

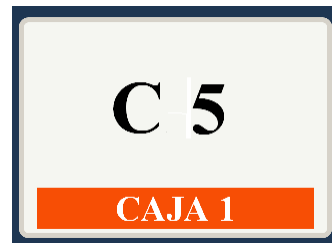
# METODOLOGÍAS DE PROGRAMACIÓN I

---

Patrón de comportamiento *Observer*

# Situación de ejemplo

- Un banco tiene terminales de autogestión para que los clientes saquen un número al momento de llegar al banco a realizar un trámite determinado.
- Cuando un cliente saca un número, en las computadoras de los cajeros debería aparecer la información de que hay un nuevo cliente en el banco (para poder ser llamado por un cajero).

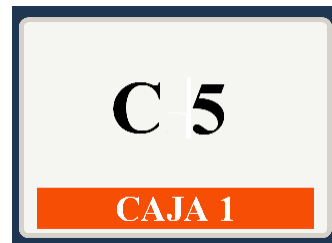


# Situación de ejemplo

Supongamos las clases *Terminal* y *Cajero*.

```
class Cajero
    void llamar()
    // llama al siguiente
    // cliente
```

```
class Terminal
    void solicitar()
    // imprime un ticket
    // con el siguiente
    // número
```

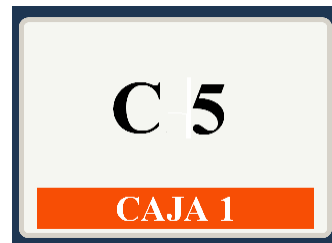


# Situación de ejemplo

Cómo se resuelve el problema cuando hay un único terminal y un único cajero.

```
class Cajero
    void llamar()
    // llama al siguiente
    // cliente
```

```
class Terminal
    void solicitar()
    // imprime un ticket
    // con el siguiente
    // número
```

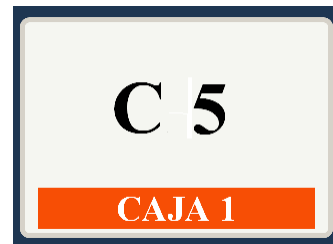


# Problema

¿Qué cambios hay que hacer en el caso de que existan más de un cajero?

```
class Cajero
    void llamar()
    // llama al siguiente
    // cliente
```

```
class Terminal
    void solicitar()
    // imprime un ticket
    // con el siguiente
    // número
```



