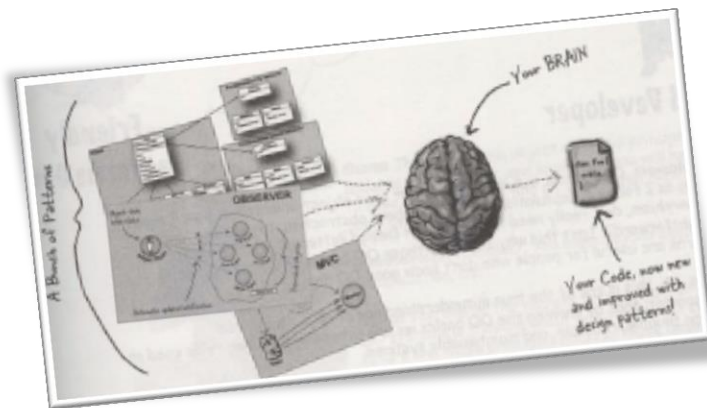




Ingeniería del Software



Introducción a Patrones de Diseño

Agenda

- Introducción a los patrones de Diseño
- El patrón de diseño “Observer”
- El patrón de diseño “Strategy”
- Mi primer patrón de diseño, ejecutemos un “Singleton”



Introducción a los patrones de Diseño



- Bases de OO

- Herencia
- Abstracción
- Polimorfismo
- Encapsulación

Permite cambiar implementación a run-time y encapsula algoritmos

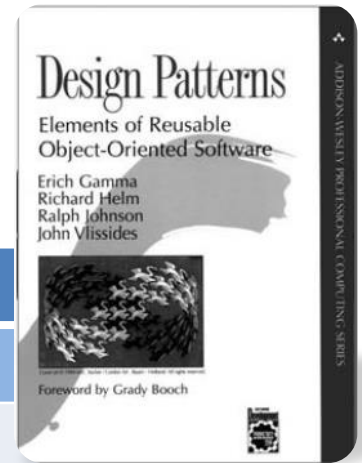
- Principios de OO


- Encapsular lo que varía
- Favorecer **composición** a herencia
- Programar a interfaces y no a implementaciones

"Tiene un"
"Es un"

<<interfaces>>

Catálogo de 22 Patrones de Diseño según GoF (el grupo de los 4)



		Propósito <i>(es lo que el patrón hace)</i>		
		Creacional	Estructural	Conductual
	Clases	Factory 	Adapter	Interpreter Template
		Abstract Factory	Bridge	Chain of Responsibility
Alcance <i>(si el patrón aplica principalmente a clases o a objetos)</i>	Objetos	Builder	Composite	Command
		Prototype	Decorator	Iterator
		Singleton	Facade	Mediator
			Proxy	Memento
				Flyweight
				Observer
				State
				Strategy
				Visitor

Asegura que solo exista una instancia de sí mismo y provee una forma fácil de excederlo

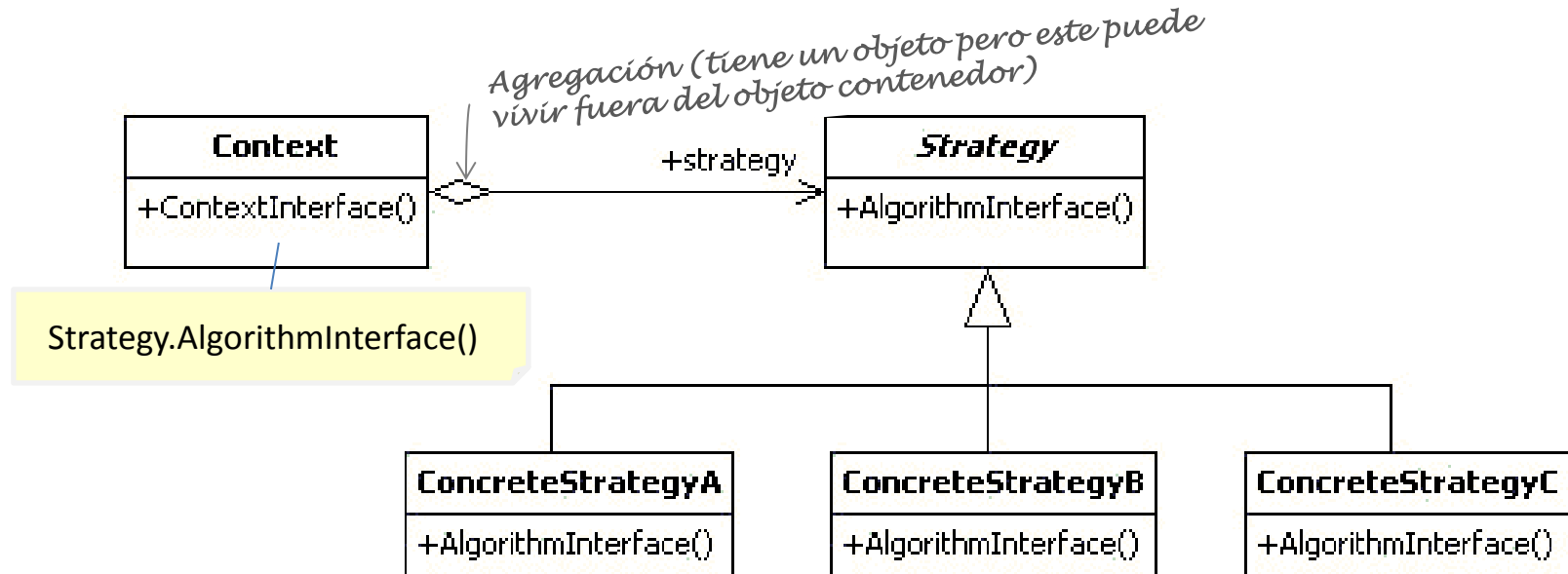
Permite actualizar información a objetos subscriptos asincrónicamente

encapsula algoritmos y permite su extensión y cambio en tiempo de ejecución



Strategy:

