

The background image shows a collaborative work environment. In the foreground, a person's hand points to a specific section of a detailed architectural floor plan spread across a wooden table. To the right, another hand holds a black pen, poised to write or mark the plan. On the left, a white hard hat is partially visible. In the background, another person in a denim shirt is partially seen, holding a smartphone. A laptop is also visible on the right side of the table. A semi-transparent blue rectangular overlay is centered over the image, containing the title and subtitle text.

# PATRONES DE DISEÑO

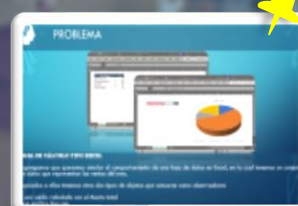
PATRONES DE COMPORTAMIENTO - OBSERVER



# PATRON DE COMPORTAMIENTO - OBSERVER



Propósito



Problema



Solución



Estructura



Aplicabilidad



Consecuencias



Ejercicio



Tarea

Terminado



# PROPÓSITO

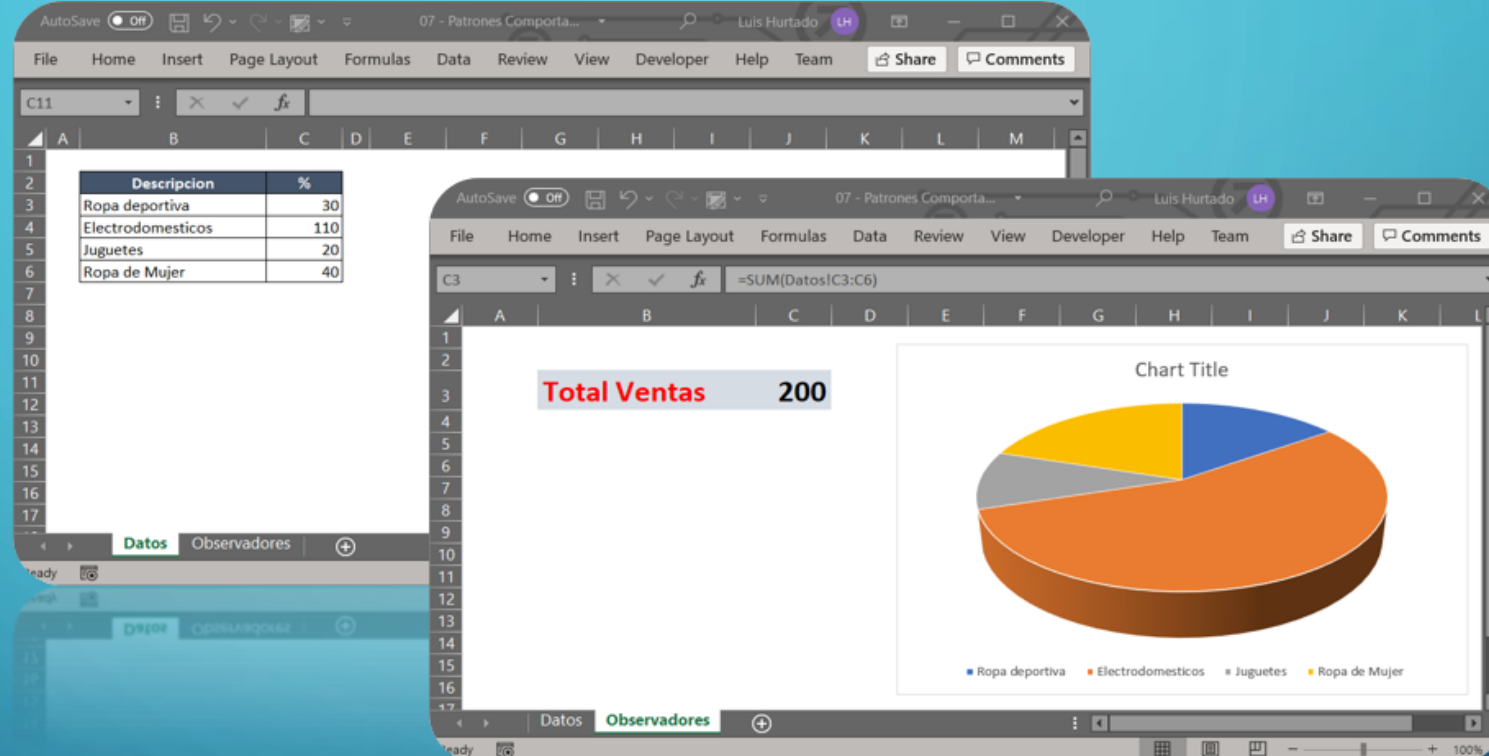
## **Patrón Observador:**

Es un patrón de diseño de software que define una dependencia del tipo uno a muchos entre objetos, de manera que cuando uno de los objetos cambia de estado, se notifica este cambio a todos los dependientes para que se actualicen automáticamente.