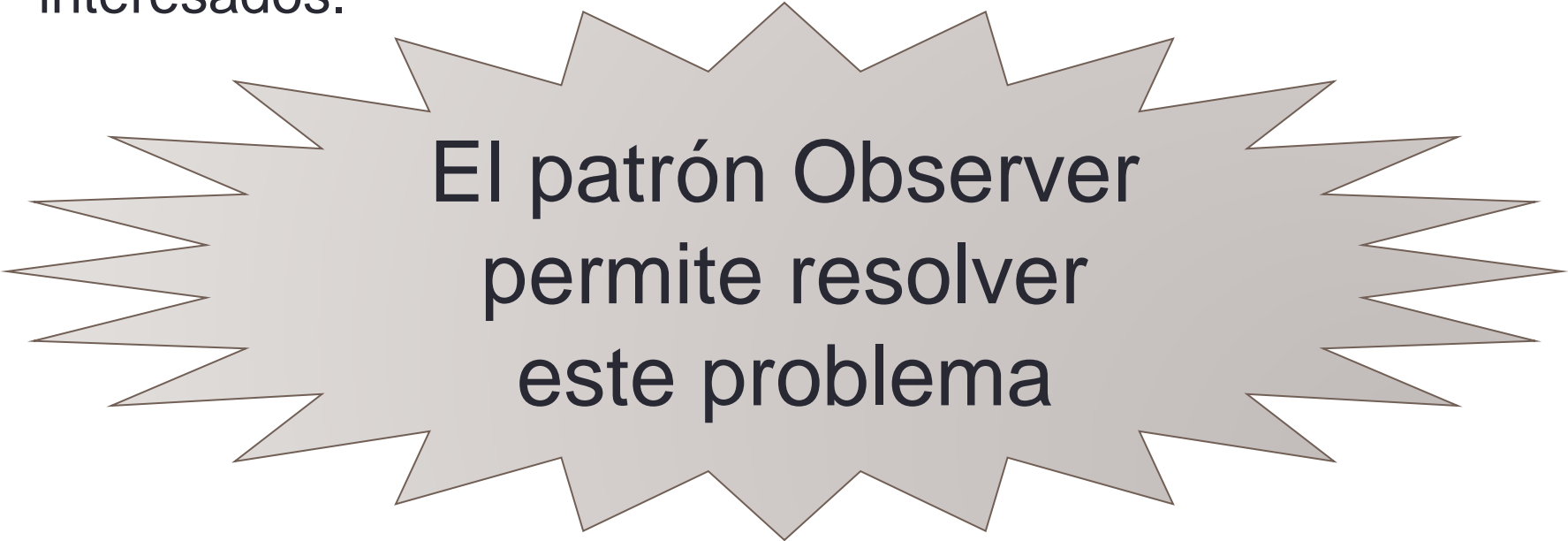
# Problema

¿Podemos hacer que la clase Terminal sea genérica y pueda ser usada en un consultorio médico, en una aseguradora, en el correo, en una carnicería y en cualquier otro lugar que se requiera otorgar números a los clientes?



# Motivación

Sería interesante contar con un mecanismo donde un objeto "anuncie" que un cliente llegó y todos los interesados "escuchen" de la llegada del cliente, sin necesidad que la terminal le avise a cada uno de los interesados.



**El patrón Observer  
permite resolver  
este problema**

# Observer

**Propósito:** Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependan de él.

**Aplicabilidad:** usarlo cuando

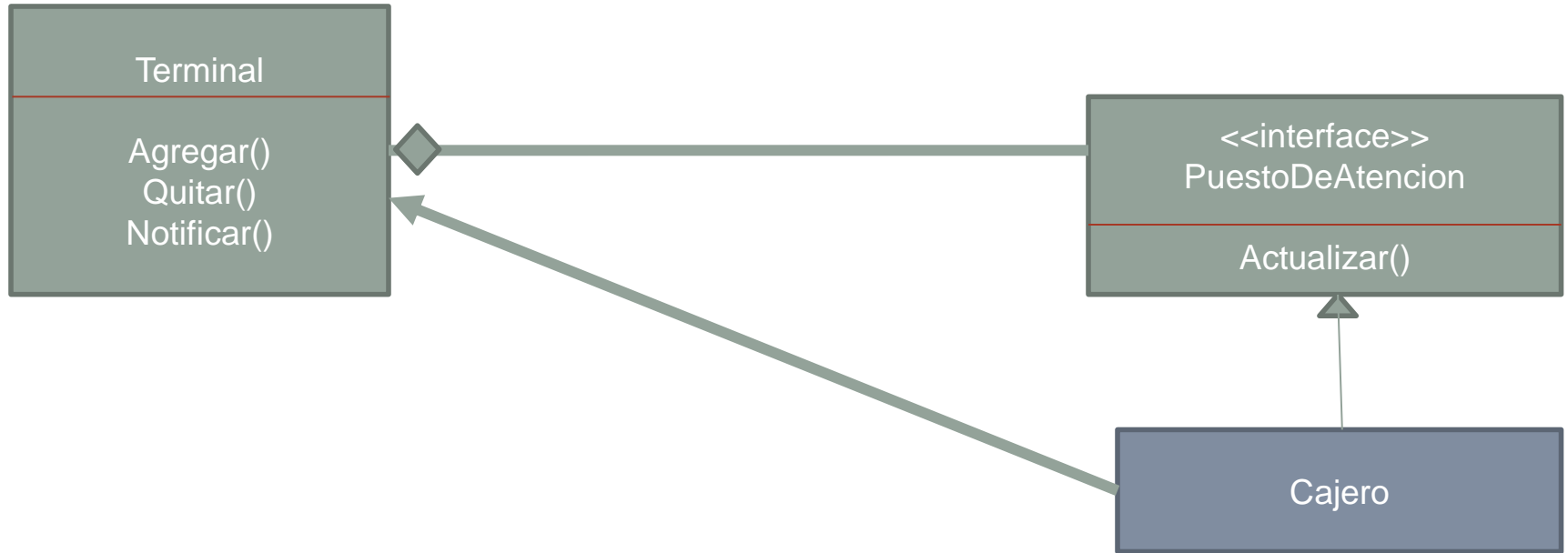
- Una abstracción tiene dos aspectos y uno depende del otro .
- Un cambio en objeto requiere cambiar otros, y no sabemos cuantos objetos necesitan cambiarse.
- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quienes son dichos objetos.



