

TC3003: Diseño y Arquitectura de Software

Dr. Juan Manuel González Calleros

Email: jmgonzale@itesm.mx

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

Facebook: [Juan Glez Calleros](https://facebook.com/Juan.Glez.Calleros)

Reuniones pedir cita



<http://www.pue.itesm.mx/precoas/>

Tarea

- Resolución de la investigación de PBL de su proyecto, faces de comprensión y documentación de la problemática con datos de INEGI
- Casos de uso de la actividad docente que debes desarrollar (e-learning, m-learning, b-learning)
- Identificación del modelo educativo del TEC y modelado de clases del mismo
- Presentar resultados el lunes 23

PROYECTO

Proyecto

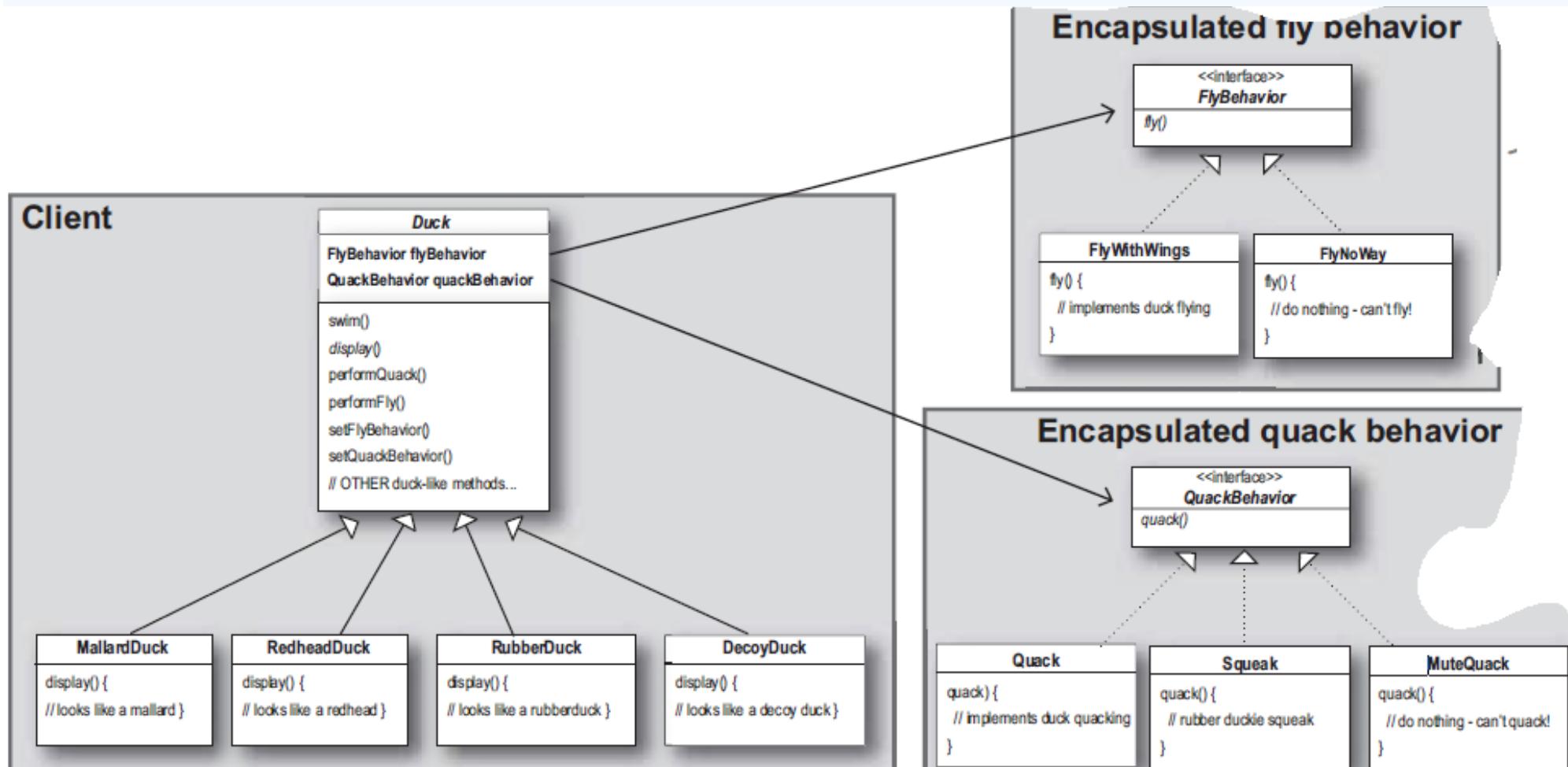
- Vamos a escuchar su investigación

@Juan__Gonzalez

Patrones Comportamentales

PATRÓN ESTRATEGIA

La encapsulación



Clase Duck

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

Interfaz de Vuelo

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

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

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

Interfaz Graznido

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

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

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

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

Prueba

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

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

Definamos el comportamiento de forma automática

```
public abstract class Duck {  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
  
    //Único método a sobre escribir en todas las subclases  
    public Duck () {...}  
  
    public abstract void display ();  
  
    //Dependiendo del constructor de las subclases será el tipo de vuelo  
    public void performFly ()  
    {...}  
  
    //Dependiendo del constructor de las subclases será el tipo de graznido  
    public void performQuack () {...}  
  
    //Este método siempre es el mismo para todos  
    public void swim () {...}  
    public void showDuck () {...}  
  
    public void setFlyBehavior (FlyBehavior fb) {  
  
        flyBehavior = fb;  
    }  
  
    public void setQuackBehavior (QuackBehavior qb) {  
  
        quackBehavior = qb;  
    }  
}
```

Duck
FlyBehavior flyBehavior;
QuackBehavior quackBehavior;
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...

Vamos a crear un pato que no vuela

- Llámenle ModelDuck

Vamos a crear un pato que no vuela

- Llámenle ModelDuck

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

Definamos ahora un comportamiento de vuelo con cohete

- Llámenle FlyRocketPowered

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

Definamos ahora un comportamiento de vuelo con cohete

- Llámenle FlyRocketPowered

Definamos ahora un comportamiento de vuelo con cohete

- Llámenle FlyRocketPowered

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

Modifiquemos el main

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

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

Felicidades

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

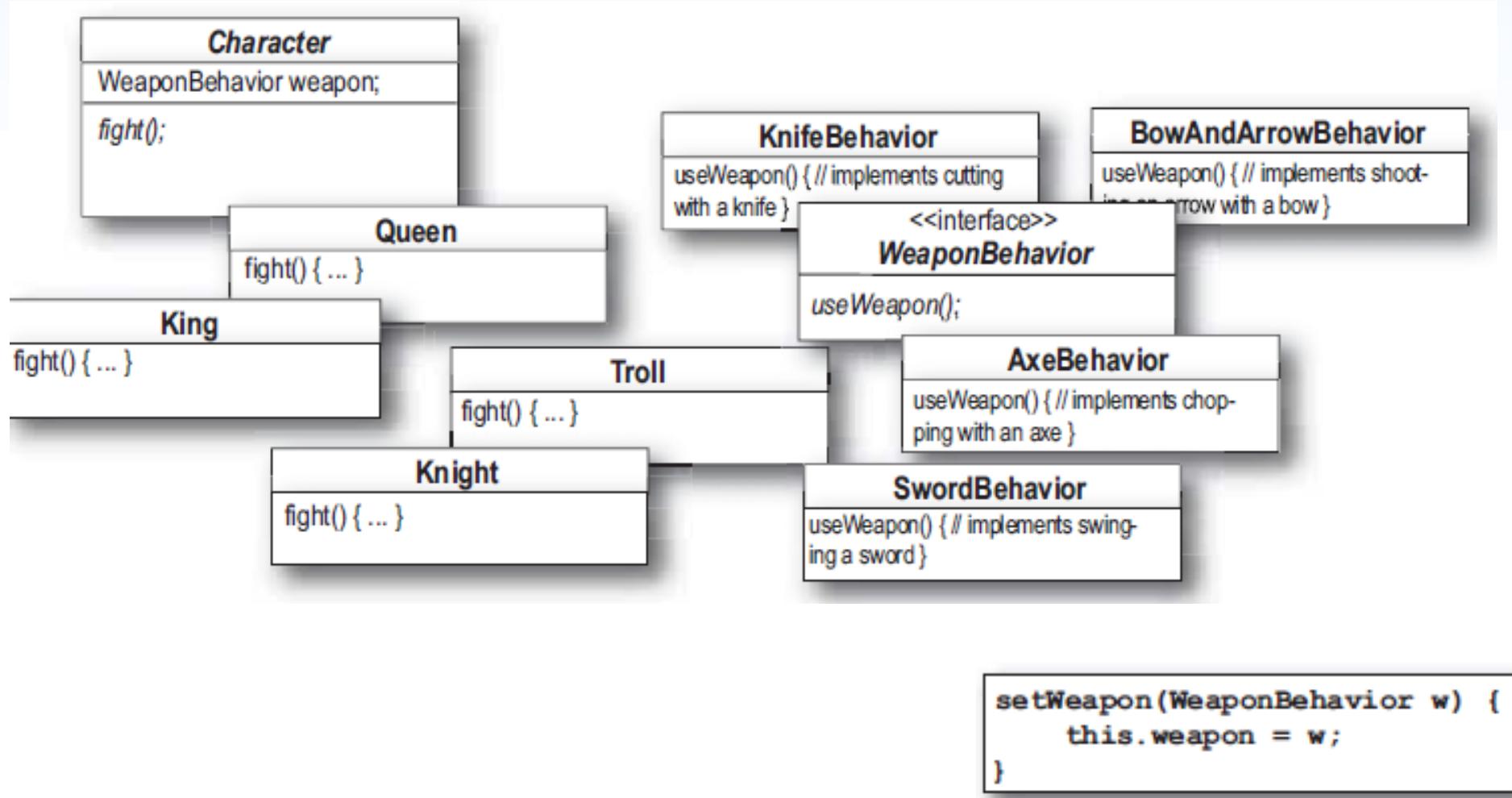
**Y ADEMÁS DE LOS PATOS
QUIÉN USA LA ESTRATEGIA**

Ejercicio – Organiza el desorden

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

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

Ejercicio – Organiza el desorden



Ahora trata de definir comportamientos de movimiento

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

Tarea Hacer el programa

@Juan__Gonzalez

¿Cómo se usan los patrones de Diseño?

