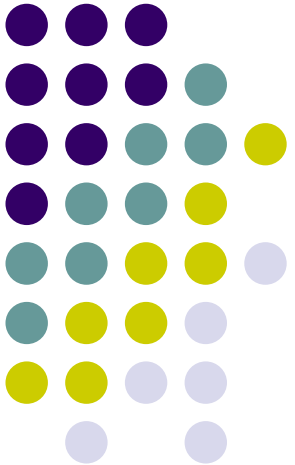
Patrones de Diseño

Patrón de comportamiento *Observer*



Observer

Propósito

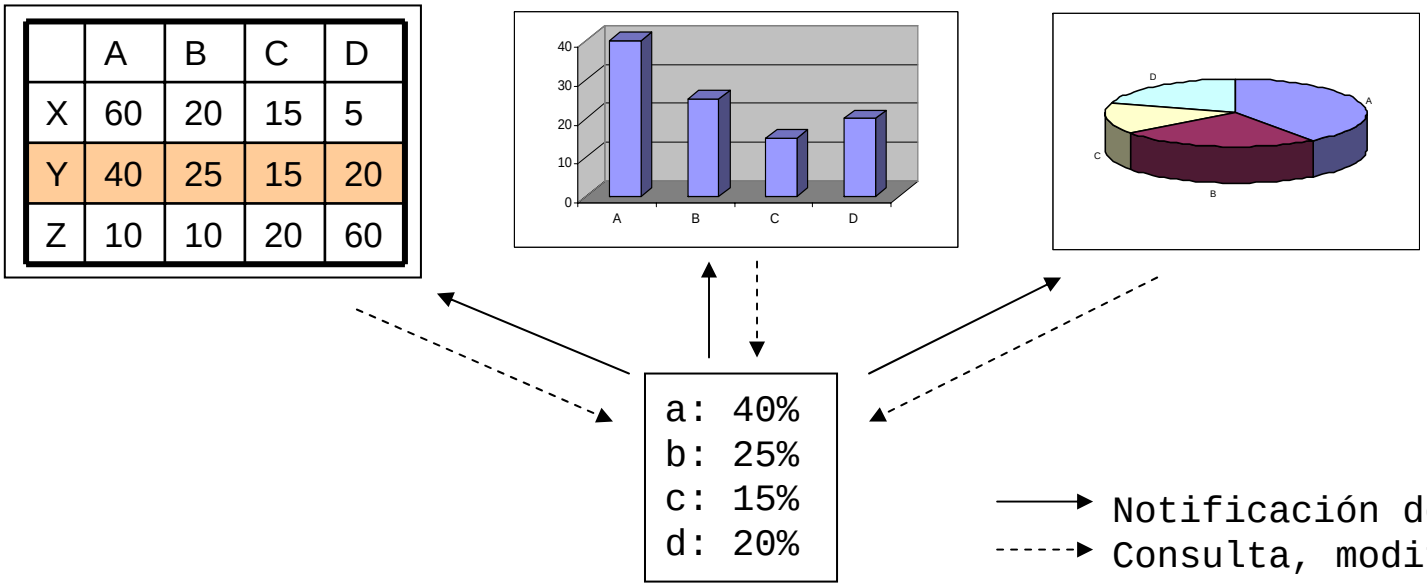
- Define una dependencia de uno-a-muchos entre objetos de forma que, cuando un objeto cambia de estado, se notifica a los objetos dependientes para que se actualicen automáticamente.

