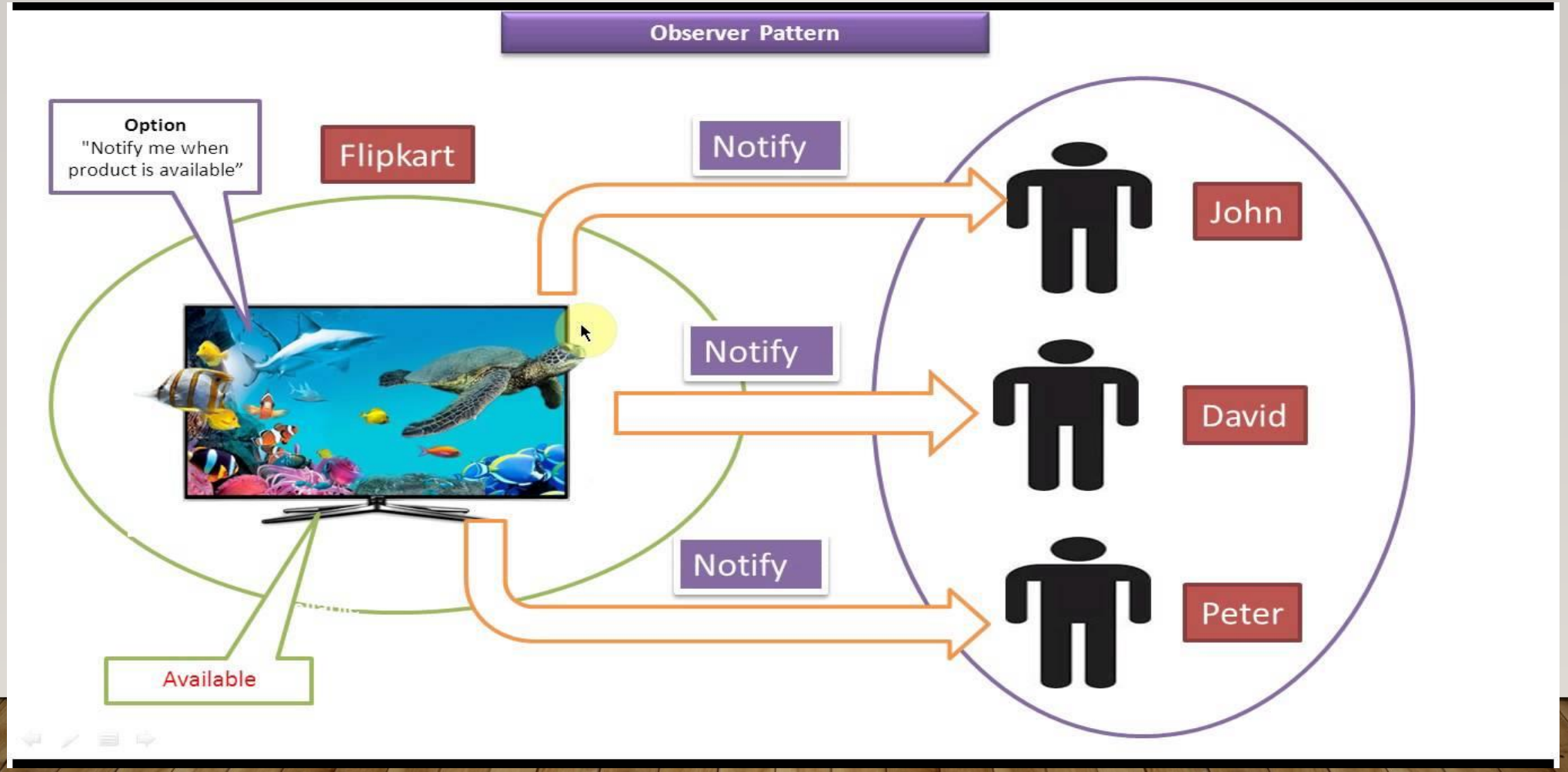
OBSERVER.

DESIGN PATTERN.



