Observer Pattern

Tomás Felipe Melli

July 15, 2025

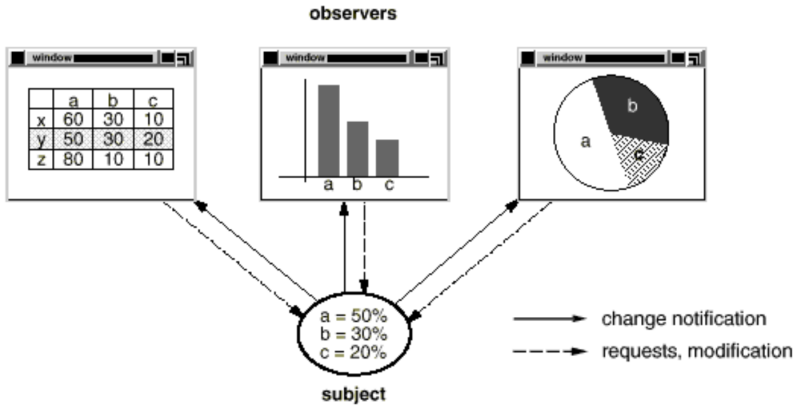
Índice

1 Intent

Definir una dependencia uno-a-muchos entre objetos, de modo que cuando uno cambie su estado, todos sus dependientes sean notificados y actualizados automáticamente.

2 Motivation

Cuando un objeto cambia, otros pueden necesitar reaccionar ante ese cambio (por ejemplo, relojes analógicos o digitales que dependen de un temporizador). En lugar de acoplar directamente estos objetos, **Observer** desacopla al sujeto de sus observadores, permitiendo que los cambios se propaguen sin dependencia directa entre ellos.

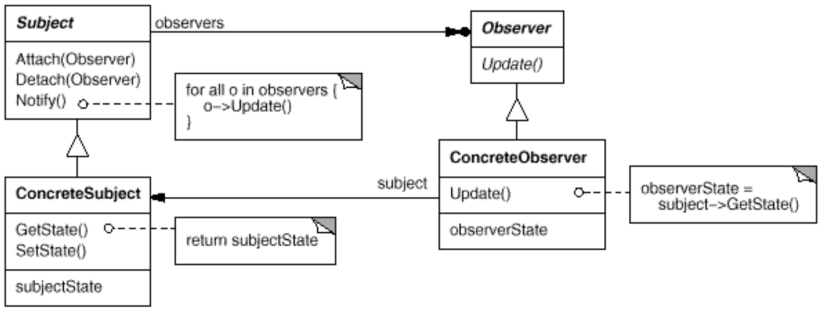


3 Applicability

Usar el patrón Observer cuando:

- Un cambio en un objeto requiere cambios en otros, pero no se sabe cuántos objetos necesitarán reaccionar.
- Se desea desacoplar el objeto que emite cambios de los objetos que reaccionan ante ellos.
- Se necesita un mecanismo de suscripción-notificación entre objetos.

4 Structure



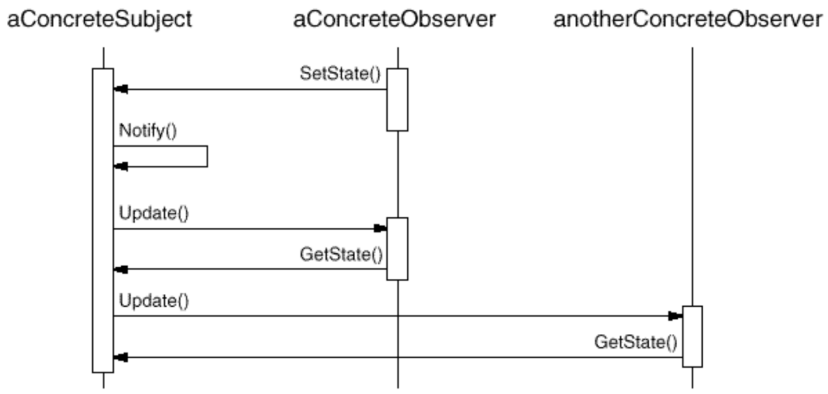
5 Participants

- **Subject**:
  - Mantiene una lista de observadores.
  - Proporciona métodos para adjuntar, eliminar y notificar observadores.
- **Observer**:
  - Define una interfaz para recibir actualizaciones del `Subject`.