# PROBLEMA



## HOJA DE CÁLCULO TIPO EXCEL

Supongamos que queremos simular el comportamiento de una hoja de datos en Excel, en la cual tenemos un conjunto de datos que representan las ventas del mes.

Asociados a ellos tenemos otros dos tipos de objetos que actuaran como observadores:

- una celda calculada con el Monto total
- un grafico tipo pie.



# DEMO

## REVISEMOS EN CÓDIGO LA IMPLEMENTACIÓN DEL PROBLEMA





# SOLUCIÓN

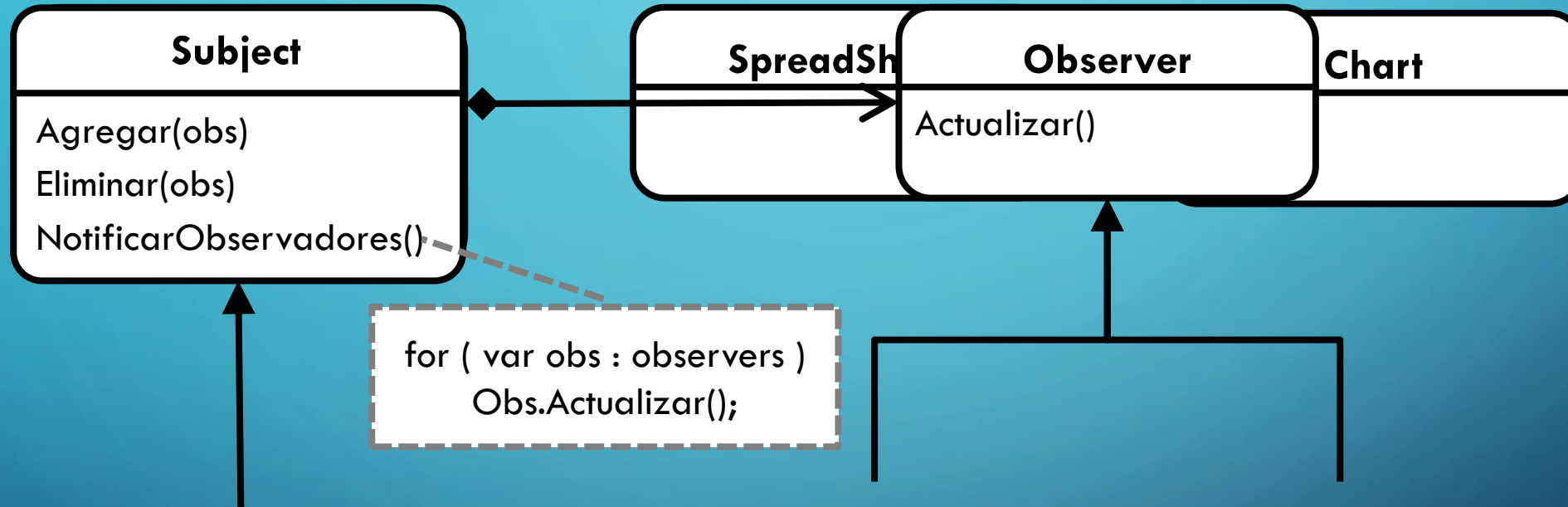


1

2

CLASE ABSTRACTA

INTERFACE





# SOLUCIÓN

CLASE ABSTRACTA

## Subject

Attach(obs)  
Detach(obs)  
Notify()