- Los patrones no son una librería, son una forma de resolver problemas, estructurar, objetos y clases.
- Los Frameworks y Librerias no son patrones de diseño, son soluciones particulares que usan, a veces, patrones

¿Cómo se usan los patrones de Diseño?

- Usando conceptos básicos de POO
 - Abstracción
 - Encapsulación
 - Polimorfismo
 - Herencia

¿Cómo se usan los patrones de Diseño?

- Principios OO
 - Encapsula lo que varia
 - Favorece la composición sobre la herencia.
 - Programa las interfaces, no la implementación

¿Cómo se usan los patrones de Diseño?

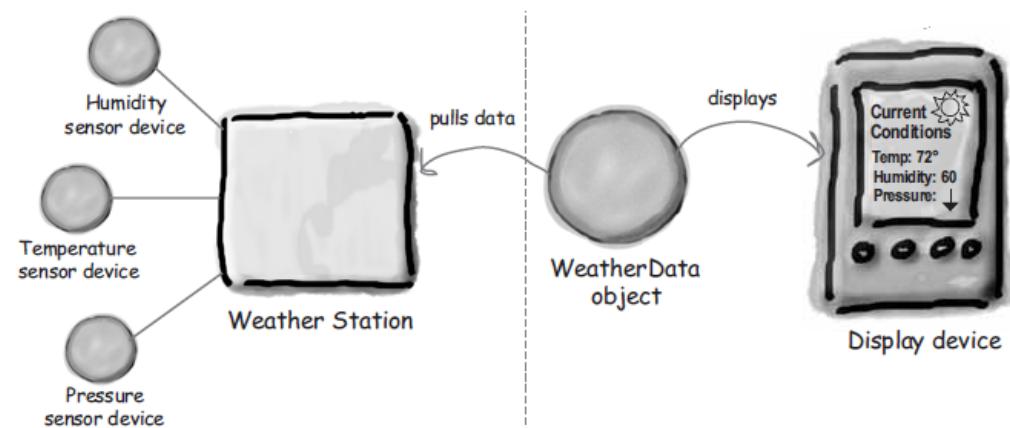
- Patrones OO
 - **Estrategia** – define una familia de algoritmos, encapsula cada uno, los hace intercambiables. La **Estrategia** hace que el algoritmo varia independientemente de los clientes que lo usan.

Patrón comportamental

PATRÓN OBSERVADOR

Patrón Observador

- Necesitamos hacer una aplicación para monitorear el clima.
 - Condiciones actuales
 - Estadísticas de clima
 - Previsión



La clase Clima

- Recuerden el cómo se manipulan los sensores no es nuestro problema

```
WeatherData
-----
getTemperature()
getHumidity()
getPressure()
measurementsChanged()

// other methods
```

La clase Clima

- Recuerden el cómo se manipulan los sensores no es nuestro problema

```
WeatherData
-----
getTemperature()
getHumidity()
getPressure()
measurementsChanged()

// other methods
```

Este se modifica cada que hay un cambio en una variable

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

La clase Clima

- Recuerden el cómo se manipulan los sensores no es nuestro problema

```
WeatherData
-----
getTemperature()
getHumidity()
getPressure()
measurementsChanged()

// other methods
```

Actualización impacta

- Condiciones actuales
- Estadísticas de clima
- Previsión

Qué sabemos al momento

- Métodos para obtener el valor de tres variables

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

Qué sabemos al momento

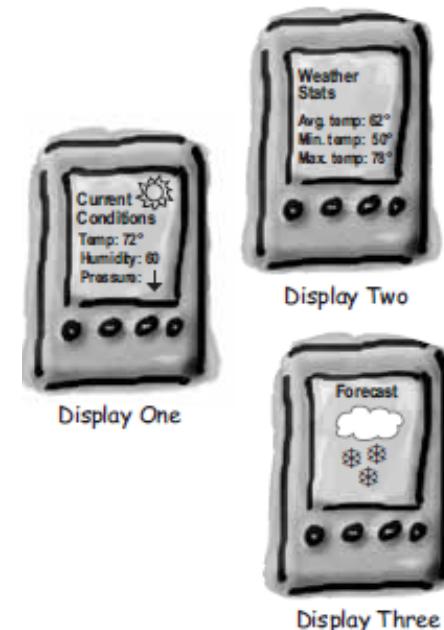
- Método que se llama cada que hay un cambio en alguna variable, solo sabemos que se manda a llamar

`measurementsChanged()`

Qué sabemos al momento

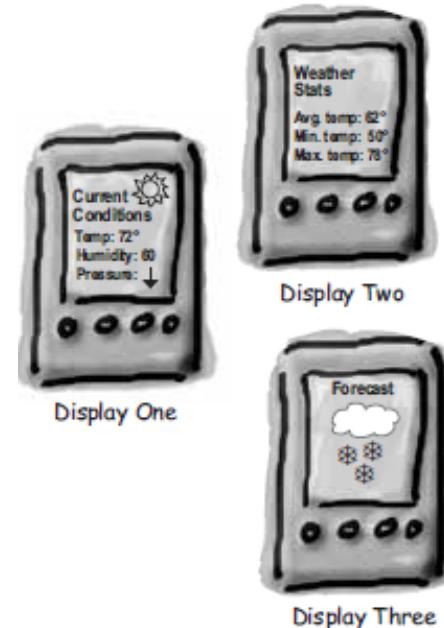
- Necesitamos implementar tres ventanas cada que se invoca el método

`measurementsChanged()`



Qué sabemos al momento

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



Así es el código actual

```
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  
}
```

¿Programa la interfaz o la implementación?

```
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  
}
```

¿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  
}
```

¿Tenemos forma de agregar o quitar elementos de la pantalla?

```
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  
}
```

¿Esta encapsulada la parte que cambia?

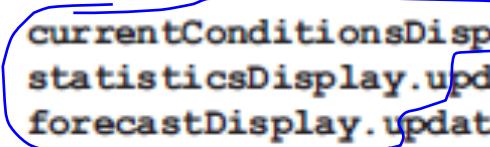
```
public class WeatherData {  
  
    // instance variable declarations  
  
    public void measurementsChanged() {  
  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);  
    }  
  
    // other WeatherData methods here  
}
```

¿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  
}
```

¿Qué falla?

- Quitar o poner elementos en las pantallas nos obliga a programar aquí

```
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  
}
```

No todo es malo

- Existe una interfaz que sabe actualizar y usa los mismos parámetros

```
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  
}
```

Necesitamos encapsular

- Existe una interfaz que sabe actualizar y usa los mismos parámetros

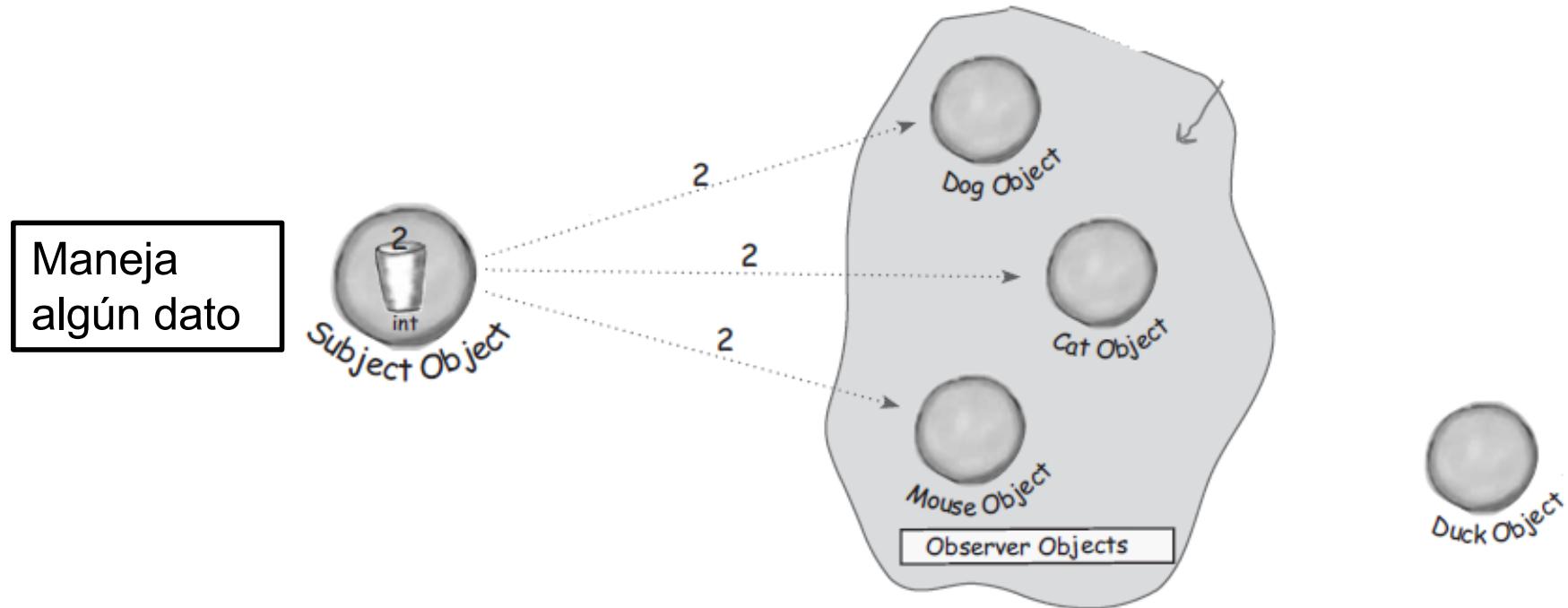
```
public class WeatherData {  
  
    // instance variable declarations  
  
    public void measurementsChanged() {  
  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);  
    }  
  
    // other WeatherData methods here  
}
```

Patrón Observador

- Usando de referente la suscripción 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 suscripción
 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.