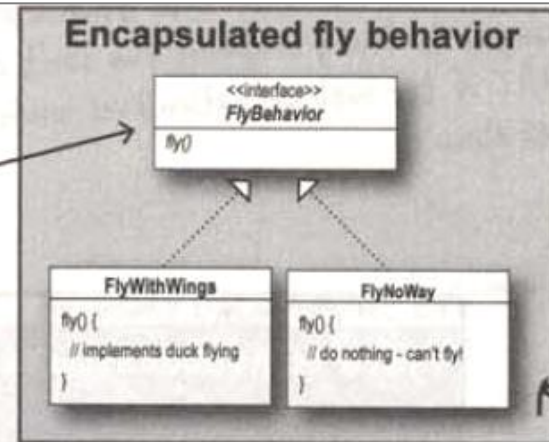
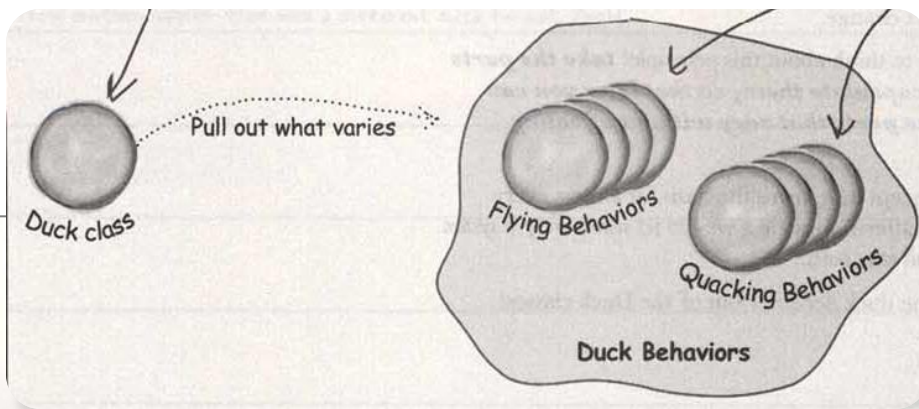
Define una familia de algoritmos, encapsula cada uno, y los hace intercambiables



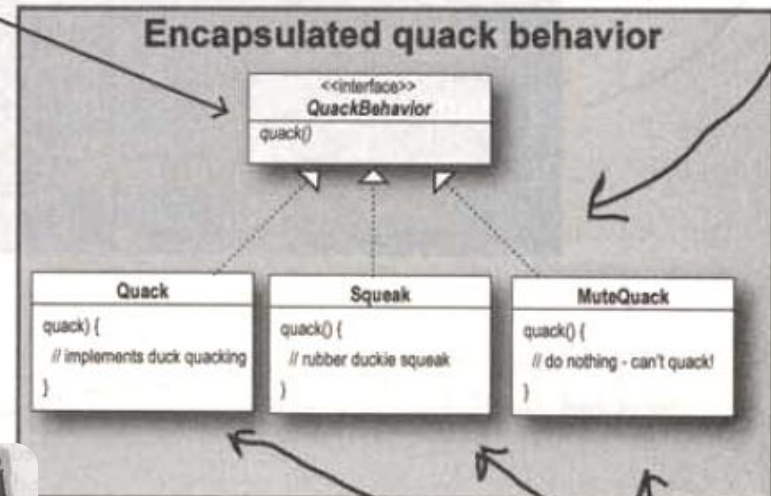
- Usa agregación para minimizar el acoplamiento
- Alternativa a usar switches y cases
- Usado en el MVC como controlador para invocar diferentes Models



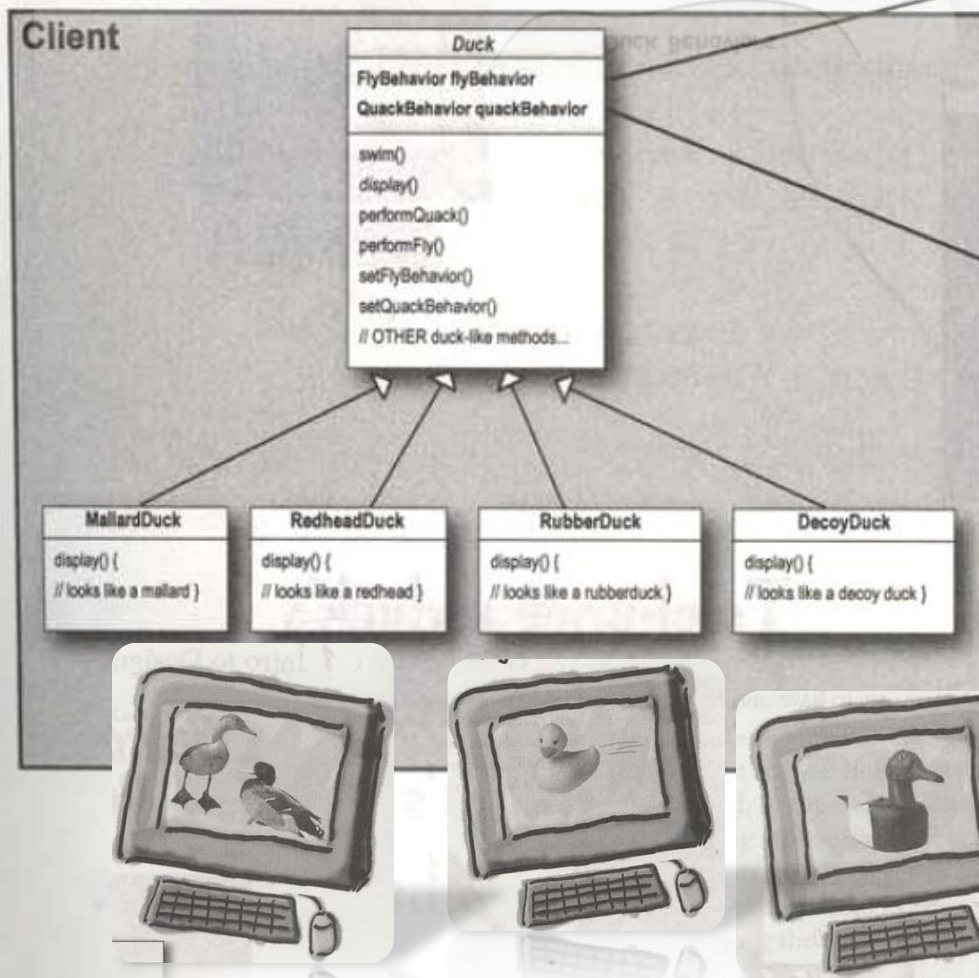
Strategy: Diseño (encapsulación de conductas)



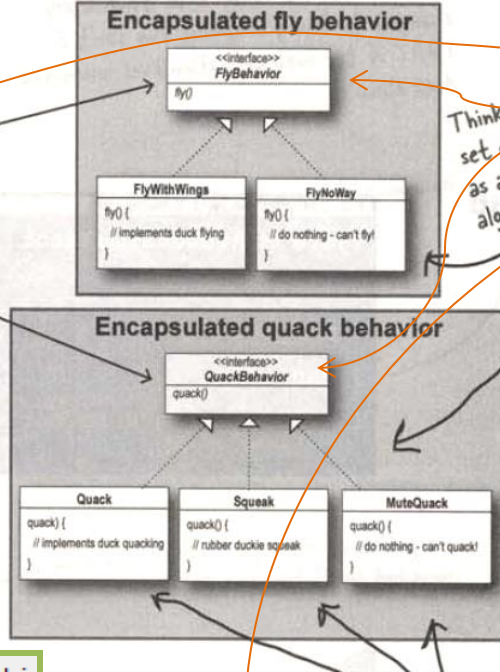
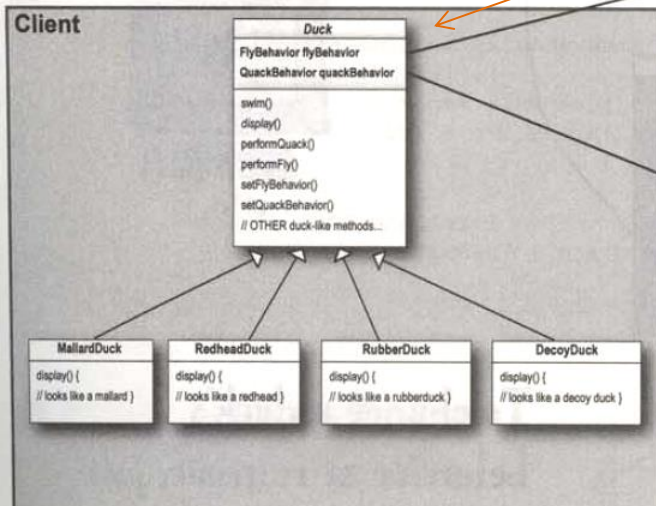
Think of each set of behaviors as a family of algorithms.