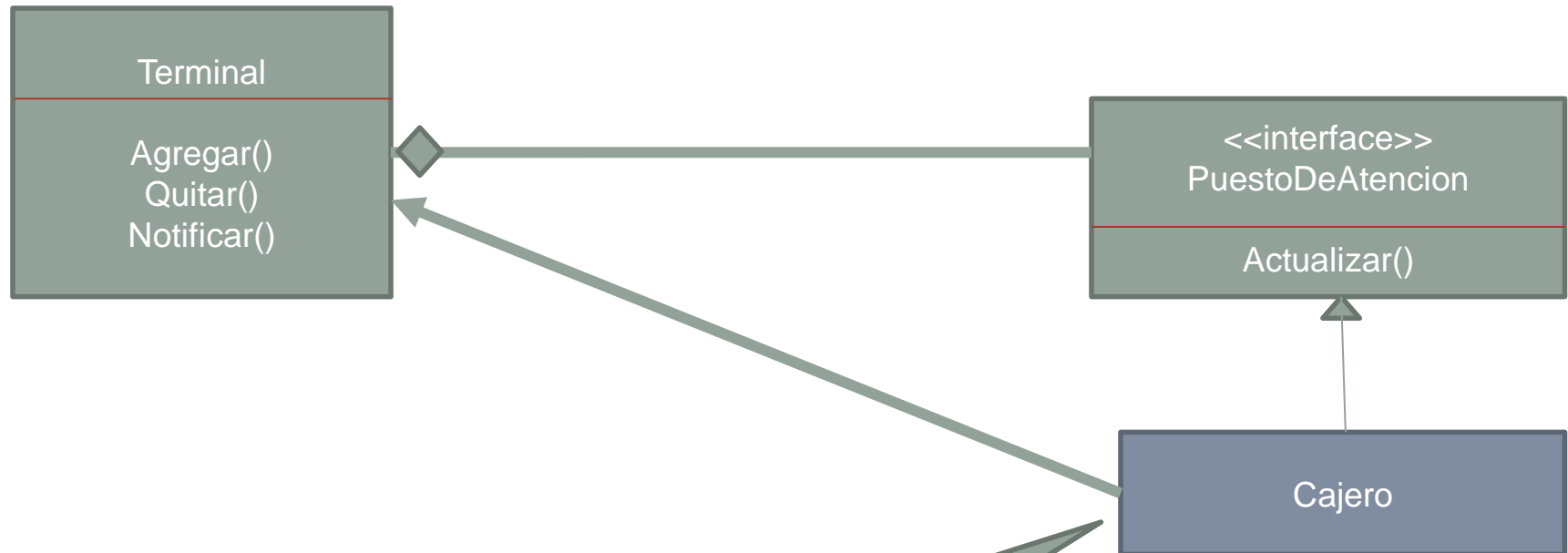
# Observer - Estructura



# Observer - Estructura



# Observer - Estructura



La interface *PuestoDeEstacion* la pueden implementar todos los interesados en "escuchar" a una terminal expendedora de números.