## ConcreteSubject

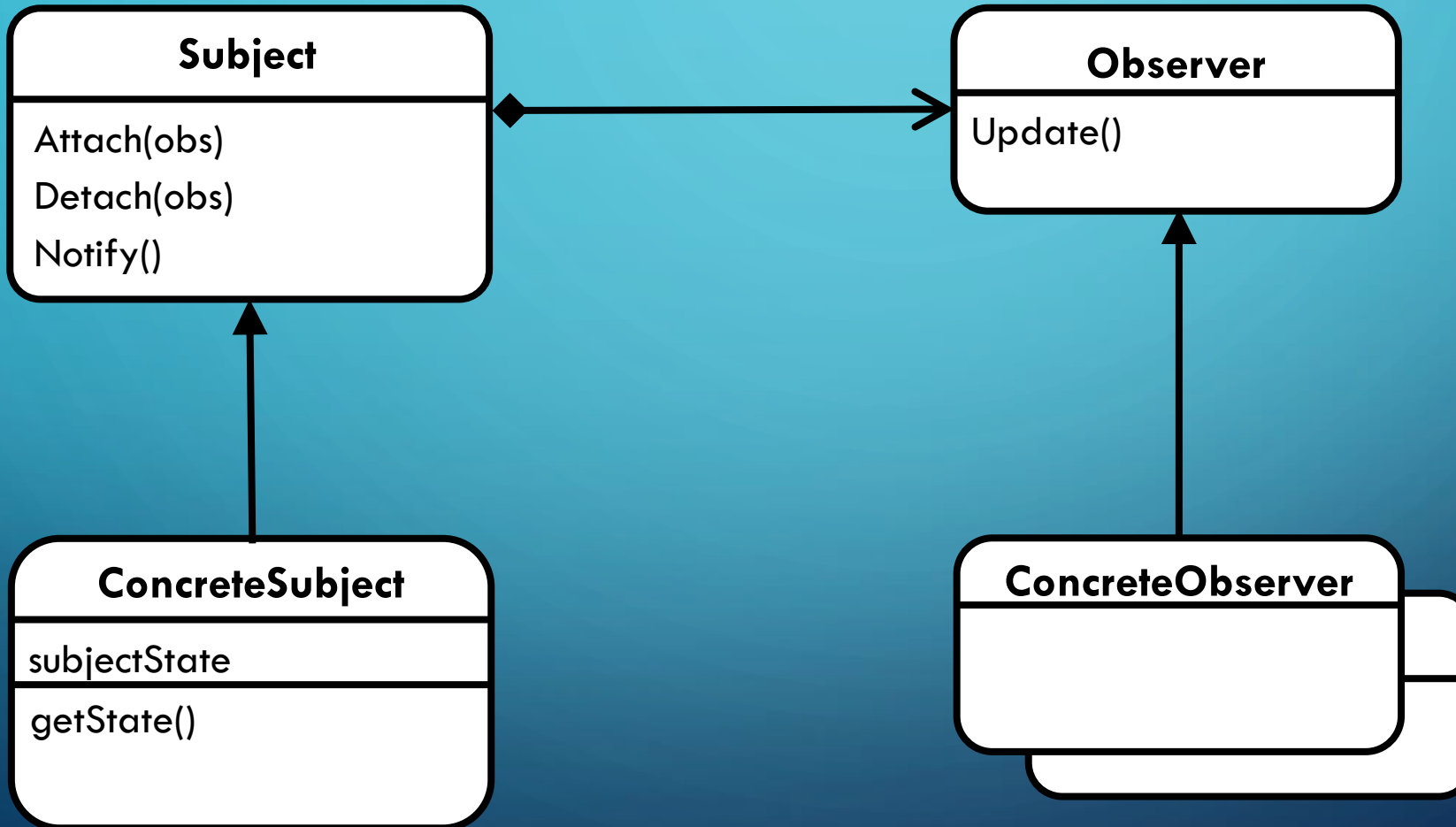
subjectState  
getState()

INTERFACE

## Observer

Update()

## ConcreteObserver



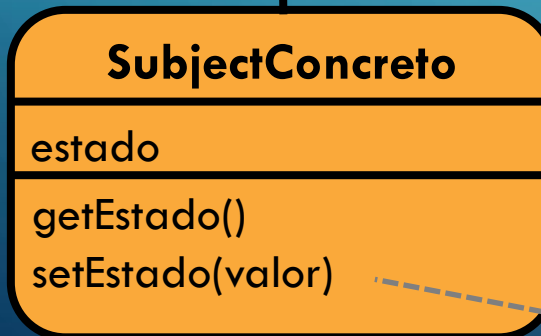
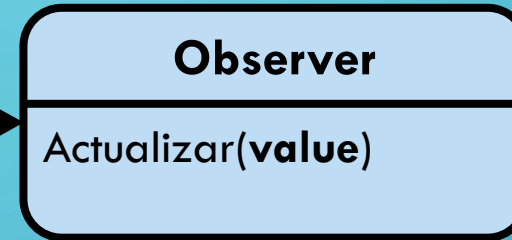
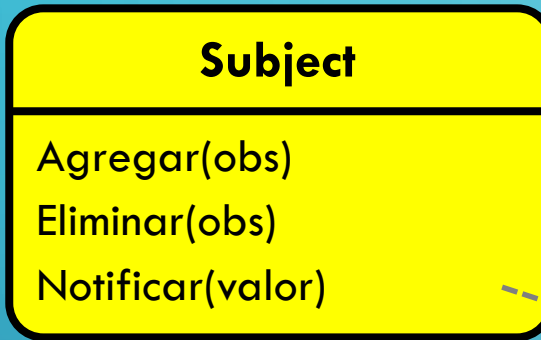