- **ConcreteSubject**:
  - Guarda el estado de interés para los observadores.
  - Notifica a los observadores cuando cambia su estado.
- **ConcreteObserver**:
  - Se suscribe al sujeto.
  - Actualiza su estado para mantenerse consistente con el sujeto.

6 Collaborations

- El `ConcreteSubject` notifica a todos los `Observers` cuando su estado cambia.
- Cada `ConcreteObserver` implementa el método `Update`, que es invocado por el sujeto.



7 Consequences

- **Acoplamiento reducido**: El sujeto y los observadores están desacoplados, lo que facilita su reutilización independiente.
- **Soporte para difusión de cambios**: Se puede tener cualquier cantidad de observadores que reaccionan automáticamente a los cambios.
- **Dependencias implícitas**: Es posible que los efectos de un cambio no sean evidentes para quien lo genera.
- **Orden de notificación no garantizado**: Puede ser difícil controlar el orden en que se notifican los observadores.

8 Implementation

Varios aspectos importantes del mecanismo de dependencia se consideran en esta sección:

1. **Mapeo de sujetos a observadores:** La forma más simple de que un `Subject` rastree sus observadores es almacenarlos directamente. Sin embargo, esto puede ser costoso en términos de memoria si hay muchos sujetos con pocos observadores. Una solución alternativa es usar una estructura asociativa (como una `hash table`) para mapear sujetos a observadores, evitando el overhead en sujetos sin observadores, a costa de mayor tiempo de acceso.
2. **Observar múltiples sujetos:** Un `Observer` puede depender de más de un `Subject`. Para ello, la interfaz `Update` debe extenderse para identificar qué sujeto envía la notificación. Esto se resuelve pasando el `Subject` como parámetro en `Update`.
3. **¿Quién dispara la actualización?** Hay dos enfoques:
   - **Notificación automática:** Las operaciones del `Subject` llaman a `Notify` al cambiar el estado. Ventaja: el cliente no necesita recordar invocar `Notify`. Desventaja: múltiples cambios consecutivos generan múltiples actualizaciones innecesarias.

- **Notificación manual:** El cliente es responsable de llamar a `Notify` luego de hacer los cambios necesarios. Ventaja: evita notificaciones intermedias. Desventaja: mayor responsabilidad del cliente, riesgo de olvidar notificar.

4. **Referencias colgantes a sujetos eliminados:** Si se elimina un `Subject`, sus `Observers` podrían quedar con referencias inválidas. Una solución es que el `Subject` notifique a sus observadores durante su destrucción, permitiéndoles limpiar sus referencias. Eliminar directamente los observadores no es viable, ya que podrían observar otros sujetos.

5. **Estado consistente antes de notificar:** Es fundamental que el estado del `Subject` esté completamente actualizado antes de llamar a `Notify`, ya que los observadores podrían consultarlo. Este principio puede violarse fácilmente si una operación de subclase llama a un método heredado que notifica antes de actualizar el estado local. Para evitar esto, se recomienda usar métodos plantilla (`Template Method`) donde `Notify` se invoca al final del algoritmo.

6. **Modelo push vs pull:**

   - **Push:** El sujeto envía a los observadores información detallada del cambio, se necesite o no. Puede acoplar demasiado el sujeto a los observadores.
   - **Pull:** El sujeto solo notifica el cambio; los observadores deben consultar los detalles. Favorece el desacoplamiento, pero puede ser menos eficiente.