These behaviors "algorithms" are interchangeable.



Client makes use of an encapsulated family of algorithms for both flying and quacking.



```

MiniDuckSimulator.java Duck.java FlyBehavior.java
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {}

    public void setFlyBehavior (FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

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

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
  
```

Agregación (HAS-A)

Think set of as a algo

Polimorfismo y delegación

```

MiniDuckSimulator.java ModelDuck.java Duck.java
  
```

```

public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }
}
  
```

```

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

```

MiniDuckSimulator.java Duck.java FlyBehavior.java
public class MiniDuckSimulator {
    public static void main(String[] args) {
  
```

```

MiniDuckSimulator.java MallardDuck.java ModelDuck.java
  
```

```

public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
  
```

```

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

```

Console Problems
<terminated> MiniDuckSimulator
Quack
Squeak
<< Silence >>
I can't fly
I'm flying with a rocket
  
```

```

MallardDuck mallard = new MallardDuck();
RubberDuck rubberDuckie = new RubberDuck();
DecoyDuck decoy = new DecoyDuck();

ModelDuck model = new ModelDuck();

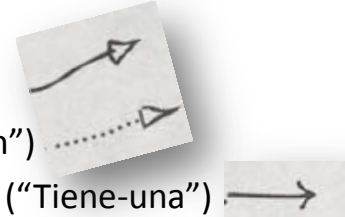
mallard.performQuack();
rubberDuckie.performQuack();
decoy.performQuack();

model.performFly();
model.setFlyBehavior(new FlyRocketPowered());
model.performFly();
  
```

Cambio de funcionalidad en tiempo de ejecución

Strategy: ejercicio

El problema....

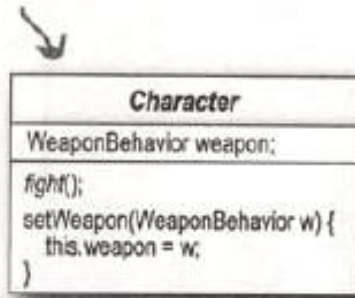
- Organice las clases
- Identifique una clase abstracta y una interfaz
- Dibuje las flechas entre las clases
 - Para herencia (“extienden”)
 - Para una interface (“implementan”)
 - Para una agregación o asociación (“Tiene-una”) 
- Ponga el método “setWeapon()” dentro de la clase correcta


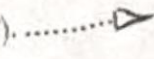



Strategy: ejercicio

La Solución....

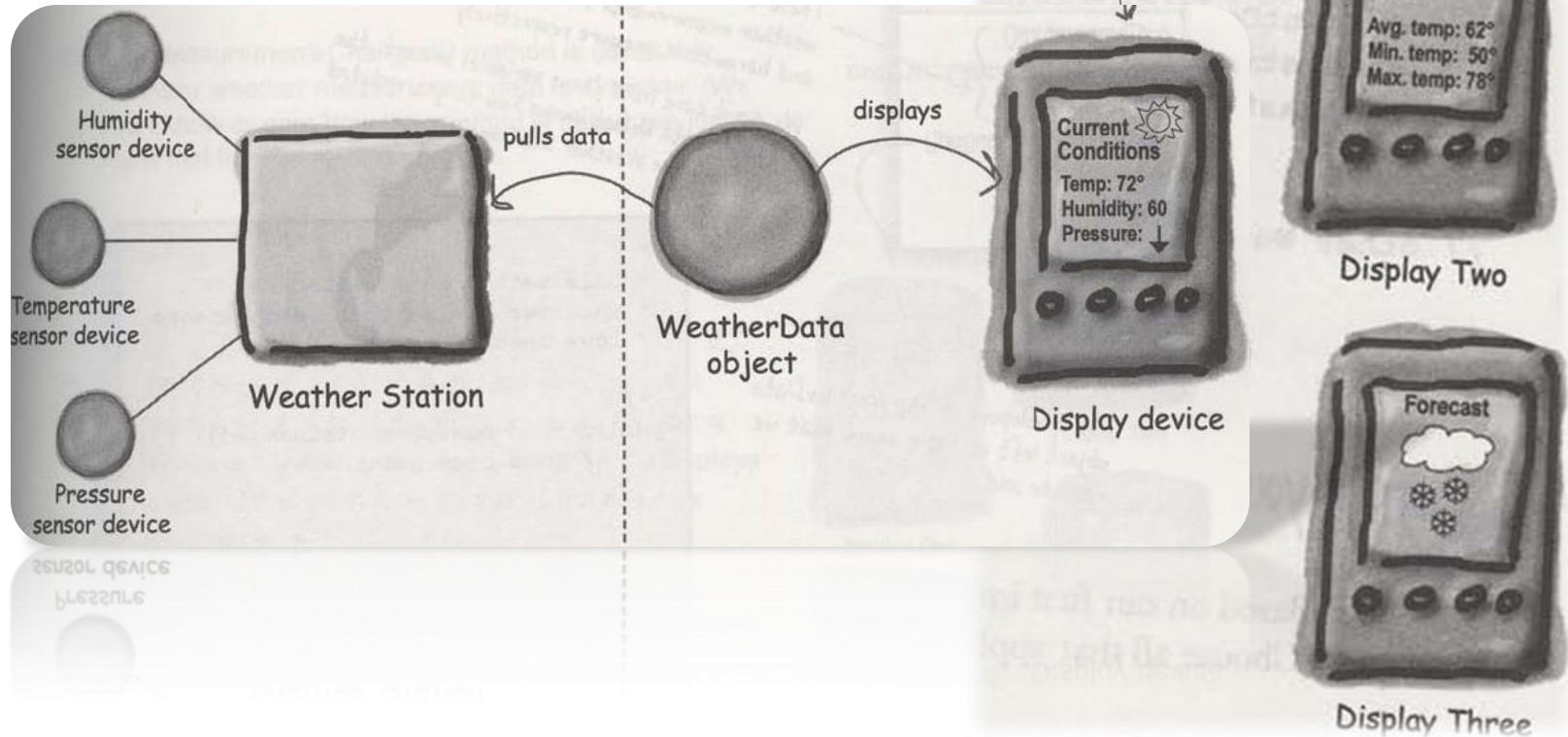
abstract



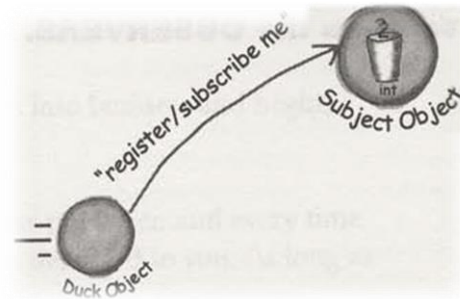
inheritance ("extends"). 
interface ("implements"). 
"HAS-A". 