# Observer - Implementación

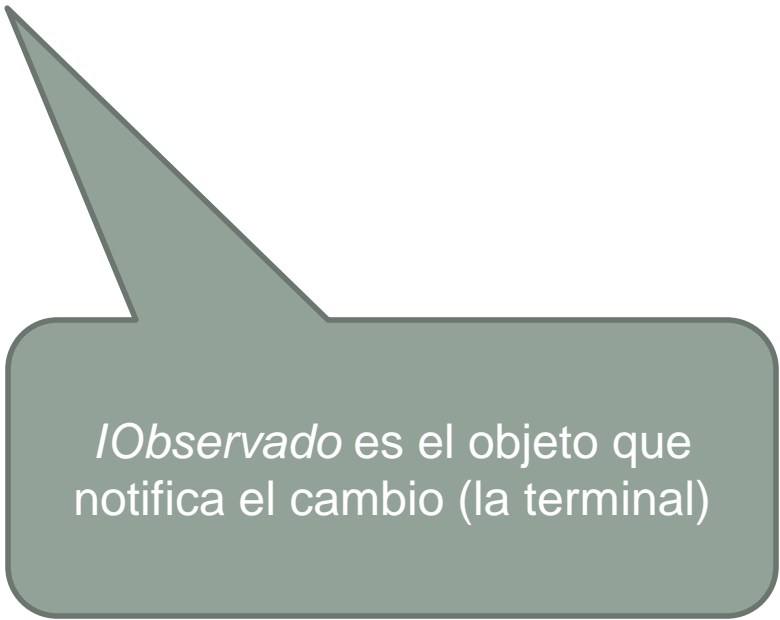
```
interface IObservador
```

```
void actualizar(IObservado o)
```

# Observer - Implementación

```
interface IObservador
```

```
void actualizar(IObservado o)
```

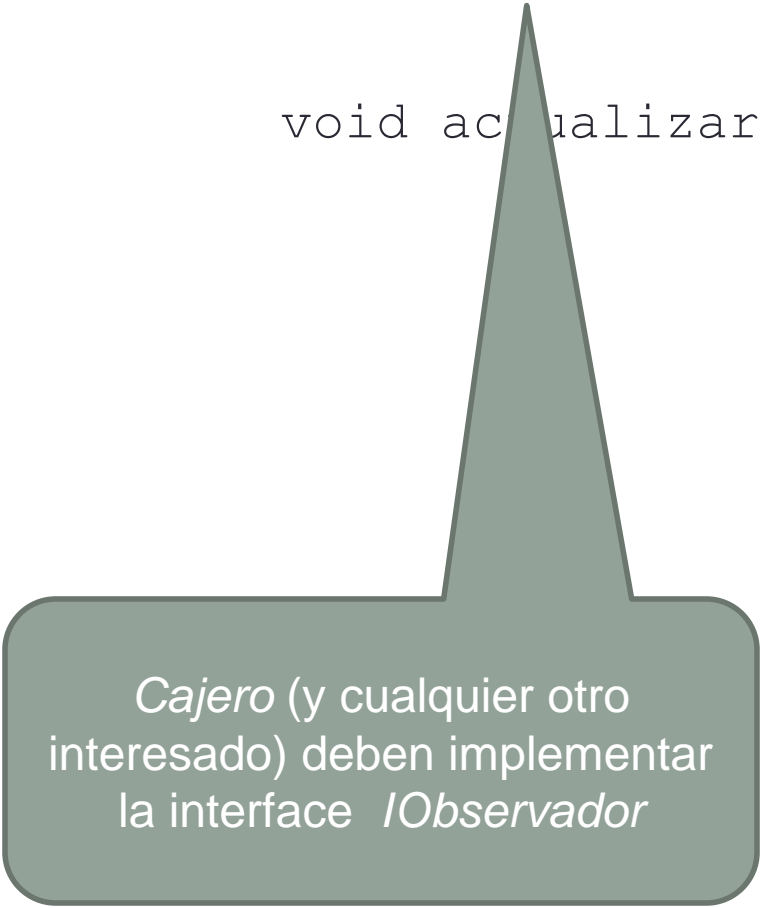


*IObservado* es el objeto que  
notifica el cambio (la terminal)

