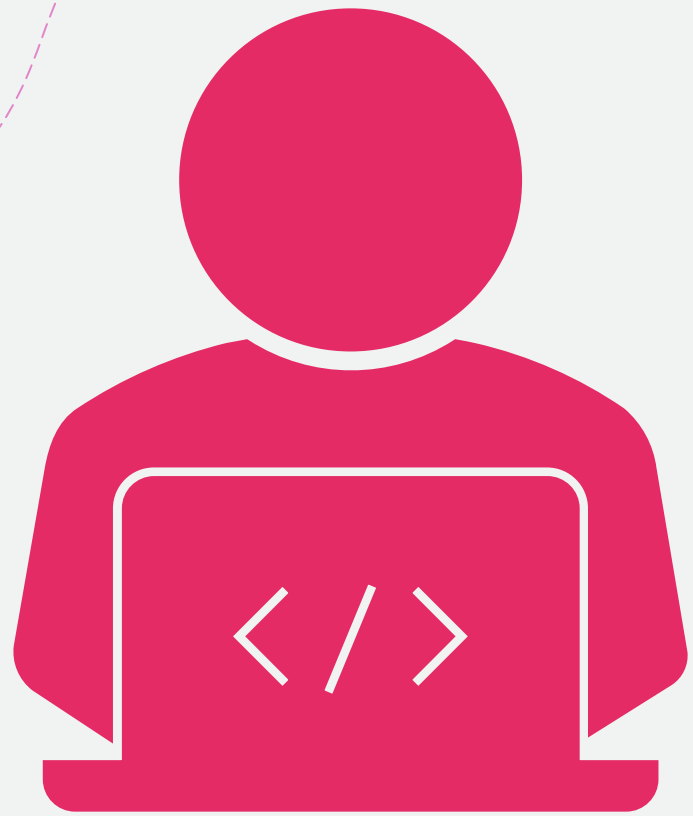




Diseña las clases a usar para este problema



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly



¿Cómo sería el código?

Codificando

```
public interface Subject {
    public void registerObserver (Observer o);
    public void remove (Observer o);
    public void notifyObservers ();
}
```

```
public interface Observer {
    public void update (float temp, float humidity,
                        float pressure);
}
```

```
public interface DisplayElement {
    public void display ();
}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Codificando

```java
public interface Subject {
    public void registerObserver (Observer o);
    public void remove (Observer o);
    public void notifyObservers ();
}
```

```java
public interface Observer {
    public void update (float temp, float humidity,
                        float pressure);
}
```

```java
public interface DisplayElement {
    public void display ();
}
```

```java
import java.util.ArrayList;

public class WeatherData implements Subject{

    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData (){...}

    public void registerObserver(Observer o) {...}

    public void remove(Observer o) {...}

    public void notifyObservers() {...}

    public void measurementsChanged (){...}

    public void setMeasurements (float temperature, float humidity, float pressure){...}

}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Codificando

```
public class WeatherData implements Subject{

    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData (){
        observers = new ArrayList();
    }

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

    public void remove(Observer o) {
        int i = observers.indexOf(o);
        if (i >=0){
            observers.remove (i);
        }
    }
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Codificando

```
    public void notifyObservers() {
        for (int i=0; i < observers.size(); i++){
            Observer observer = (Observer) observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged (){
        notifyObservers();
    }

    public void setMeasurements (float temperature, float humidity, float pressure){
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