Observer: el concepto (el modelo de subscripción)

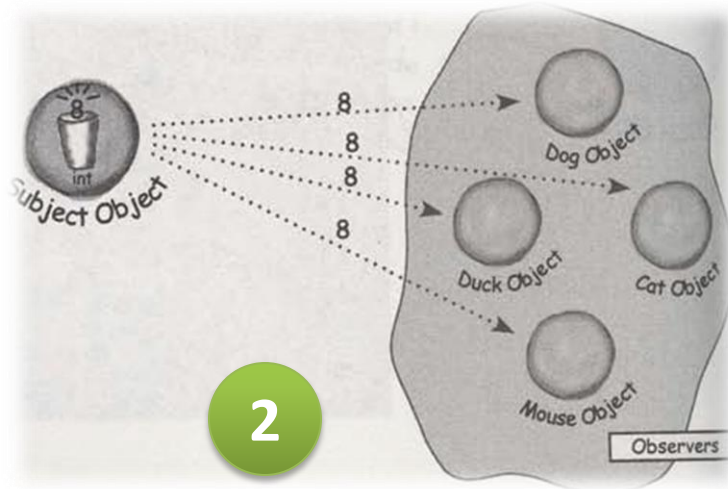
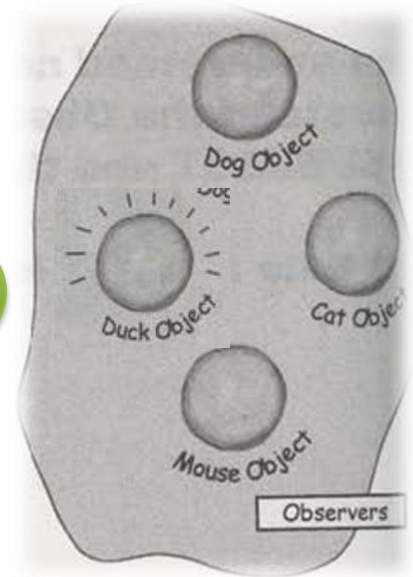
- Necesitamos tener 3 displays:*
1. uno de condición actual
 2. Otro de pronóstico
 3. Otro de estadísticas del tiempo



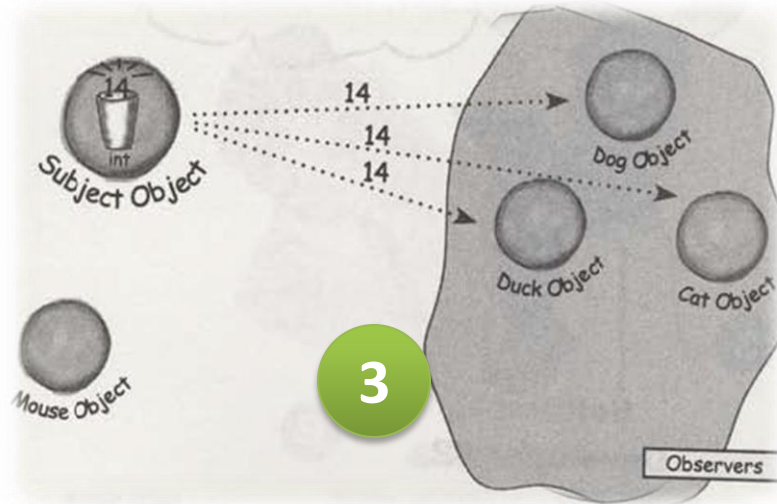
Observer: el concepto (el modelo de subscripción)



1

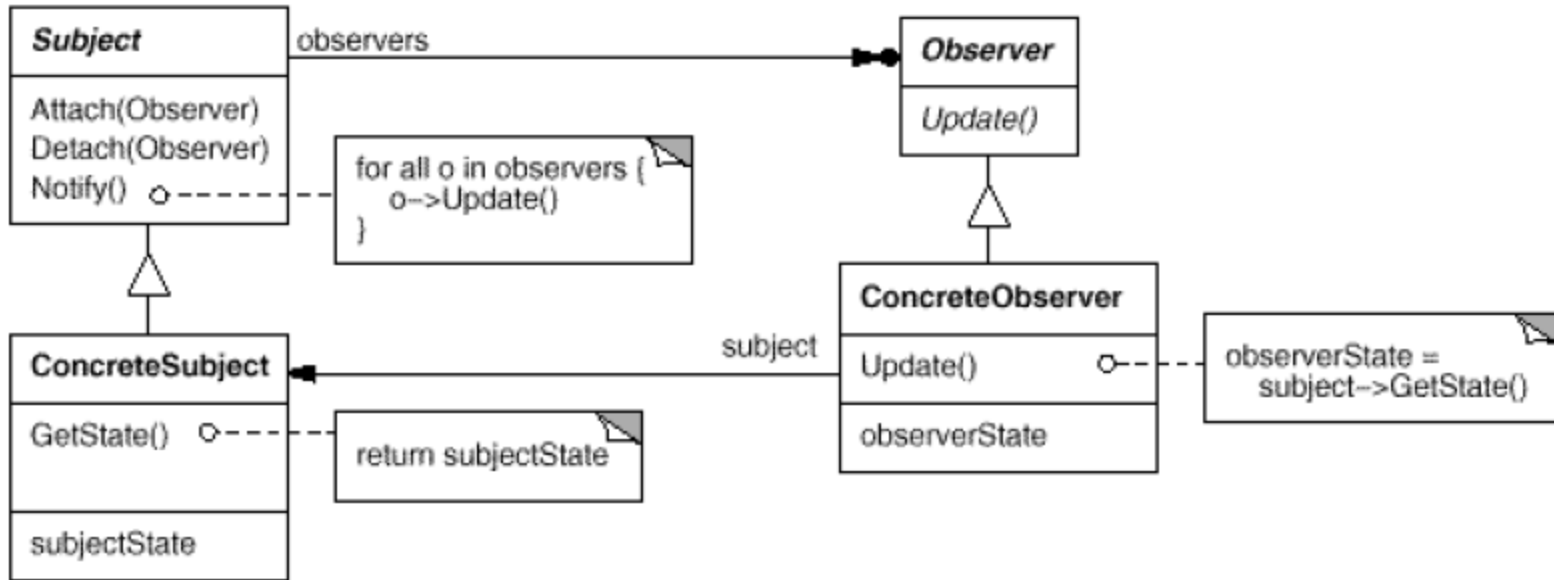


2



3

Observer: Sirve para notificar observadores cuando los valores del sujeto cambian