# SOLUCIÓN

PUSH STYLE

CLASE ABSTRACTA

INTERFACE



for ( var obs : observers )  
Obs.Actualizar(valor);

Notificar(this.estado);

estado = value;



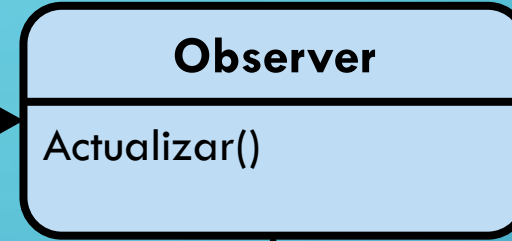
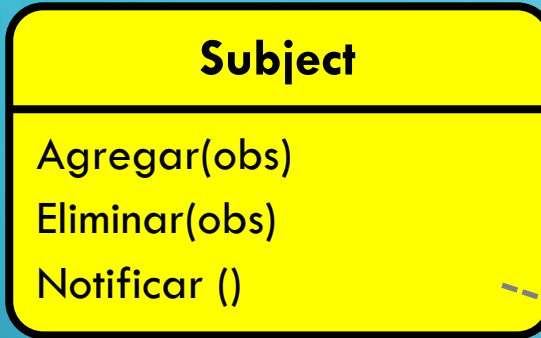


# SOLUCIÓN

PULL STYLE

CLASE ABSTRACTA

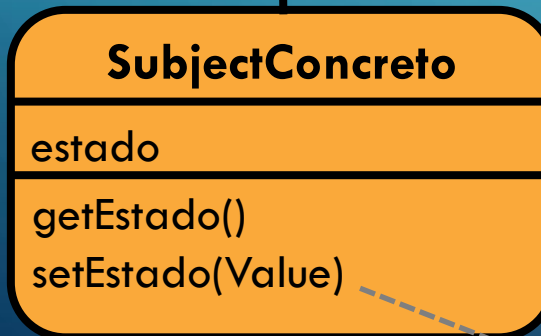
INTERFACE



```

for ( var obs : observers )
  Obs.Actualizar();

```



```

Notificar();

```

```

estado = sujeto.getEstado;

```

```
operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
    operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True
```

```
#selection at the end -add
mirror_ob.select= 1
mirror_ob.select=1
context.scene.objects.active
("Selected" + str(modifier_ob))
mirror_ob.select = 0
= bpy.context.selected_object
data.objects[one.name].select
```