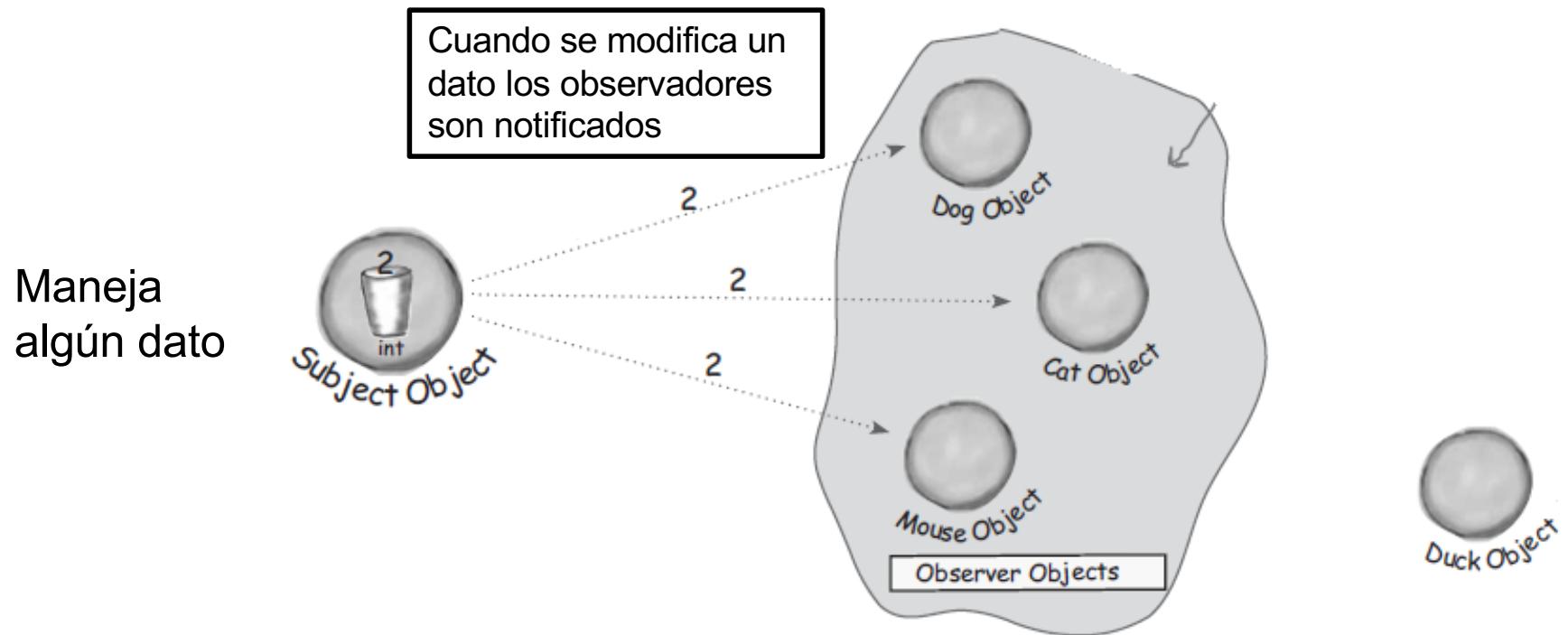
Editorial + Suscriptores = Patrón Observador

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



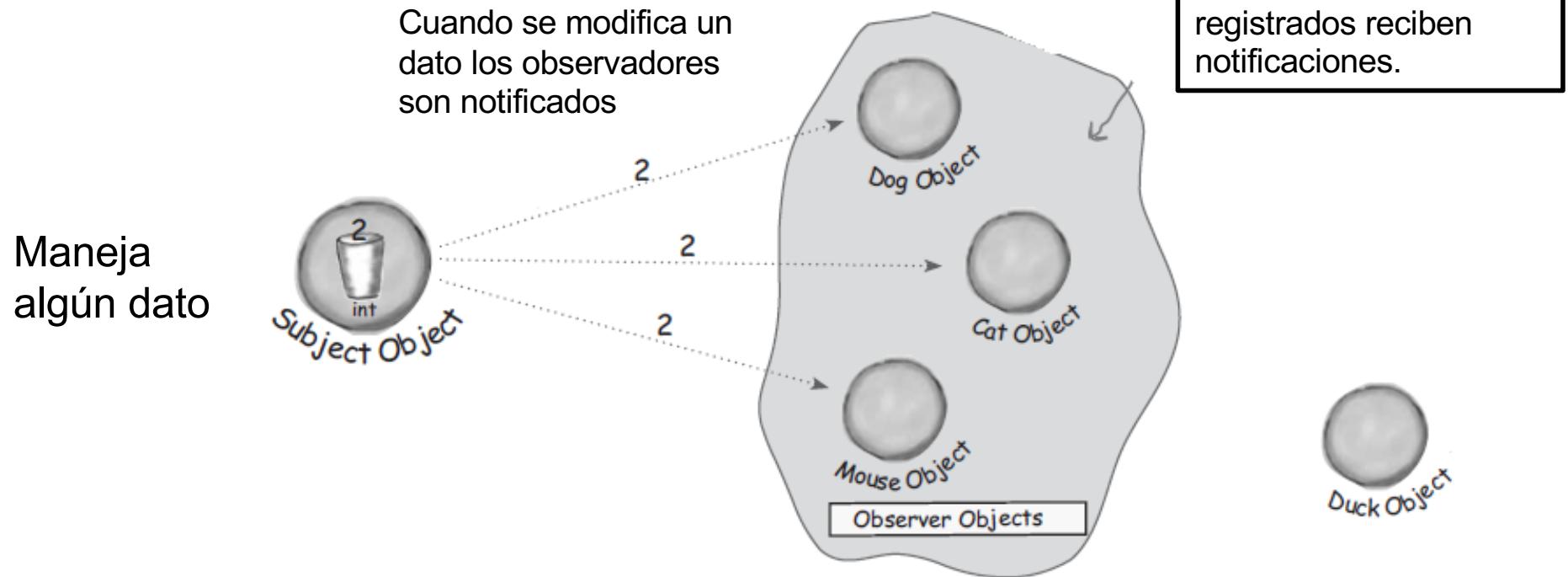
Editorial + Suscriptores = Patrón Observador

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



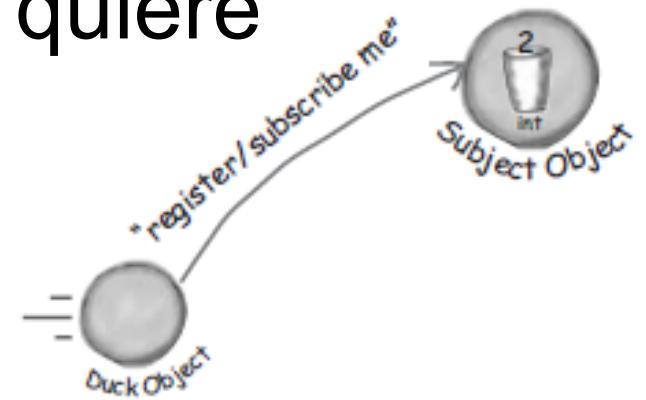
Editorial + Suscriptores = Patrón Observador