NOMBRE Y CLASIFICACIÓN

- **Observer. (Observador).**

- **Behavioral Patterns (Diseños basados en comportamiento).**

OTROS NOMBRES

- **Dependents (Dependientes).**

- **Publish-Suscribe (Publicar-suscribir).**

MOTIVACIÓN

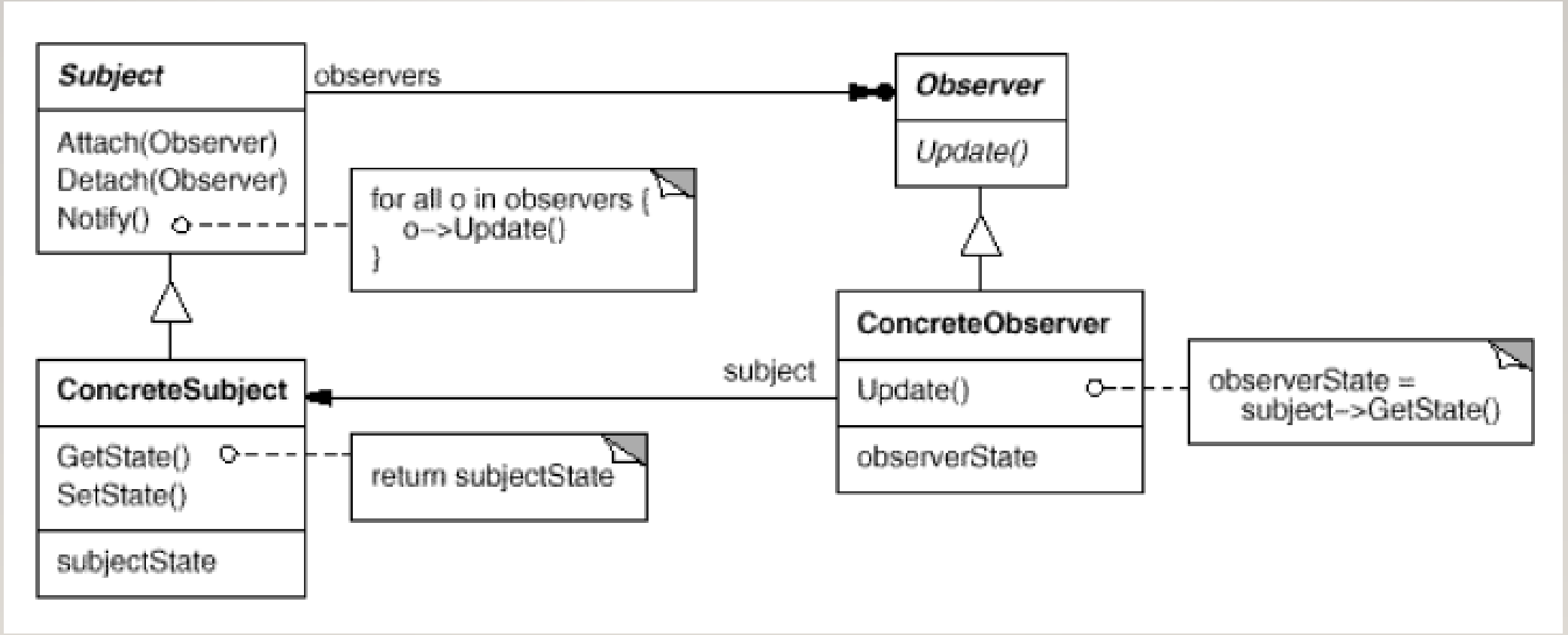
- **Situación: Un objeto 'A' cambia de estado y, por determinada razón, otro objeto 'B' necesita conocer ese cambio.**

- **Dos objetos relacionados necesitan mantener comunicación constante y, para mantener la reusabilidad del código, entonces se diseña un método en el que se crean dos tipos de objeto, un 'sujeto' y un 'observador'.**

APLICACIÓN

- **Dos aspectos de una misma abstracción necesitan coordinarse y a la vez estar encapsuladas para mantener reusabilidad.**
- **Un cambio en un objeto requiere o provoca el cambio en otros.**
- **Notificar a otros objetos sin necesidad de conocer su estructura o naturaleza.**