# Observer - Implementación

```
interface IObservador
```

```
void actualizar(IObservado o)
```



*Cajero (y cualquier otro interesado) deben implementar la interface `IObservador`*

# Observer - Implementación

```
interface IObservado
```

```
    void agregarObservador (IObservador o)
```

```
    void quitarObservador (IObservador o)
```

```
    void notificar()
```

# Observer - Implementación

```
interface IObservado
```

```
void agregarObservador (IObservador o)  
void quitarObservador (IObservador o)  
void notificar ()
```

*IObservado* es una  
interface que permite  
agregar y sacar los  
interesados en los cambios.



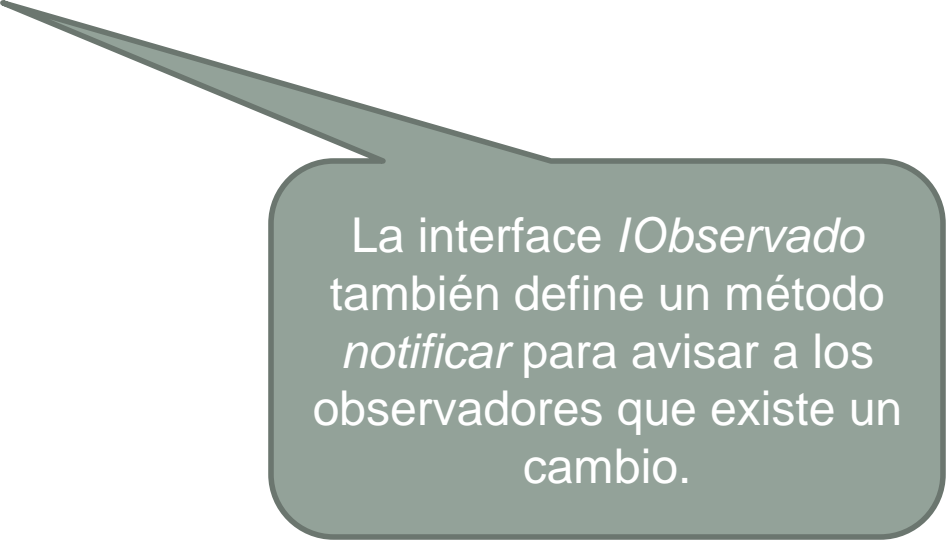
# Observer - Implementación

```
interface IObservado
```

```
void agregarObservador(IObservador o)
```

```
void quitarObservador(IObservador o)
```

```
void notificar()
```



La interface *IObservado* también define un método *notificar* para avisar a los observadores que existe un cambio.

# Observer - Implementación

```
class Terminal : IObservado
    List observadores = new List<IObservador>()

    void agregarObservador(IObservador o)
        observadores.Add(o)
    void quitarObservador(IObservador o)
        observadores.Remove(o)
    void notificar()
        foreach(IObservador o in observadores)
            o.actualizar(self)
```

# Observer - Implementación

```
class Terminal : IObservado
```

```
List observadores = new List<IObservador>()
```

```
void agregarObservador(IObservador o)  
    observadores.Add(o)
```

```
void quitarObservador(IObservador o)  
    observadores.Remove(o)
```

```
void notificar()  
    foreach (IObservador o in observadores)  
        o.actualizar(s)
```