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



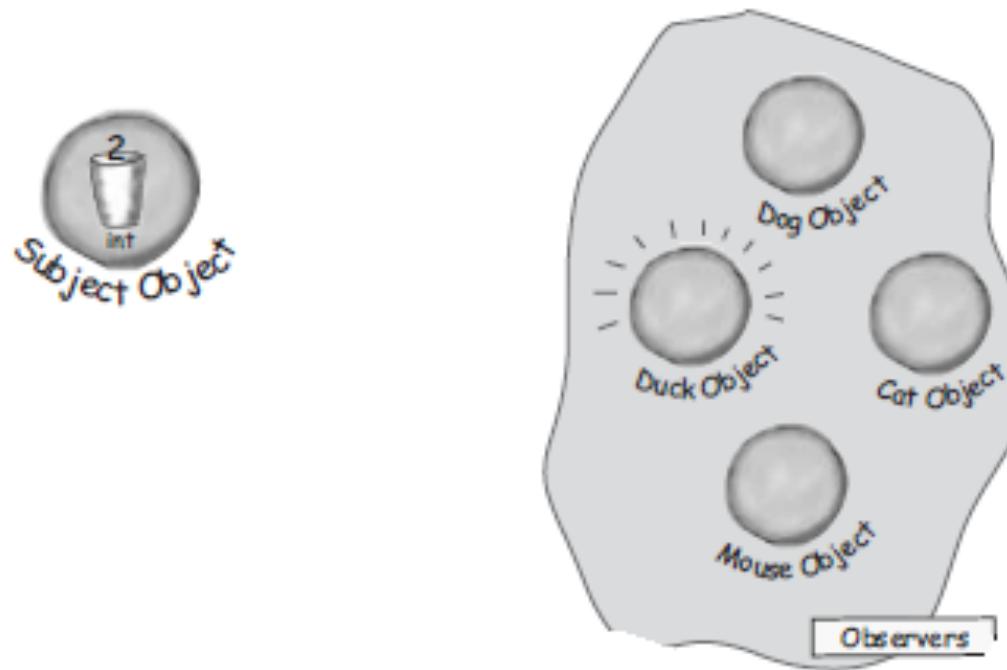
Un día en la vida del patrón Observador

- Un día llega un objeto que quiere registrarse



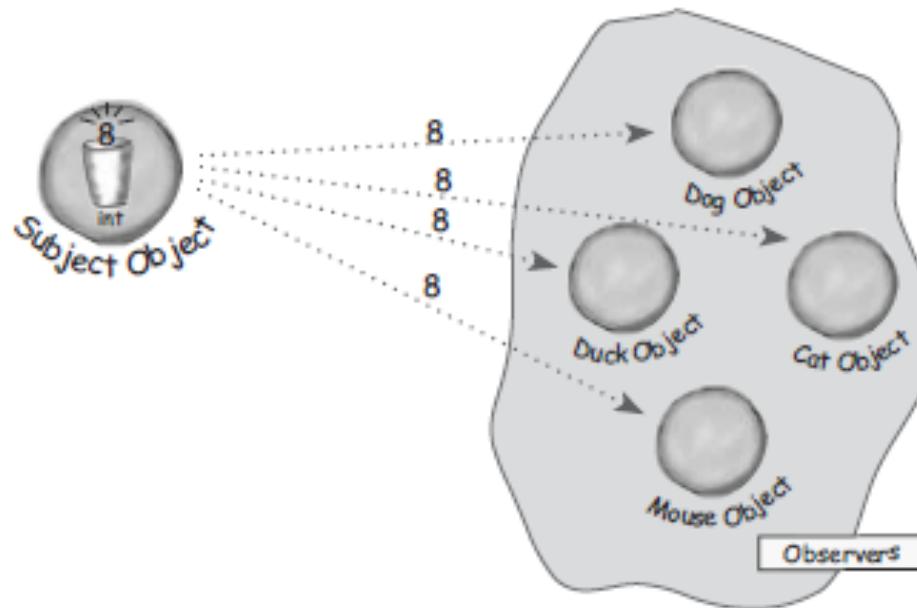
Un día en la vida del patrón Observador

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



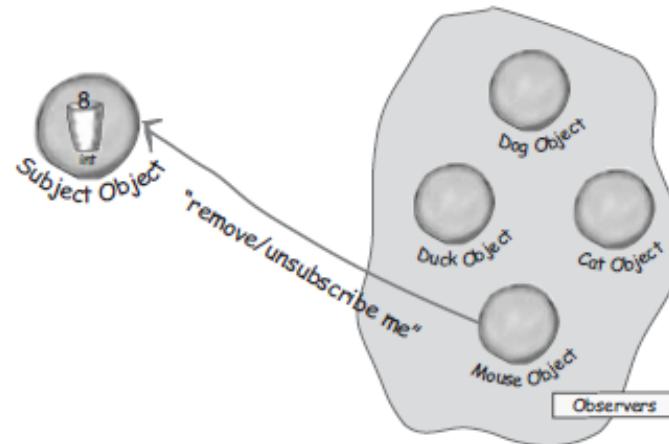
Un día en la vida del patrón Observador

- Los cambios todos los reciben



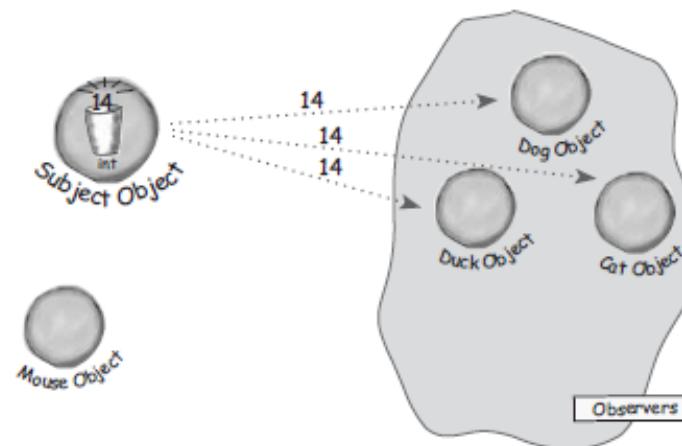
Un día en la vida del patrón Observador

- Ahora el ratón ya no quiere ser observador



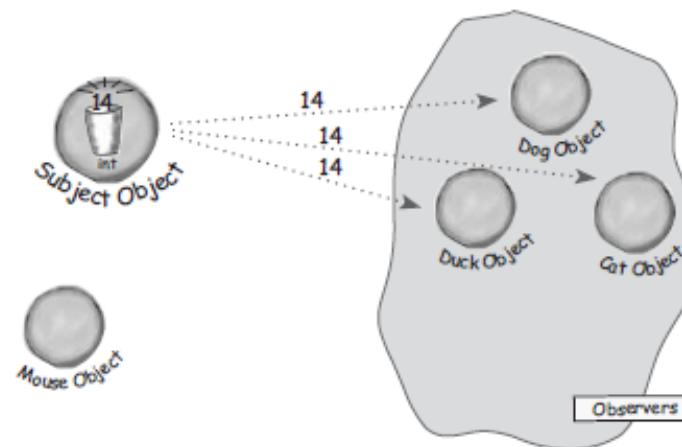