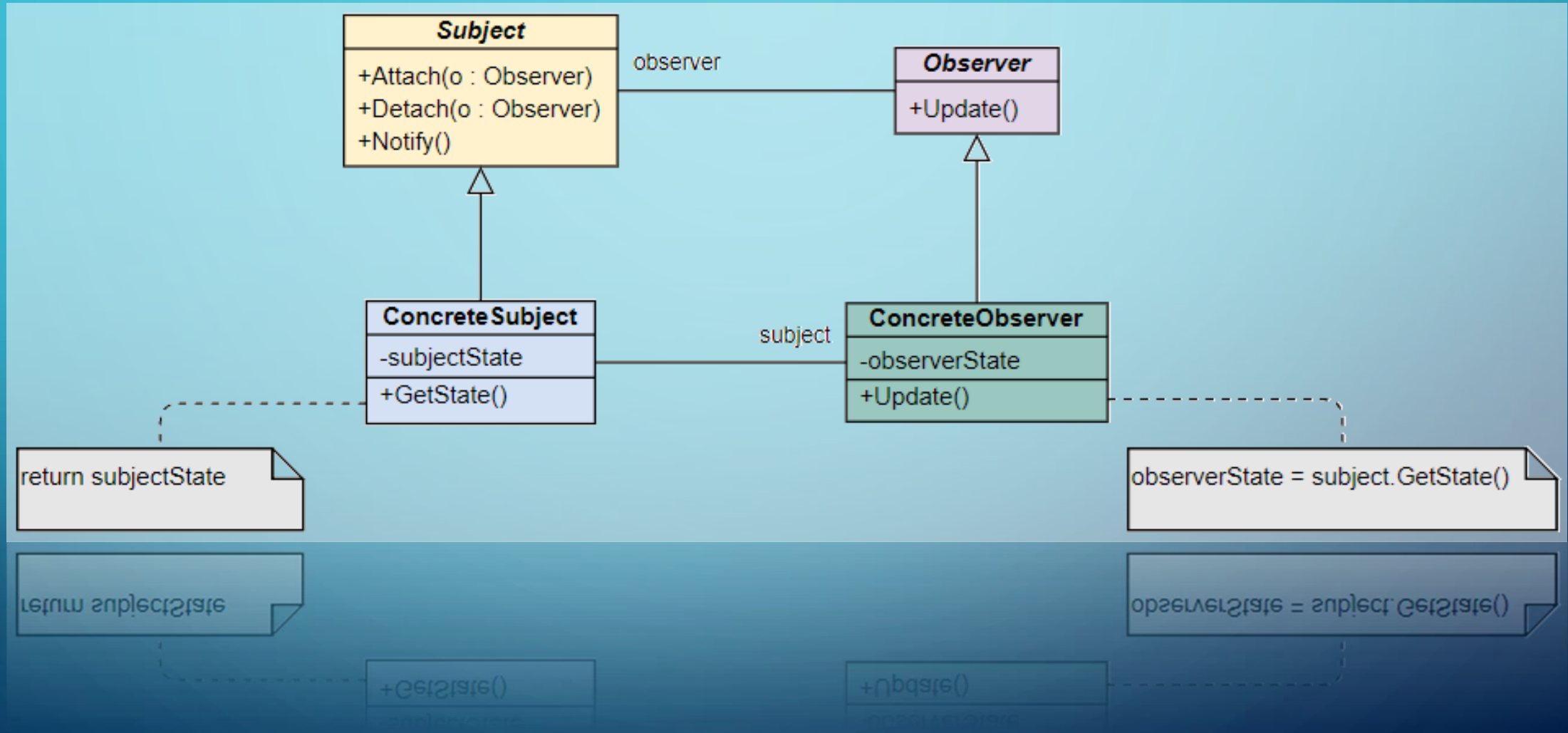


# DEMO

REVISEMOS EN CÓDIGO LA IMPLEMENTACIÓN DE LA SOLUCIÓN



# ESTRUCTURA







# APLICABILIDAD

## CUÁNDO UTILIZAR ESTE PATRÓN?

Utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.

Utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.



# CONSECUENCIAS

## VENTAJAS:

- ✓ Principio de abierto/cerrado. Puedes introducir nuevas clases suscriptoras sin tener que cambiar el código de la notificadora (y viceversa si hay una interfaz notificadora).
- ✓ Puedes establecer relaciones entre objetos durante el tiempo de ejecución.

## DESVENTAJAS:

- ✗ Los suscriptores son notificados en un orden aleatorio.



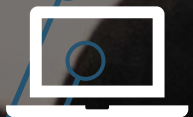
## EJERCICIO



## BOLSA DE VALORES

- Cree una pequeña aplicación de la bolsa de valores para unas 3 empresas importantes del rubro tecnológico.
- Permita que varios sujetos se puedan añadir a la lista de notificados y mostrar los valores que recibieron
- Algunos de estos subscriptores podrían abandonar nuestra bolsa de valores.
- Realice el ejemplo utilizando el patrón Observer





# DEMO

## REVISEMOS EN CÓDIGO LA SOLUCION DE LA PRACTICA



# TAREA

## Aplicación para ver acciones

Hay dos lugares en nuestra aplicación donde necesitamos mostrar las acciones:

- **StatusBar**: muestra las acciones populares
- **StockListView**: muestra la lista completa de acciones

Cuando el precio de una acción cambia, las vistas correspondientes(*StatusBar* y/o *StockListView*) deben actualizarse para reflejar el último precio.

Nuestra aplicación no tiene actualmente la capacidad de comunicar el cambio de precios de las acciones a las vistas correspondientes.

Utilice el patrón observador para resolver este problema.





**Muchas Gracias...**  
**los espero la siguiente clase**