La terminal debería guardar en una colección a los interesados en saber cuando cambia la terminal (llega un cliente)

# Observer - Implementación

```
class Terminal : IObservado
    List observadores = new List<IObservador>()

    void agregarObservador(IObservador o)
        observadores.Add(o)
    void quitarObservador(IObservador o)
        observadores.Remove(o)
    void notificar()
        foreach(IObservador o in observadores)
            o.actualizar(self)
```

También implementar las operaciones para suscribir y desuscribir a los observadores

# Observer - Implementación

```
class Terminal : IObservado
    List observadores = new List<IObs

    void agregarObservador (IObservado
        observadores.Add(o)
    void quitarObservador (IObservador o)
        observadores.Remove(o)

    void notificar()
        foreach (IObservador o in observadores)
            o.actualizar(self)
```

El método notificar es utilizado cada vez que la terminal cambia (llega un cliente) y se necesita avisar a los observadores.

Notar que la terminal (objeto que cambió) se pasa como argumento.

# Observer - Implementación

```
class Terminal : IObservado
    numero = 0
    void sacarNumero()
        numero++
        print("Imprimir ticket con número " + numero)
        self.notificar()

    void notificar()
        foreach(IObservador o in observadores)
            o.actualizar(self)
```

# Observer - Implementación

```
class Cajero : IObservador
    siguiente = 1
    ultimoALlamar = 0

    void llamar()
        if(siguiente <= ultimoALlamar)
            print("Llamando al número " + siguiente)
            siguiente = siguiente + 1

    void actualizar(IObservado o)
        ultimoALlamar = o.ultimoNumeroOtorgado()
```

# Observer - Implementación

```
class Cajero : IObservador
    siguiente = 1
    ultimoALlamar = 0

    void llamar()
        if(siguiente <= ultimoALlamar)
            print("Llamando al numero " + siguiente)
            siguiente = siguiente + 1

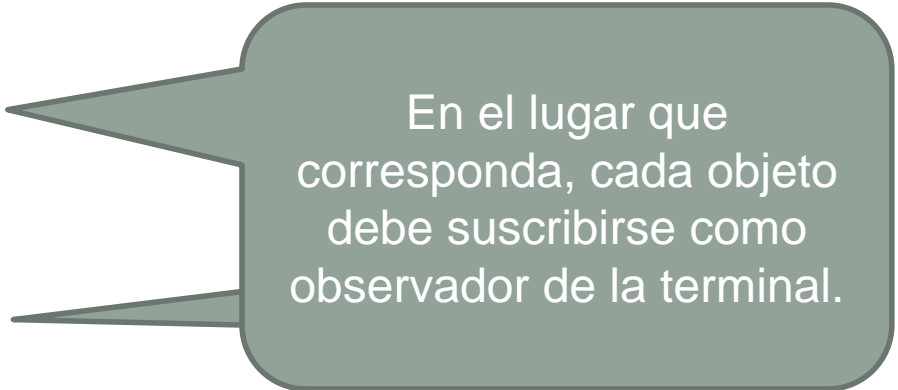
    void actualizar(IObservado o)
        ultimoALlamar = o.ultimoNumeroOtorgado()
```

Cuando el *IObservado* se notifique el *Cajero* podrá saber cual fue el último que se imprimió, para saber hasta qué número llamar



# Observer - Implementación

```
void metodo()  
    t = new Terminal()  
  
    c1 = new Cajero()  
    t.agregarObservador(c1)  
  
    c2 = new Cajero()  
    t.agregarObservador(c2)
```



En el lugar que corresponda, cada objeto debe suscribirse como observador de la terminal.

# Observer – Ventajas

- Permite modificar los sujetos y los observadores de forma independiente. Permite añadir observadores sin modificar el sujeto.
- Acoplamiento abstracto entre sujeto y observador. El sujeto no conoce ninguna clase concreta de observador.
- Al sujeto no le interesa saber cuantos observadores hay.