Un día en la vida del patrón Observador

- El ratón se va y ya no recibe notificaciones



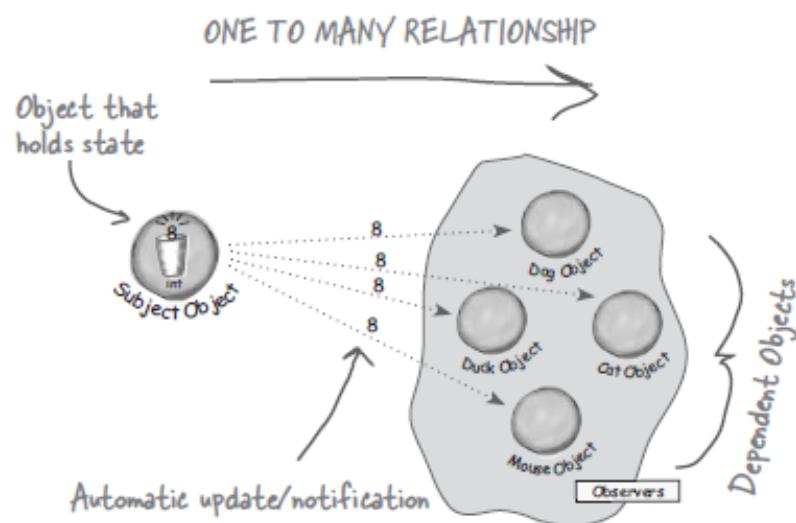
Un día en la vida del patrón Observador

- El ratón se va y ya no recibe notificaciones

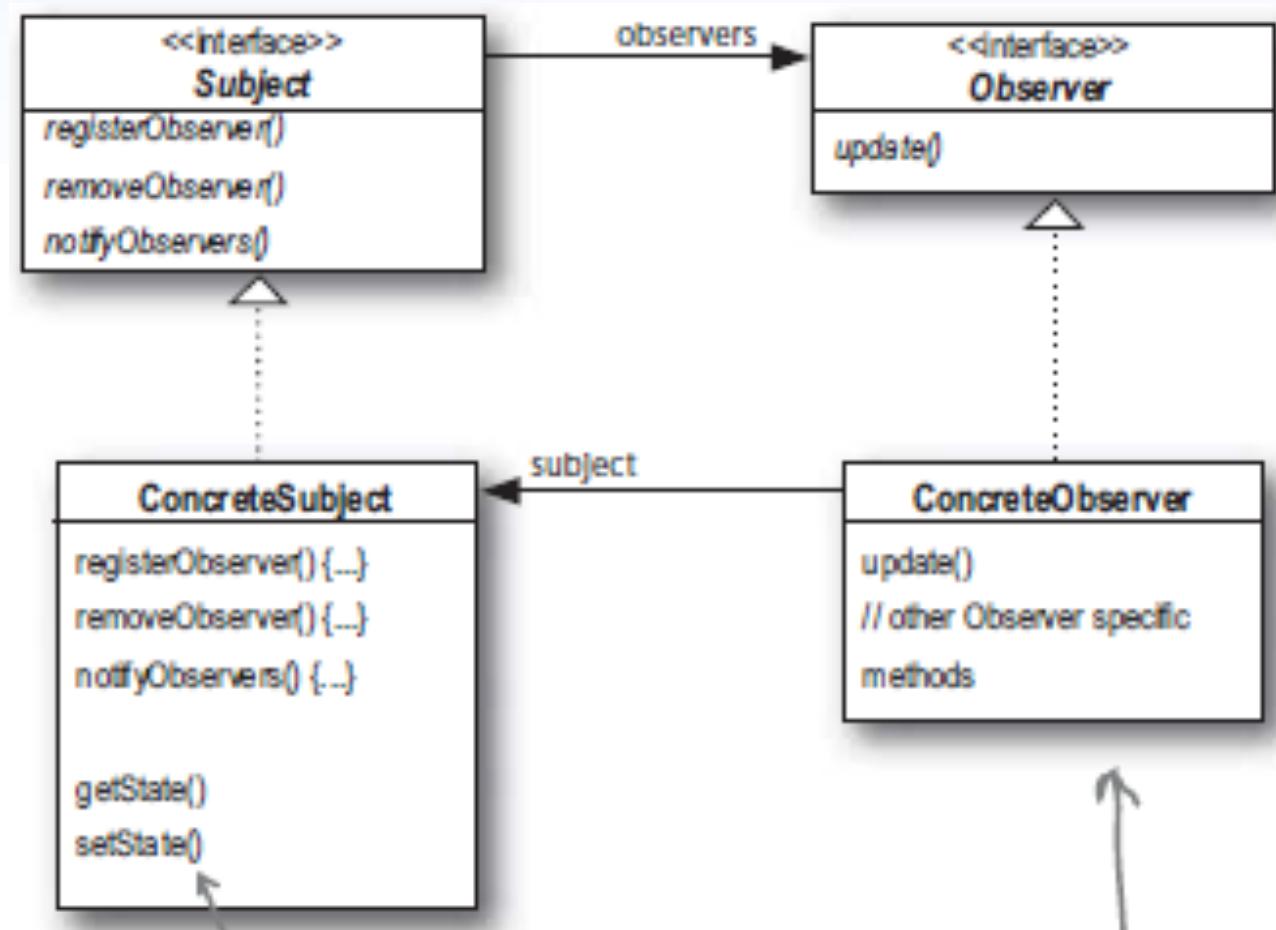


Definición de patrón **Observador**

- 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.

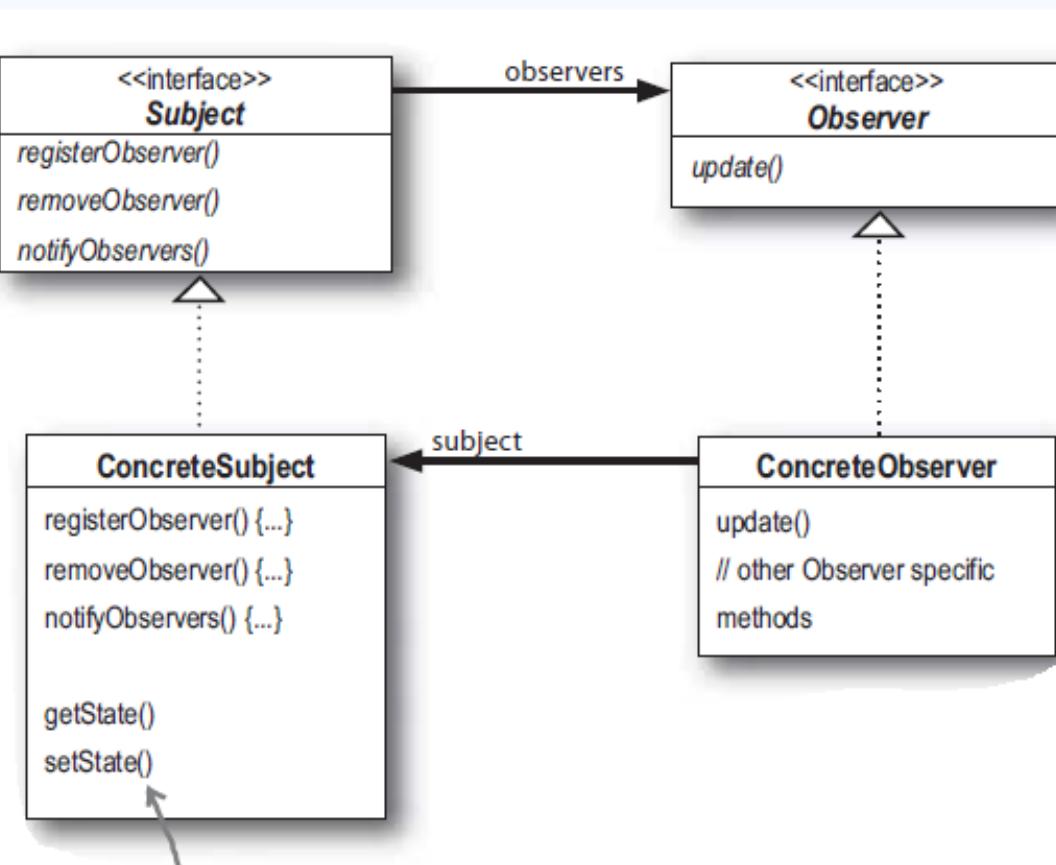


Definición de patrón Observador



Definición de patrón Observador

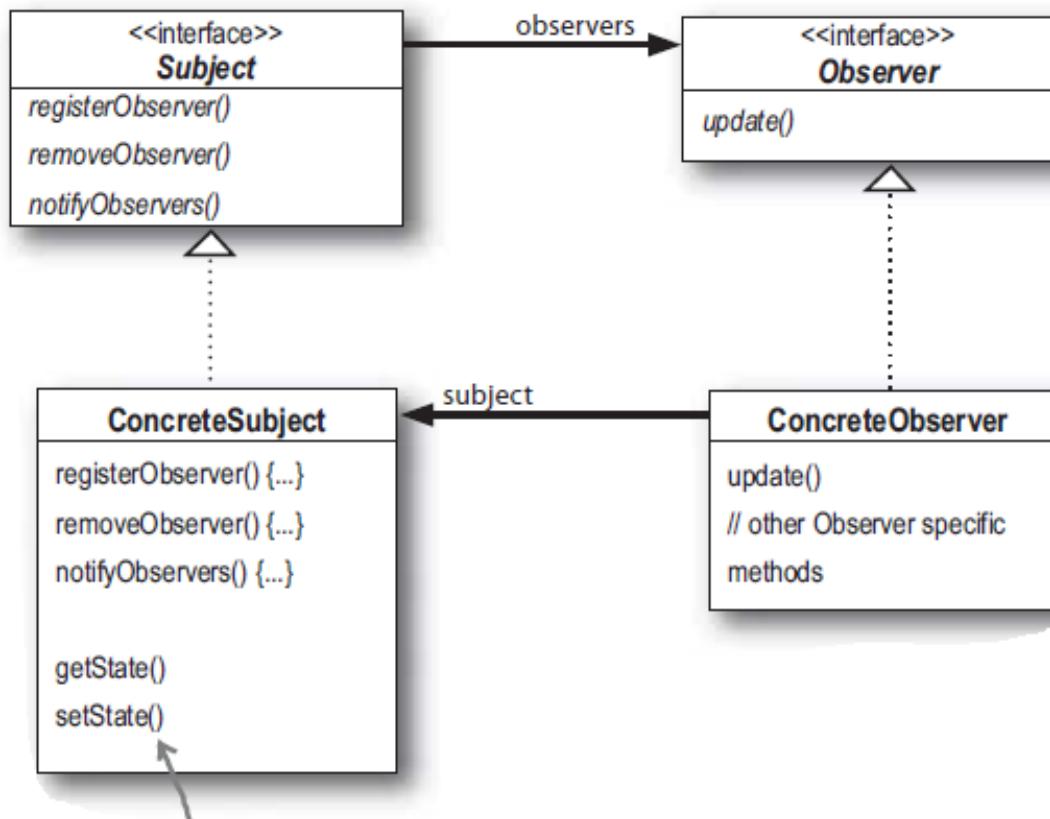
La interfaz del sujeto. Los objetos usan esta para registrarse o removverse como observadores