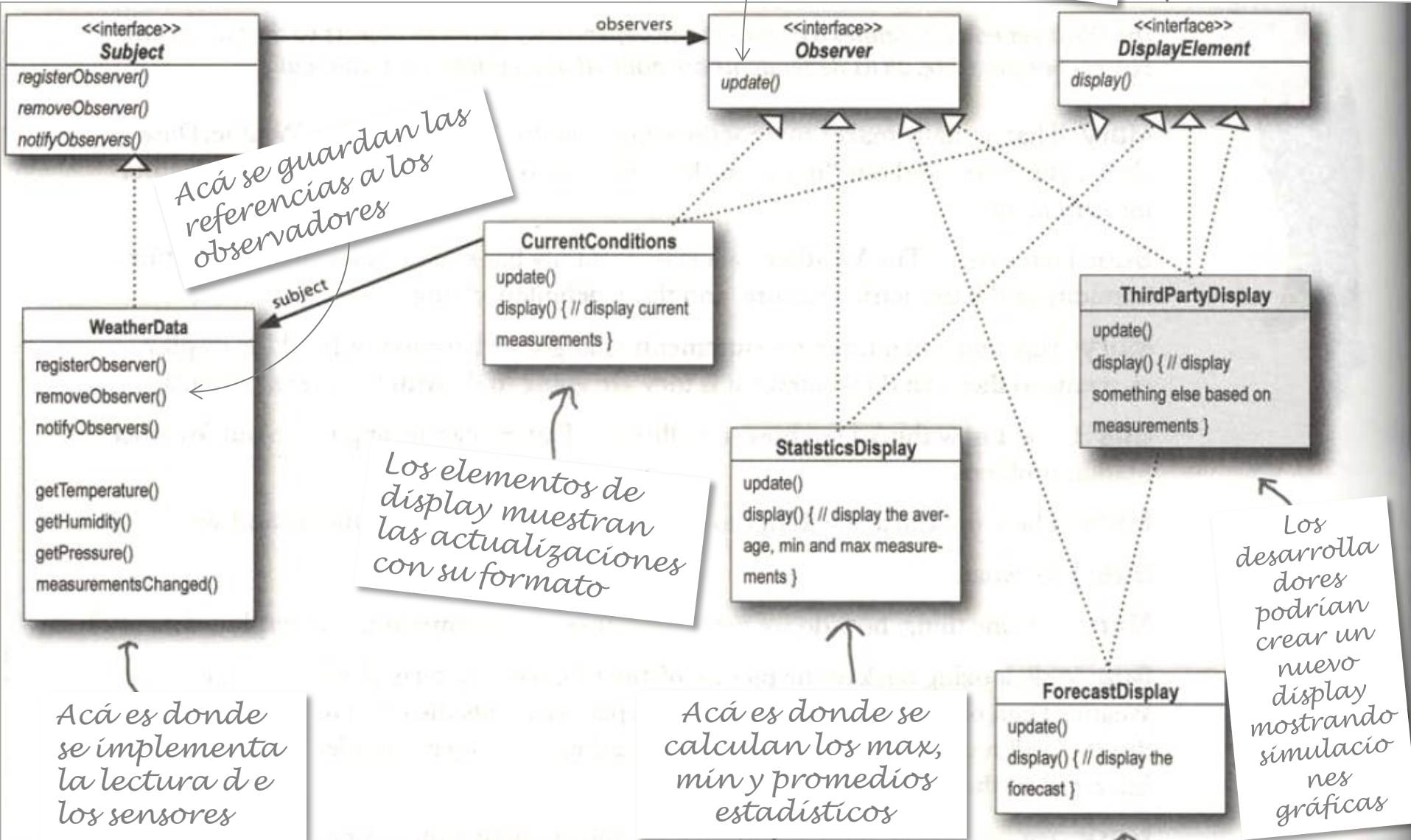
- El sujeto tiene referencias a los obj. observadores para poder avisarles cuando tiene un cambio
- Como tal referencia es de un tipo de interfaz conocida (i.e. `Observer`) se que los observadores tienen que implementar si o si el método público `update()` con los parámetros definidos
- Los observadores tienen una referencia del objeto a observar para poder registrarse en el como observador. Para esto, cada sujeto tiene que implementar al menos los métodos `attachObserver()` y `detachObserver()` para que el observador los pueda llamar.