}
```

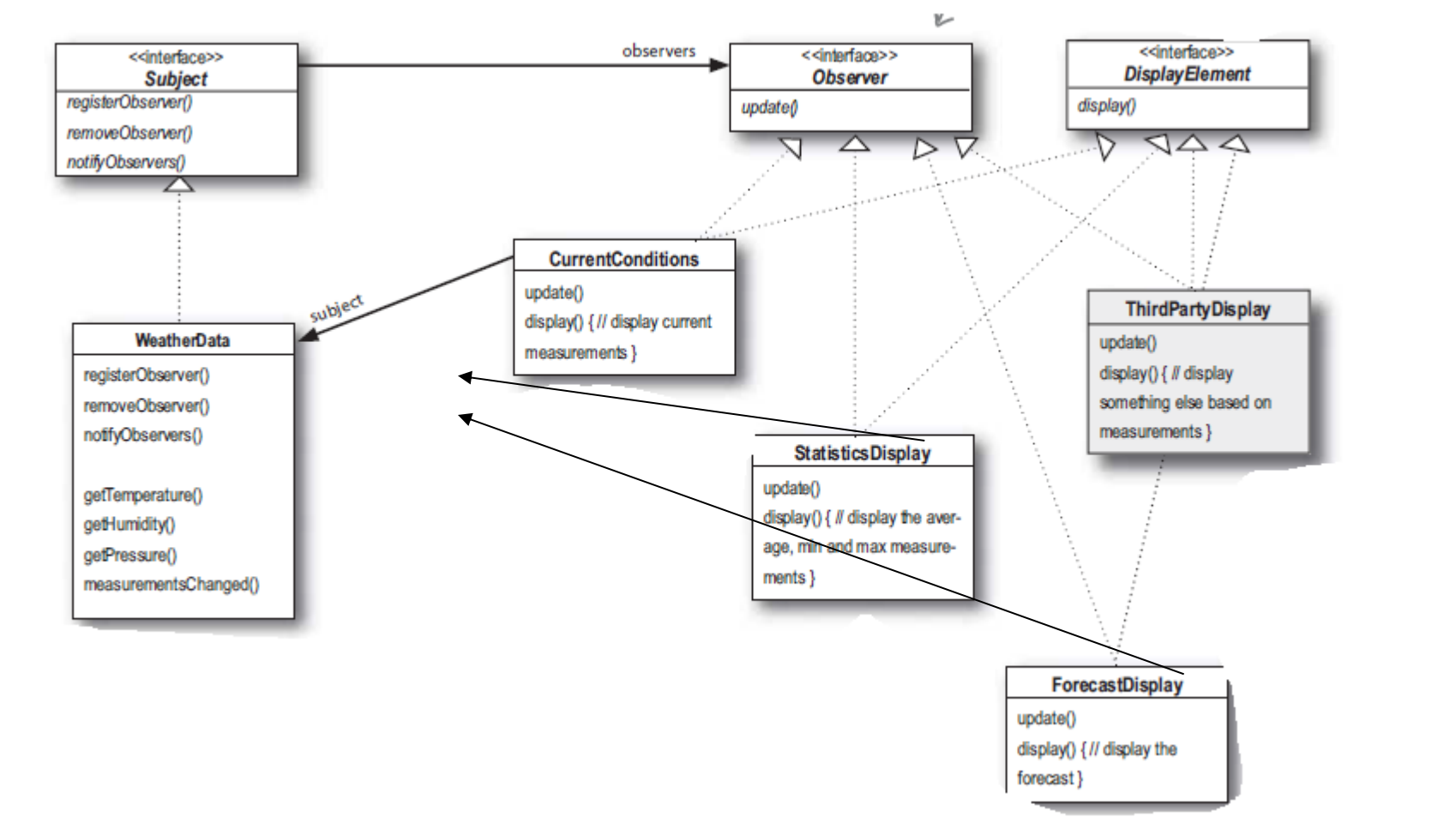
Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Codificando

```java
public class WeatherData implements Subject{

    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData (){
        observers = new ArrayList();
    }

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

    public void remove(Observer o) {
        int i = observers.indexOf(o);
        if (i >=0){
            observers.remove (i);
        }
    }

    public void notifyObservers() {
        for (int i=0; i < observers.size(); i++){
            Observer observer = (Observer) observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged (){
        notifyObservers();
    }

    public void setMeasurements (float temperature, float humidity, float pressure){
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Y la clase CurrentConditions



Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Codificando, condición actual

```
public class CurrentConditionsDisplay implements Observer, DisplayElement{

    private float temperature;
    private float humidity;
    private float pressure;
    private Subject weatherData;


    public CurrentConditionsDisplay(Subject weatherData){
        this.weatherData = weatherData;
        weatherData.registerObserver (this);
}

    public void update(float temp, float humidity, float pressure) {
        this.temperature= temperature;
        this.humidity = humidity;
        this.pressure=pressure;
        display();
    }


    public void display() {
        System.out.println ("Current conditions: "+temperature +
                            "F degrees," + humidity + "% humidity, and "+
                             pressure+ "pressure" );
    }

}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly

Codificando Forecast

+ La presión 29.92f se considera normal.
+ Si hay una variación por encima de esta medida entonces el clima cambia positivamente, si esta por debajo entonces el clima se viene lluvioso y frío.

Codificando Forecast

```java
import java.util.*;

public class ForecastDisplay implements Observer, DisplayElement {
        private float currentPressure = 29.92f;
        private float lastPressure;
        private WeatherData weatherData;

        public ForecastDisplay(WeatherData weatherData) {
                this.weatherData = weatherData;
                weatherData.registerObserver(this);
        }

        public void update(float temp, float humidity, float pressure) {
                lastPressure = currentPressure;
                currentPressure = pressure;

                display();
        }

        public void display() {
                System.out.print("Forecast: ");
                if (currentPressure > lastPressure) {
                        System.out.println("Improving weather on the way!");
                } else if (currentPressure == lastPressure) {
                        System.out.println("More of the same");
                } else if (currentPressure < lastPressure) {
                        System.out.println("Watch out for cooler, rainy weather");
                }
        }
}
```

Codificando Statistics

+Nos interesa la temperatura promedio, máxima y mínima

Codificando Statistics

```
public class StatisticsDisplay implements Observer, DisplayElement {
        private float maxTemp = 0.0f;
        private float minTemp = 200;
        private float tempSum= 0.0f;
        private int numReadings;
        private WeatherData weatherData;

        public StatisticsDisplay(WeatherData weatherData) {
                this.weatherData = weatherData;
                weatherData.registerObserver(this);
        }

        public void update(float temp, float humidity, float pressure) {
                tempSum += temp;
                numReadings++;

                if (temp > maxTemp) {
                        maxTemp = temp;
                }

                if (temp < minTemp) {
                        minTemp = temp;
                }

                display();
        }

        public void display() {
                System.out.println("Avg/Max/Min temperature = " +
                        (tempSum / numReadings) + "/" + maxTemp +
                        "/" + minTemp);
        }
}
```

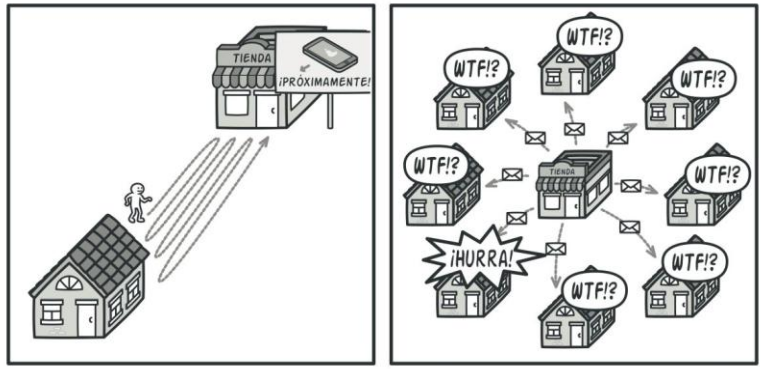
Probemos el programa

```java
import java.util.*;

public class WeatherStation {

        public static void main(String[] args) {
                WeatherData weatherData = new WeatherData();

                CurrentConditionsDisplay currentDisplay =
                        new CurrentConditionsDisplay(weatherData);
                StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
                ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

                weatherData.setMeasurements(80, 65, 30.4f);
                weatherData.setMeasurements(82, 70, 29.2f);
                weatherData.setMeasurements(78, 90, 29.2f);
        }
}
```

tput - Observer Pattern (run)