Definición de patrón Observador

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

Cada sujeto puede tener muchos **observadores**

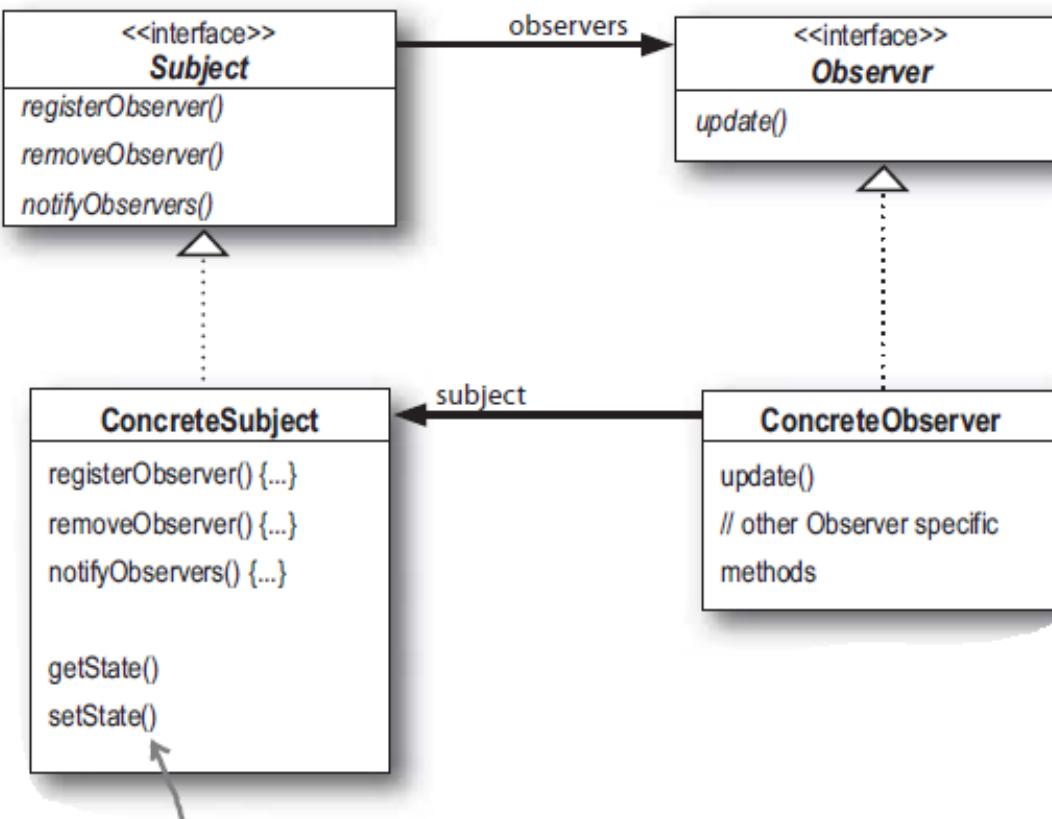


Definición de patrón Observador

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

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

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

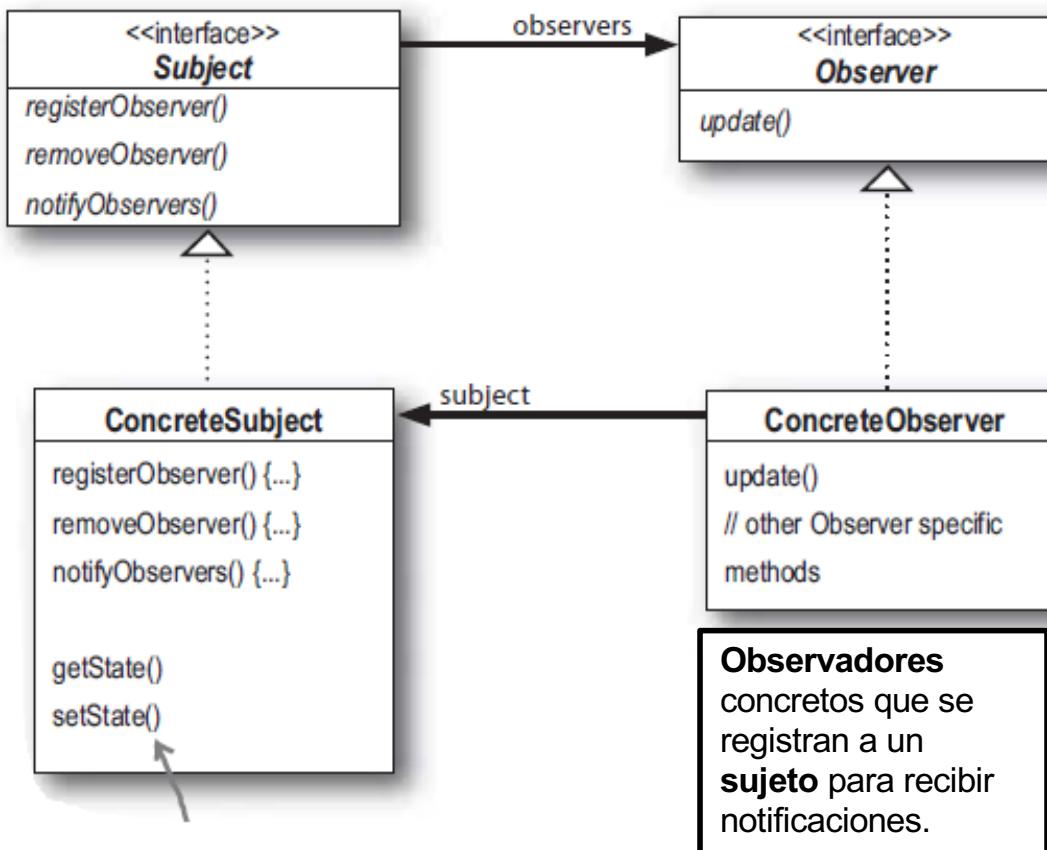


Definición de patrón Observador

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

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

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

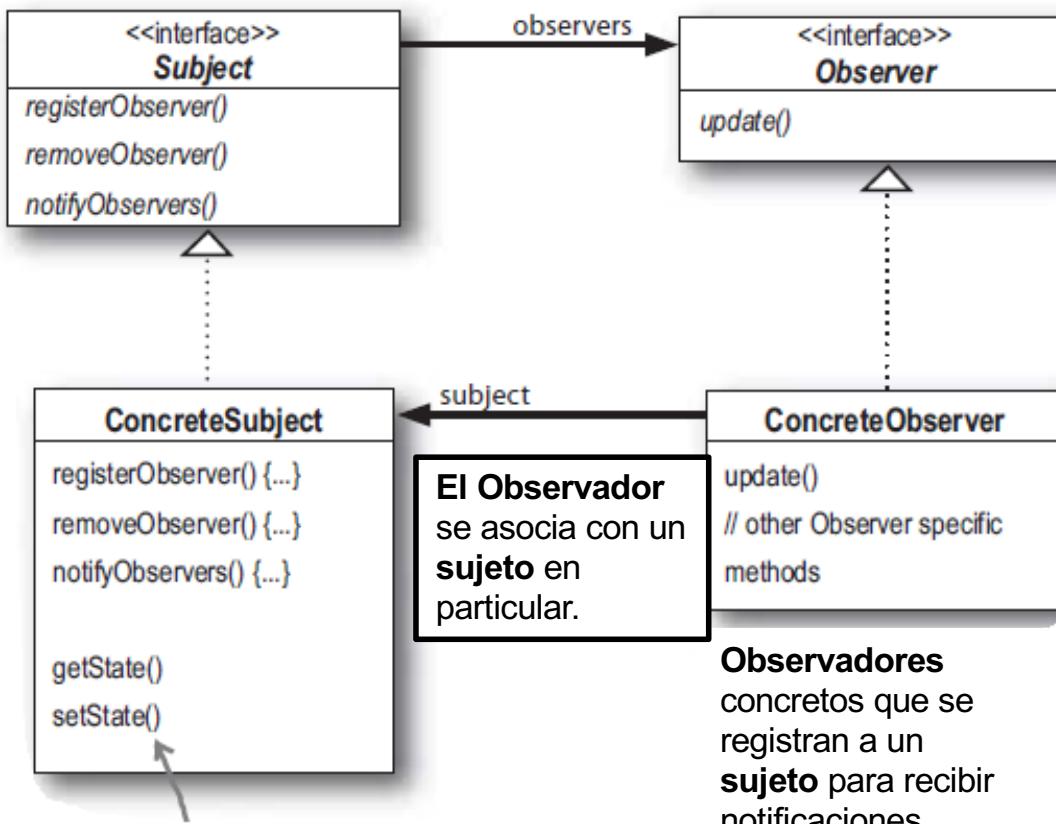


Definición de patrón Observador