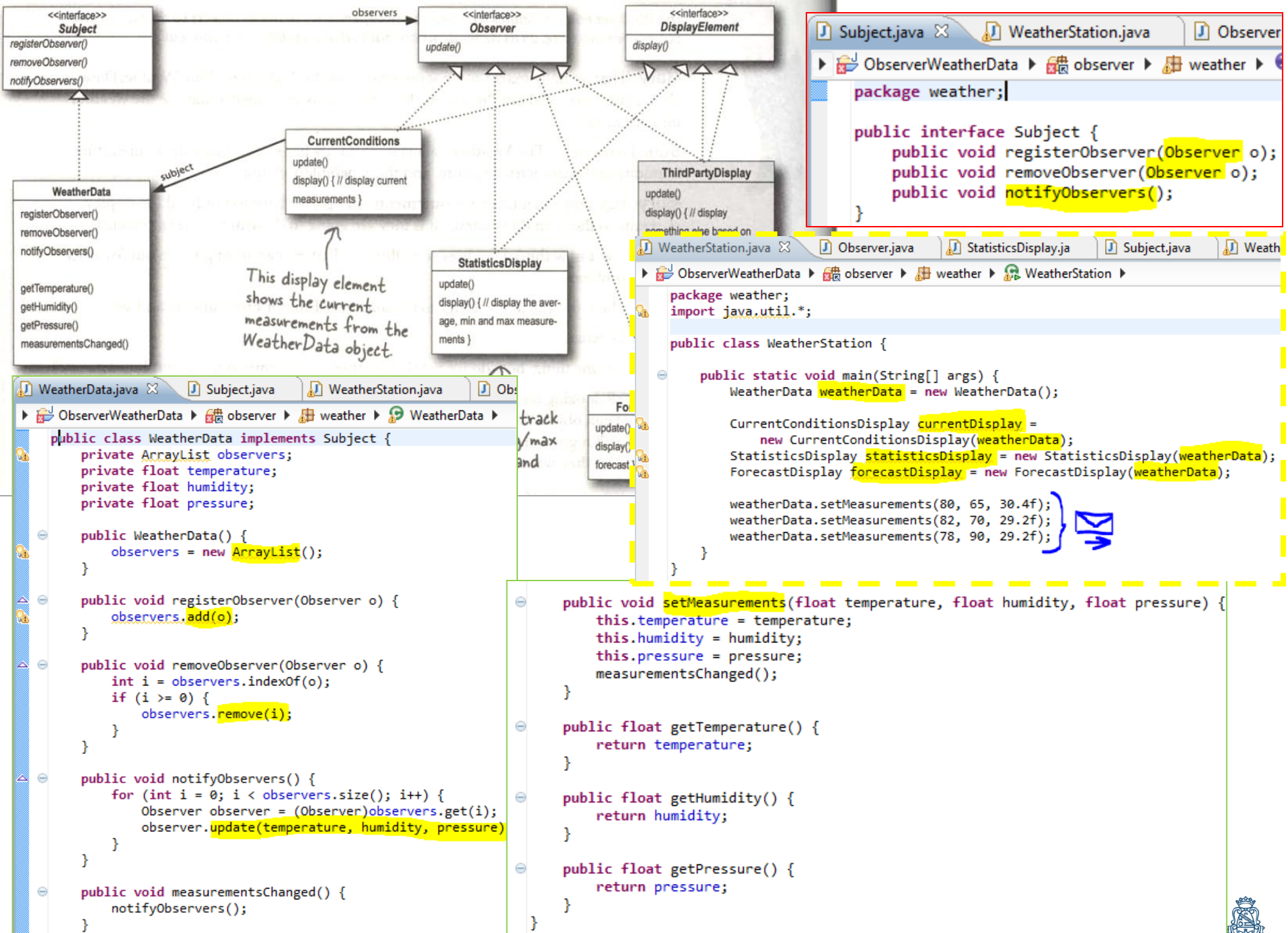


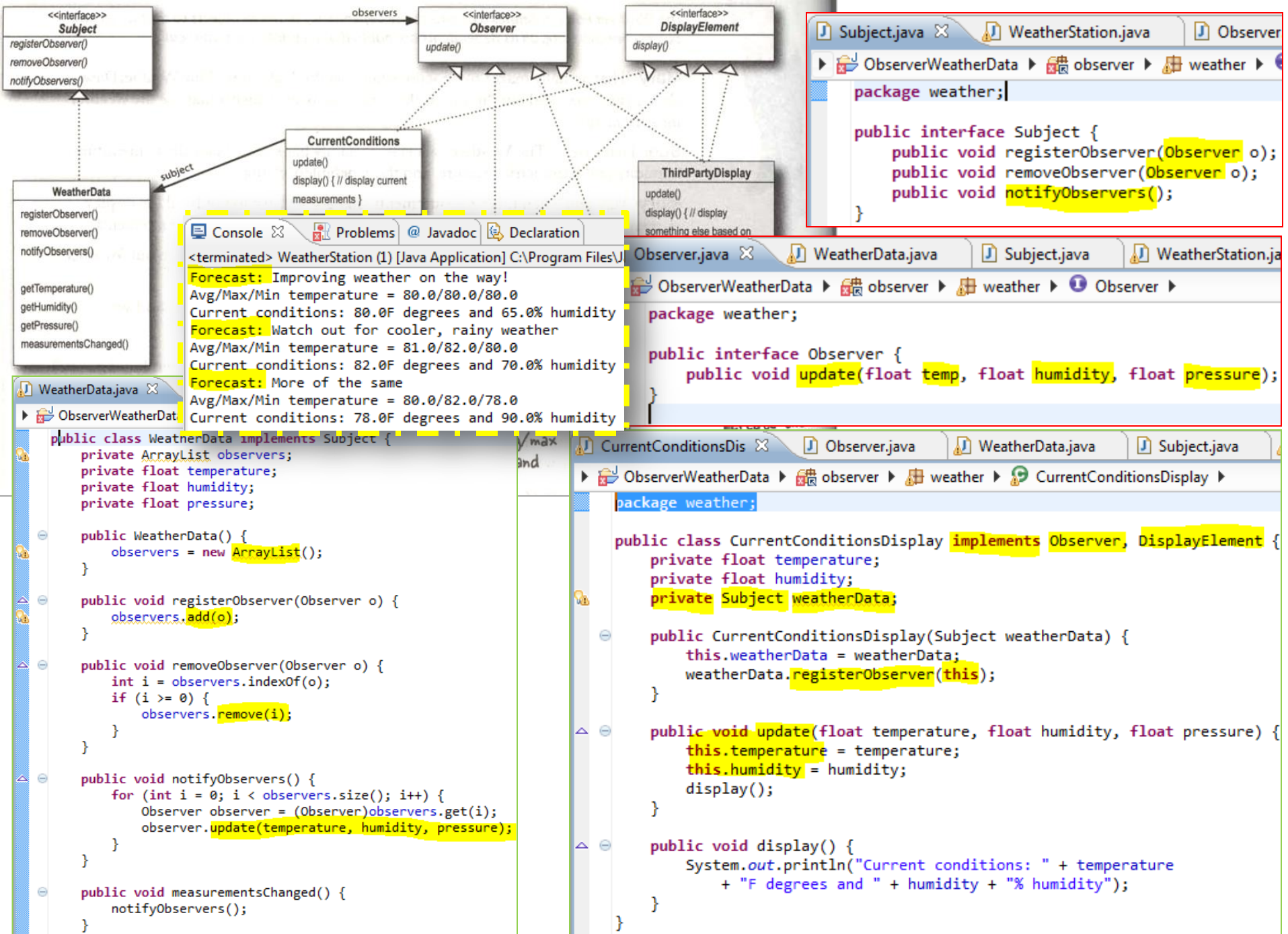
Observer: Diseño (desacoplando mensajes y actualizaciones)