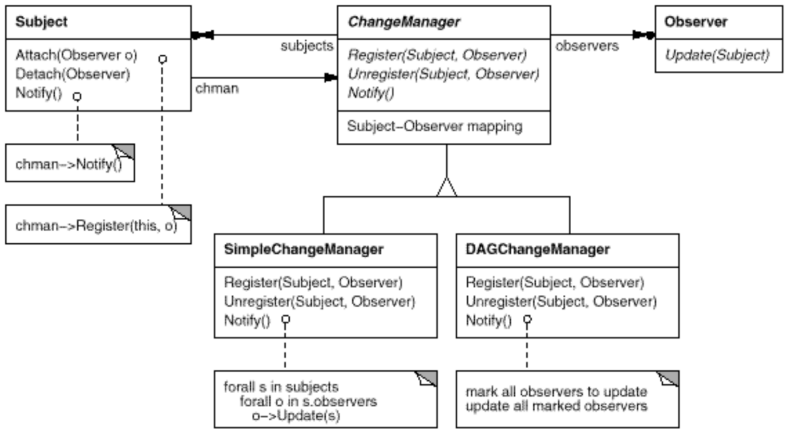
7. **Especificar intereses explícitos:** La eficiencia puede mejorar si los observadores se registran para eventos específicos usando "aspectos" (`Aspect`). En este enfoque, `Subject::Attach` recibe también el aspecto de interés. Luego, en la notificación, el `Subject` envía el aspecto modificado como parámetro:

```
void Subject::Attach(Observer*, Aspect& interest);
void Observer::Update(Subject*, Aspect& interest);
```

8. **Encapsular dependencias complejas con ChangeManager:** Cuando las dependencias son complejas, puede usarse un objeto adicional: `ChangeManager`. Sus responsabilidades son:

   - Mapear sujetos a observadores y mantener ese mapeo.
   - Definir una estrategia de actualización.
   - Coordinar la actualización de observadores.

   Hay versiones simples (`SimpleChangeManager`) que notifican a todos los observadores siempre, y versiones más sofisticadas como `DAGChangeManager`, que manejan grafos acíclicos dirigidos para evitar actualizaciones redundantes cuando un observador observa múltiples sujetos.



9 Sample Code

subsection*Interfaces base

```
// Declaraci n adelantada
class Subject;

// Interfaz Observer
class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
```

```
protected:
    Observer();
};
```

Esta implementación permite que un `Observer` observe múltiples `Subject`, ya que el parámetro recibido en `Update` indica cuál sujeto emitió la notificación.

```
// Interfaz Subject
class Subject {
public:
    virtual ~Subject();
    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*>* _observers;
};

void Subject::Attach(Observer* o) {
    _observers->Append(o);
}

void Subject::Detach(Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify() {
    ListIterator<Observer*> i(_observers);
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

Clase concreta ClockTimer

`ClockTimer` es un `Subject` concreto que mantiene el tiempo actual y notifica a sus observadores cada segundo:

```
class ClockTimer : public Subject {
public:
    ClockTimer();
    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();
    void Tick();
};

void ClockTimer::Tick() {
    // actualizar el estado interno del reloj
    // ...
    Notify();
}
```

Clase observadora DigitalClock

`DigitalClock` hereda de `Observer` y de una clase base `Widget`, y muestra la hora en formato digital:

```
class DigitalClock : public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void Update(Subject*);
    virtual void Draw(); // m todo gr fico para redibujar el reloj
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock(ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock() {
```

```
    _subject->Detach(this);
}

void DigitalClock::Update(Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw() {
    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc...
    // c digo para dibujar el reloj digital
}
```

Clase observadora AnalogClock

Una implementación similar puede hacerse para un reloj analógico:

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

Uso del patrón

La siguiente secuencia crea dos relojes (digital y analógico) sincronizados con el mismo `ClockTimer`:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Cada vez que el temporizador llama a `Tick`, ambos relojes son notificados y actualizan su visualización.