```
run:
Current conditions: 0.0F degrees,65.0% humidity, and 30.4pressure
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 0.0F degrees,70.0% humidity, and 29.2pressure
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 0.0F degrees,90.0% humidity, and 29.2pressure
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
```

Modela la tienda



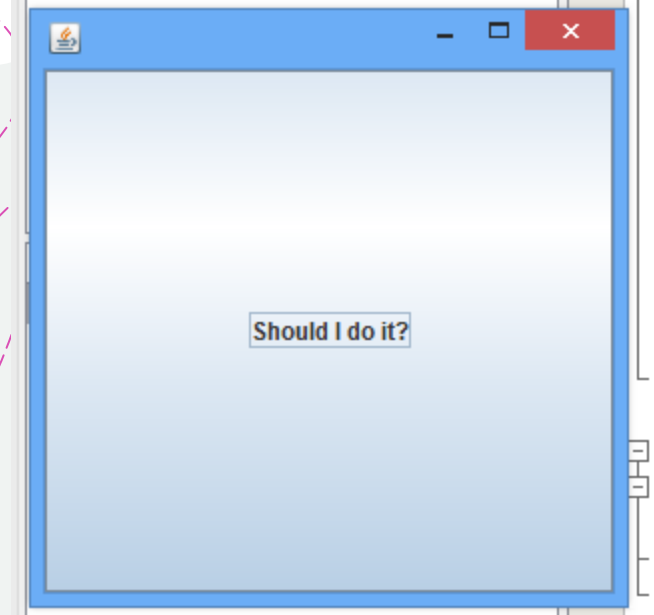
+ Imagina que tienes dos tipos de objetos: un objeto Cliente y un objeto Tienda. El cliente está muy interesado en una marca particular de producto (digamos, un nuevo modelo de iPhone) que estará disponible en la tienda muy pronto.
+ El cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero, mientras el producto está en camino, la mayoría de estos viajes serán en vano.
+ or otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados en los nuevos productos.
+ Parece que nos encontramos ante un conflicto. O el cliente pierde tiempo comprobando la disponibilidad del producto, o bien la tienda desperdicia recursos notificando a los clientes equivocados.

¿Dónde vemos más observadores?

- JButton sujeto
- Lleno de observadores
  - Listeners

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class SwingObserverExample {
        JFrame frame;

        public static void main(String[] args) {
                SwingObserverExample example = new SwingObserverExample();
                example.go();
        }

        public void go() {
                frame = new JFrame();

                JButton button = new JButton("Should I do it?");
                button.addActionListener(new AngelListener());
                button.addActionListener(new DevilListener());
                frame.getContentPane().add(BorderLayout.CENTER, button);

                // Set frame properties
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.getContentPane().add(BorderLayout.CENTER, button);
                frame.setSize(300,300);
                frame.setVisible(true);
        }

        class AngelListener implements ActionListener {
                public void actionPerformed(ActionEvent event) {
                        System.out.println("Don't do it, you might regret it!");
                }
        }

        class DevilListener implements ActionListener {
                public void actionPerformed(ActionEvent event) {
                        System.out.println("Come on, do it!");
                }
        }
}
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly




```
16                  JButton button = new JButton("Should I do it?");
                    button.addActionListener(new AngelListener());
                    button.addActionListener(new DevilListener());
                    frame.getContentPane().add(BorderLayout.CENTER, button);

                    // Set frame properties
                    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    frame.getContentPane().add(BorderLayout.CENTER, button);
                    frame.setSize(300,300);
                    frame.setVisible(true);
            }

            class AngelListener implements ActionListener {
                    public void actionPerformed(ActionEvent event) {
                            System.out.println("Don't do it, you might regret it!");
                    }
            }

34
35          class DevilListener implements ActionListener {
                    public void actionPerformed(ActionEvent event) {
37                          System.out.println("Come on, do it!");
38                  }
39          }
40  }
41
```

Output - Observer Pattern (run)

```
run:
Come on, do it!
Don't do it, you might regret it!
Come on, do it!
Don't do it, you might regret it!
```

Fuente: Freeman, E., Freeman E., Head First Design Patterns, O'Reilly