Interfaz obligatoria a implementar para ser llamados

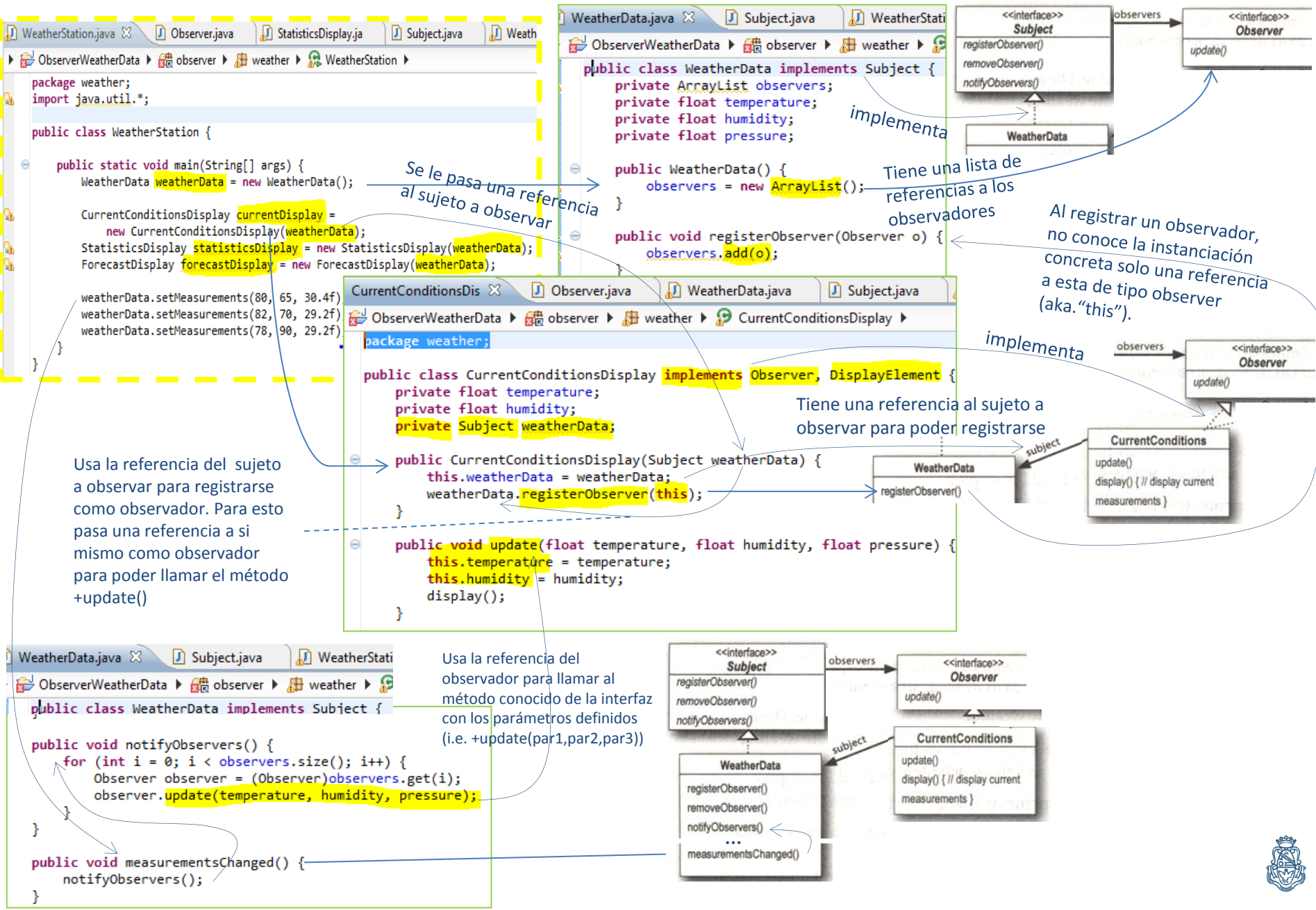
¿Suena familiar? Strategy Pattern





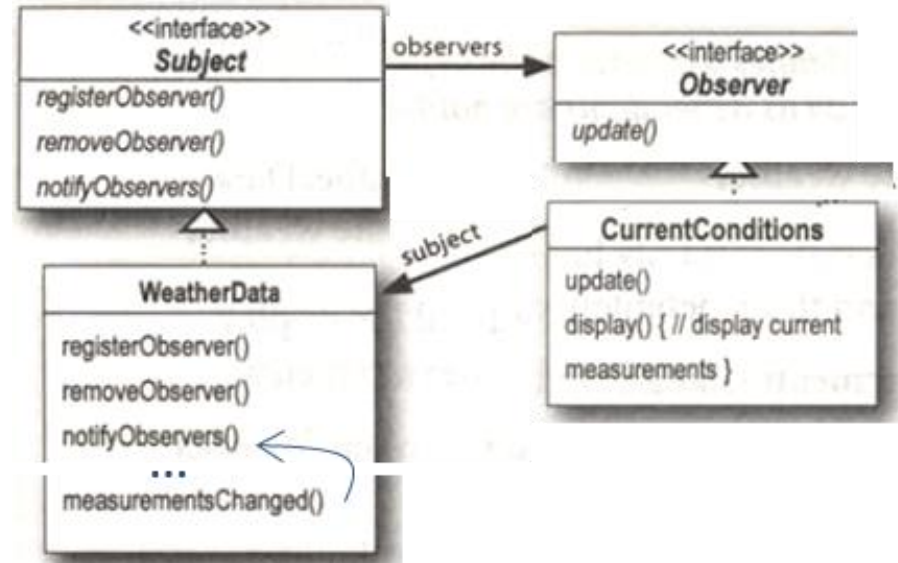


Vista en tiempo de ejecución de un observer ...



Observer: resumen

- El sujeto tiene referencias a los obj. observadores para poder avisarles cuando tiene un cambio
- Como tal referencia es de un tipo de interfaz conocida (i.e. Observer) se que los observadores tienen que implementar si o si el método público update() con los parámetros definidos
- Los observadores tienen una referencia del objeto a observar para poder registrarse en el como observador. Para esto, cada sujeto tiene que implementar al menos los métodos registerObserver() y removeObserver() para que el observador los pueda llamar