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

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

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

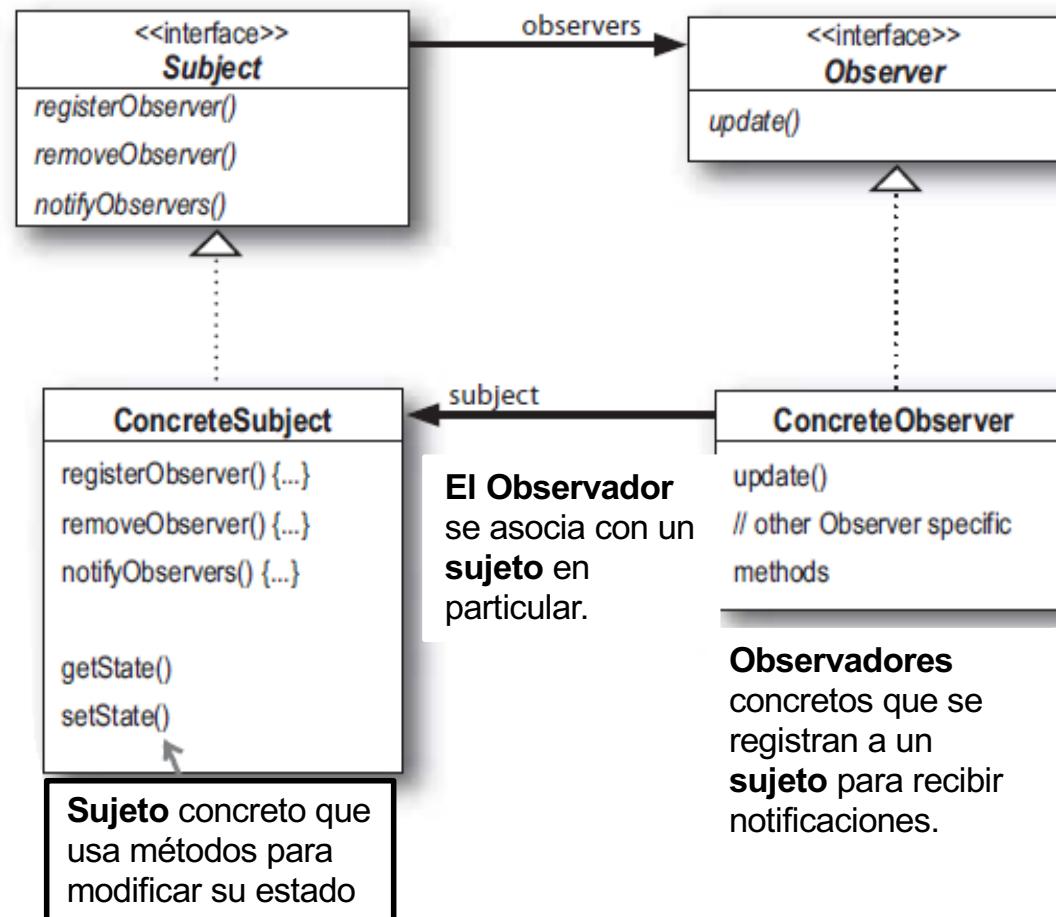


Definición de patrón Observador

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

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

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

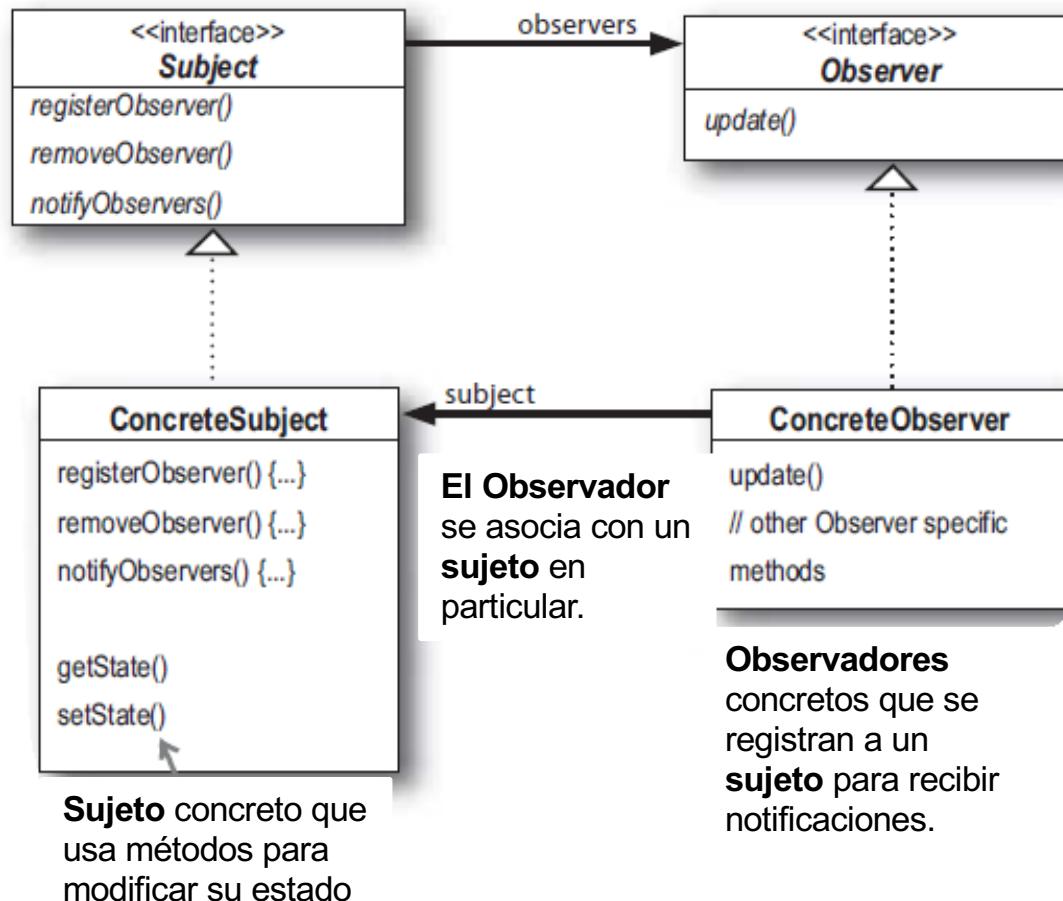


Definición de patrón Observador

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

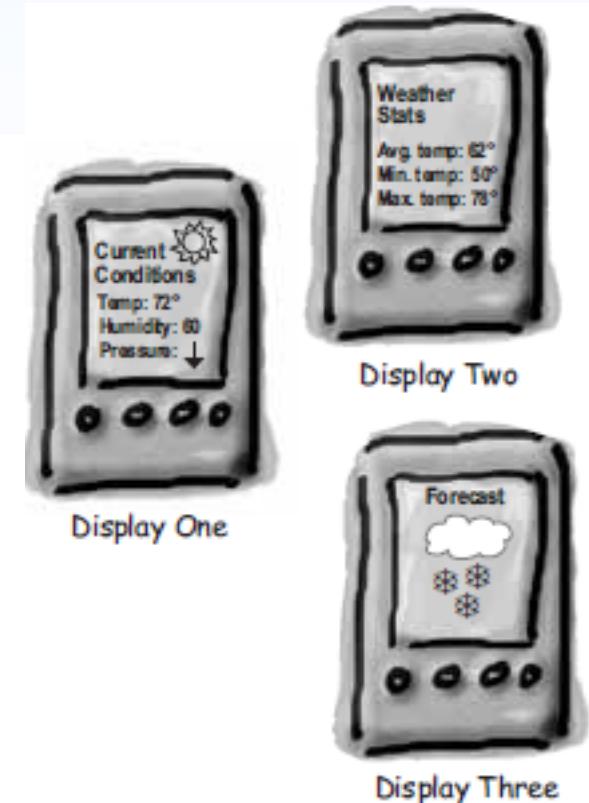
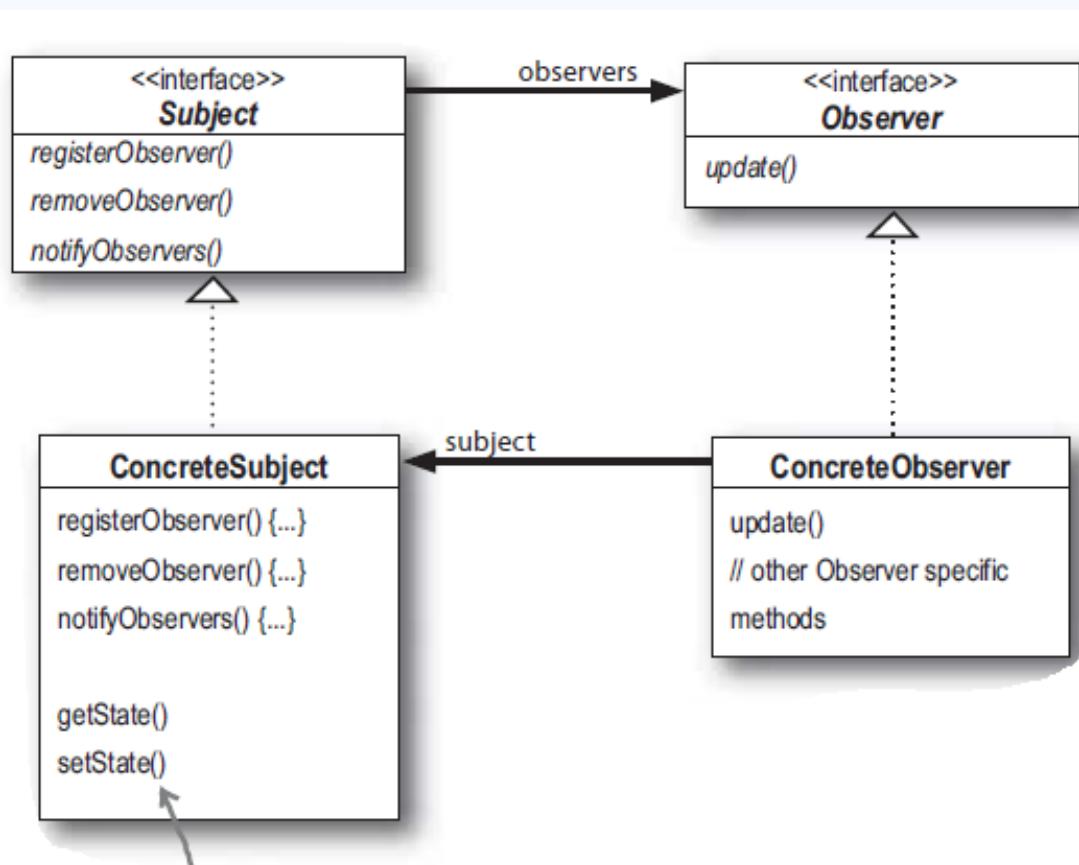
Sujeto concreto
Que debe implementar los métodos