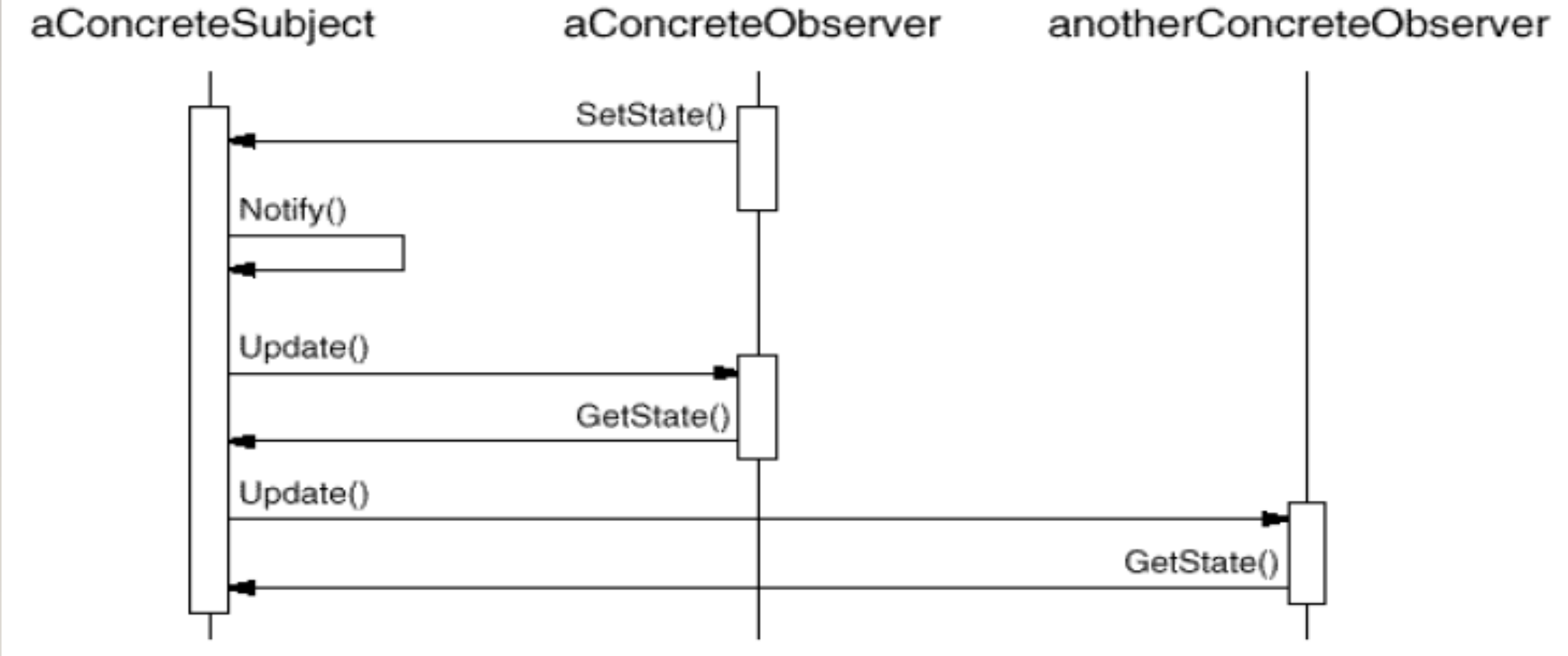
ESTRUCTURA


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState

PARTICIPANTES

- Subject (Sujeto)
- Observer (Observador).
- ConcreteSubject.
- ConcreteObserver.

COLABORACIONES



CONSECUENCIAS

- Pros:
- Pueden implementarse a distinto nivel de abstracción debido a la independencia del sujeto y que éste no necesita realmente especificar a qué objeto le envía la notificación.
- La comunicación puede ser generalizada, lo que significa que objetos de distinta naturaleza pueden recibir el mismo tipo de mensaje, sin que haya problema alguno.

CONSECUENCIAS

- Contras:
  Actualizaciones no deseadas. Cuando un sujeto no recibe una respuesta de su dependencia objeto (observador), puede causar una serie de notificaciones o actualizaciones que se conviertan en información no deseada, debido a que los observadores no se comunican entre sí, generando un posible desfase.

IMPLEMENTACIÓN

- Alto coste de implementación cuando hay muchos sujetos y pocos observadores (mucha creación de objetos observadores = pesadez innecesaria)
  - Solución: uso de tablas Hash par aplicar sobre ella iteraciones cada determinado tiempo para seguir el mapeo sujeto-observador.
- Multidependencia: cuando un observador necesita de más de un sujeto, entonces se necesita también la coordinación de los sujetos además de que el objeto ya necesita conocer cuál sujeto le está enviando la información en ese momento.

IMPLEMENTACIÓN

- ¿Quién lanza la actualización? Para evitar este error, lo mejor sería que el cliente (objeto) sea quien lance la actualización después de recibir el mensaje del sujeto, pero esto podría generar errores de cualquier tipo debido a la sobrecarga de responsabilidad al cliente.
- Mantener referencias a sujetos que han dejado de existir. Los observadores pueden presentar referencias a sujetos que ya no existen y borrar a dichos sujetos no es opción debido a que por lo común este tipo de objetos es referenciado por otros. Solución: avisar al observador para que haga un reset de sus datos.
- Verificar que el sujeto es consistente.

USOS CONOCIDOS

- **Redes sociales y servicios web en general (Servicios de mensajes SMS, correos automatizados, etc.)**

- **Sistemas de captura de información física o de conteo.**

PATRONES RELACIONADOS

- **Mediador**
- **Singleton.**