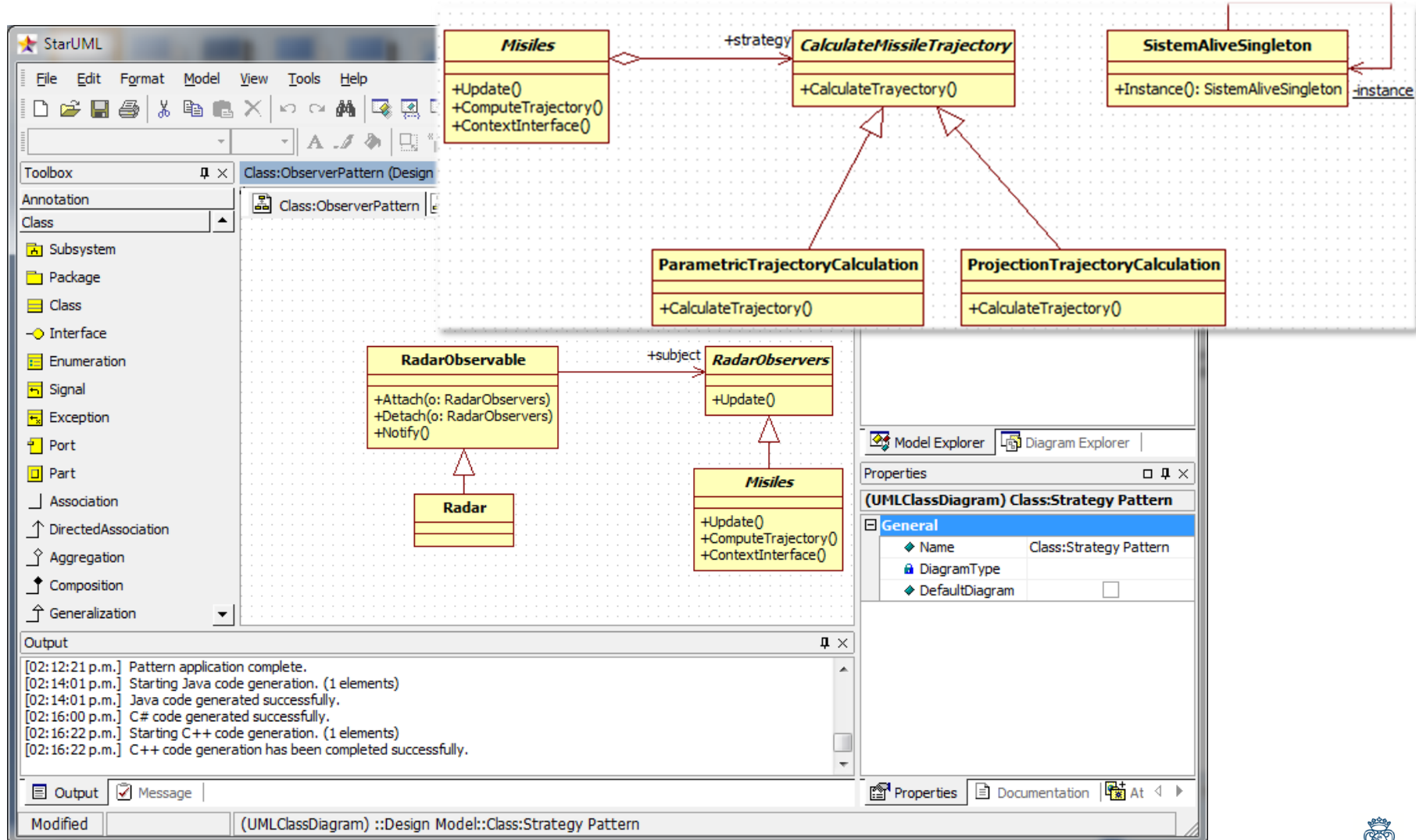
Realice un diagrama de secuencia del Observer

- Usando el ejemplo anterior de una estación del clima con un sólo display (ej. `CurrentConditionsDisplay`), graficar la secuencia de llamadas y creaciones de objetos según se muestra en el siguiente ejemplo resumido de `WeatherStation.java`
- Puede usar `StarUML` o `Lucidcharts` para realizar el ejercicio. Luego responda al email del grupo de la materia con el gráfico, nombre de su grupo e integrantes que participaron.

```
public class WeatherStation {  
    public static void main(String[] args) {  
        WeatherData weatherData = new WeatherData();  
        CurrentConditionsDisplay currentDisplay =  
            new CurrentConditionsDisplay(weatherData);  
        weatherData.setMeasurements(80, 65, 30.4f);  
    }  
}
```



Mirando una herramienta de generación de código en base patrones de diseño en UML



Entendiendo un nuevo patrón (Singleton)

- Buscar en la web la descripción de este patrón
- Dibuje el diagrama de clases
- Escribir y ejecutar el código para implementarlo
- De al menos dos ejemplos de usos
- Pruebe modificarlo para ver que no se crean nuevas instancias agregando un contador de intentos de creación (vea el código de la siguiente filmina)



Singleton: Solución al ejercicio

The image shows a screenshot of an IDE with two Java files: `Singleton.java` and `SingletonClient.java`. The `Singleton.java` file contains the implementation of the Singleton pattern, and the `SingletonClient.java` file contains the client code that uses the Singleton.

Singleton.java

```
// NOTE: This is not thread safe!

public class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
    public String getDescription() {
        return "I'm a classic Singleton!";
    }
}
```

SingletonClient.java

```
public class SingletonClient {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        System.out.println(singleton.getDescription());
    }
}
```

A diagram shows a box labeled "Singleton" with a dashed line pointing to a box labeled "return uniqueInstance".

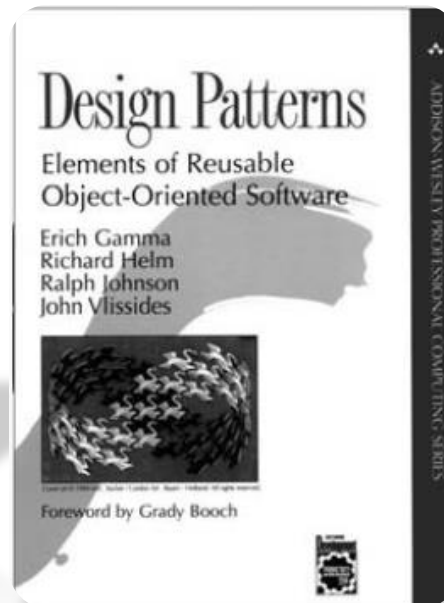
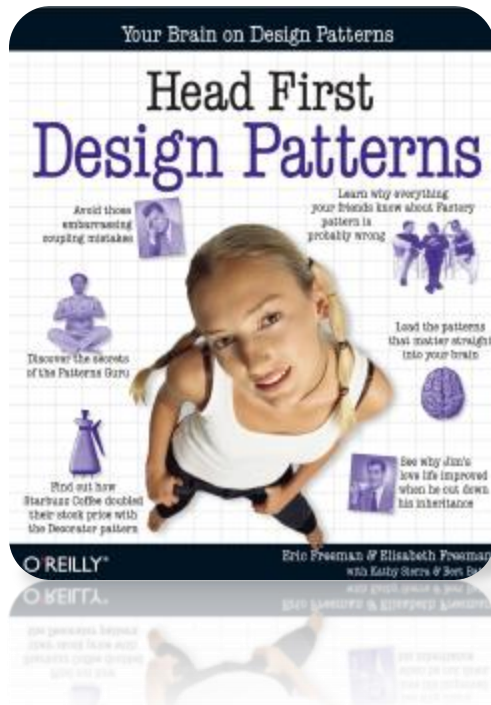
The IDE's console output shows the results of running the application:

```
<terminated> SingletonClient [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (M
I'm a classic Singleton! that was called 1 time(s)
I'm a classic Singleton! that was called 2 time(s)
I'm a classic Singleton! that was called 3 time(s)
```



Bibliografía y referencias

Autor	Título
Eric & Elisabeth Freeman	Head First Design Patterns (2004)
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides	Design Patterns, Elements of Reusable Object-Oriented Software (1995)
Sommerville, Ian	Software Engineering, 9th edition (2007)



¿Preguntas?



Historia de Versiones

Versión	Comentarios	Fecha	Autor
1.0.0	Versión inicial basado en el libro HeadFirst Design Patterns	15-Jun-2012	Martín Miceli
1.0.1	Agregado de explicación contextual para observer y strategy.	26-May-2014	Martín Miceli