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

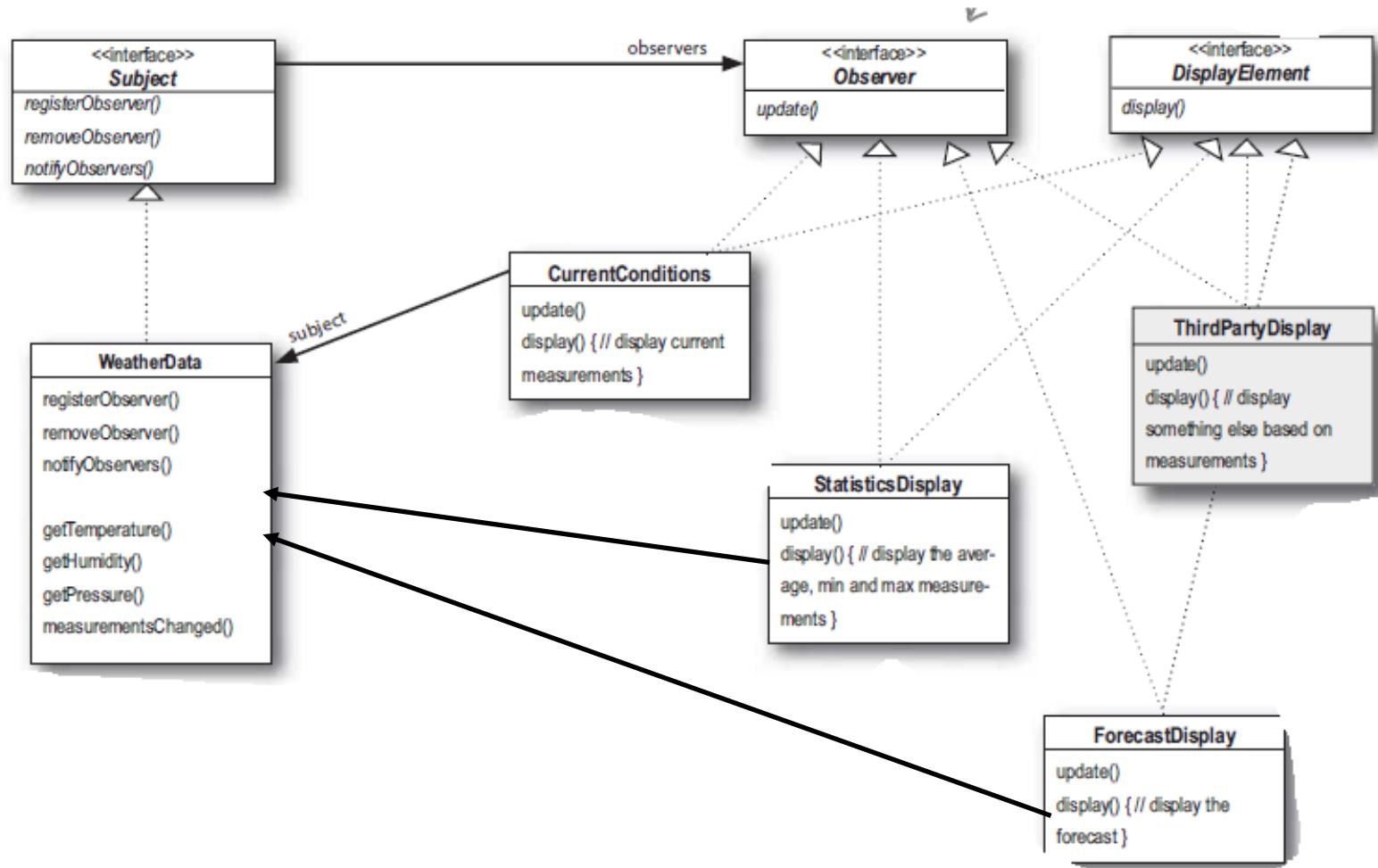


Todos los **observadores** potenciales deben implementar la Interfaz
Sólo un método que llamará el **sujeto**

Diseña las clases a usar para este problema



Diseña las clases a usar para este problema



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 ();  
}
```

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);  
}
```

```
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) {...}  
}
```

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

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);  
        }  
    }  
}
```

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();
}
```



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);
        }
    }

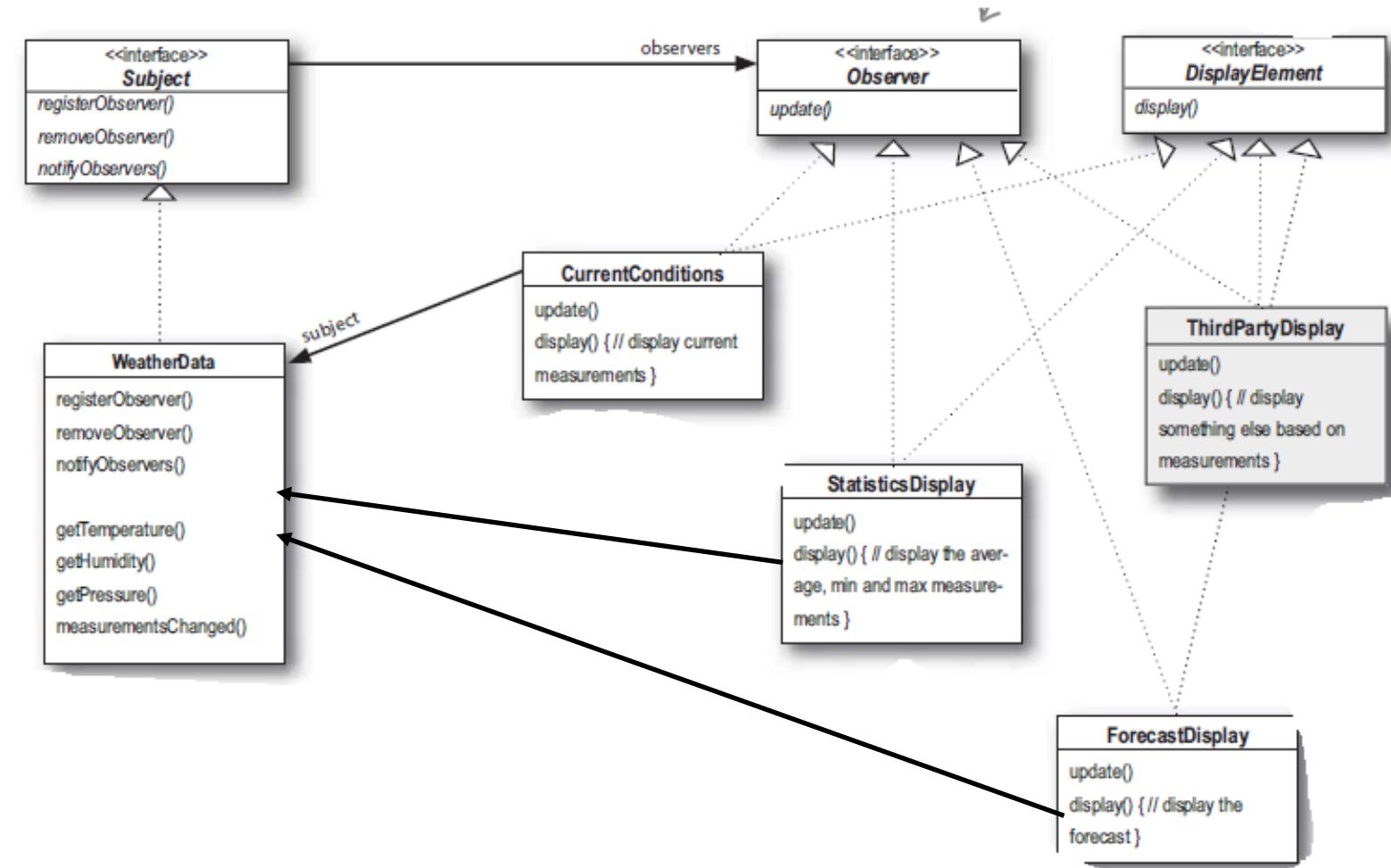
    public void notifyObservers() {
        for (int i=0; i < observers.size(); i++){
            Observer observer = (Observer) observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged (){
        notifyObservers();
    }

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



Y la clase CurrentConditions



[@Juan__Gonzalez](#)

Codificando

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {  
    private float temperature;  
    private float humidity;  
    private Subject weatherData;  
  
    public CurrentConditionsDisplay(Subject weatherData) {  
        this.weatherData = weatherData;  
        weatherData.registerObserver(this);  
    }  
  
    public void update(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        display();  
    }  
  
    public void display() {  
        System.out.println("Current conditions: " + temperature  
            + "F degrees and " + humidity + "% humidity");  
    }  
}
```

, as an ~

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" );  
    }  
}
```

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

```
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");
        }
    }
}
```

@Juan_Gonzalez

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

The screenshot shows a Java code editor with the following code:

```
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);
    }
}
```

The code uses the Observer pattern to display current, statistics, and forecast data for a weather station. The `ForecastDisplay` line is highlighted in yellow.

Below the code editor is a terminal window titled "Output - Observer Pattern (run)" showing the program's output:

```
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
```

¿Dónde vemos más observadores?

- JButton sujeto
- Lleno de observadores
 - Listeners

```
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 here
    }

    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!");
        }
    }
}
```

@Juan_Gonzalez

Tarea para el proyecto

- Tiene sentido definir las relaciones alumno, curso con el patrón observador.
- ¿Qué otras?
- ¿Qué me dicen del patrón Estrategia?
- Hacer un re-modelado de clases considerando estos patrones.
- Hacer el código que pruebe tus modelos.

OBSERVABLE OBJECTS IN JAVA

¿Dónde vemos más observadores?

- JButton sujeto
- Lleno de observadores
 - Listeners

```
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!");
        }
    }
}
```

The screenshot shows a Java application window titled "Should I do it?" running in a blue-themed IDE. The window contains a single button labeled "Should I do it?". Below the window, the Java code is displayed, showing the creation of a frame, addition of a button, and implementation of ActionListener for two inner classes, AngelListener and DevilListener.

```
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!");
    }
}

class DevilListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        System.out.println("Come on, do it!");
    }
}
34
35
36
37
38
39
40
41
```

Output - Observer Pattern (run) *

Icon	Text
Run	run: Come on, do it!
Stop	Don't do it, you might regret it!
Reset	Come on, do it!
Stop and Reset	Don't do it, you might regret it!

[@Juan__Gonzalez](#)

[@Juan__Gonzalez](#)