- También conocido como *dependents*, *publish-subscribe*

Observer
Motivación

- Mantener la consistencia entre objetos relacionados, sin aumentar el acoplamiento entre clases

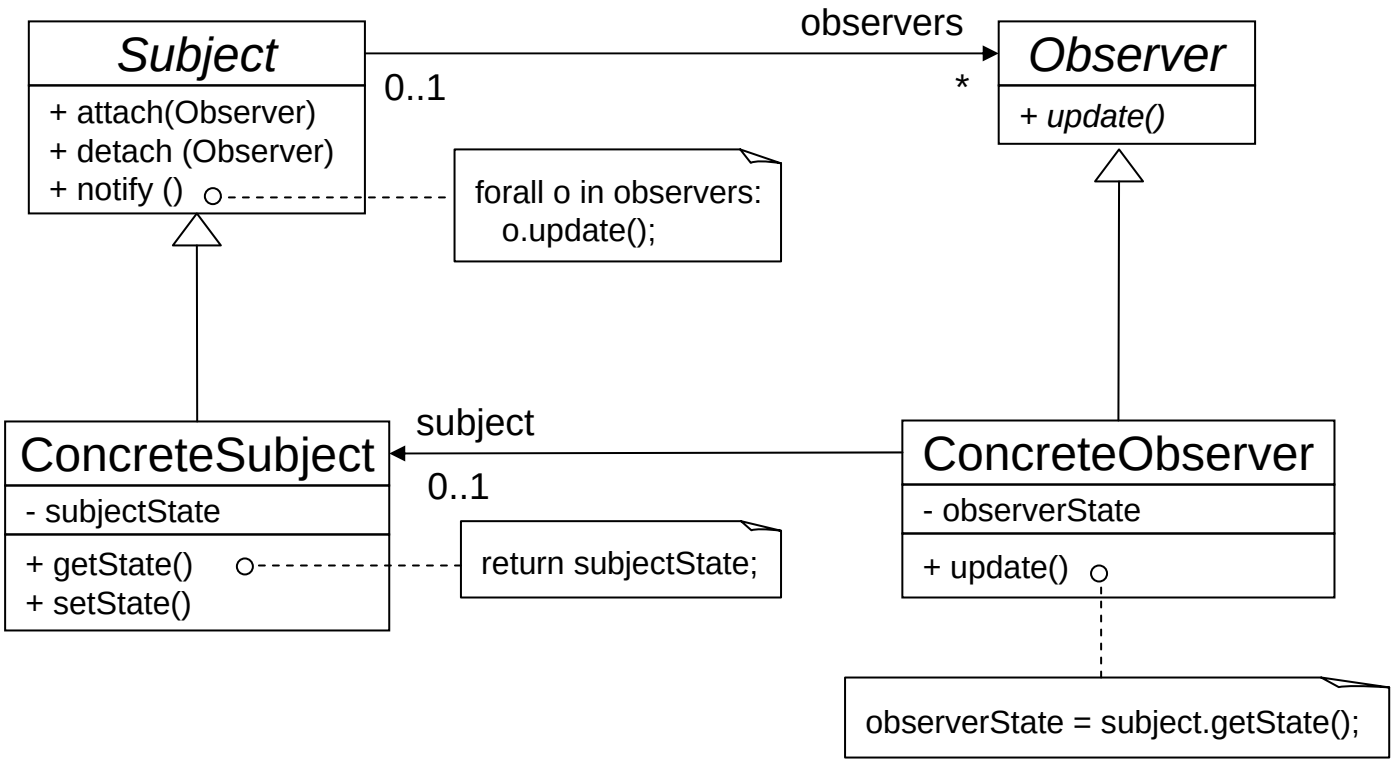
- Ej: separación de la capa de presentación en una interfaz de usuario de los datos de aplicación subyacentes



Observer
Aplicabilidad

- Usa el patrón *Observer*:
  - Cuando una abstracción tiene dos aspectos, y uno depende del otro. Encapsular los aspectos en objetos distintos permite cambiarlos y reutilizarlos.
  - Cuando cambiar un objeto implica cambiar otros, pero no sabemos exactamente cuántos hay que cambiar
  - Cuando un objeto debe ser capaz de notificar algo a otros sin hacer suposiciones sobre quiénes son dichos objetos. Esto es, cuando se quiere bajo acoplamiento.

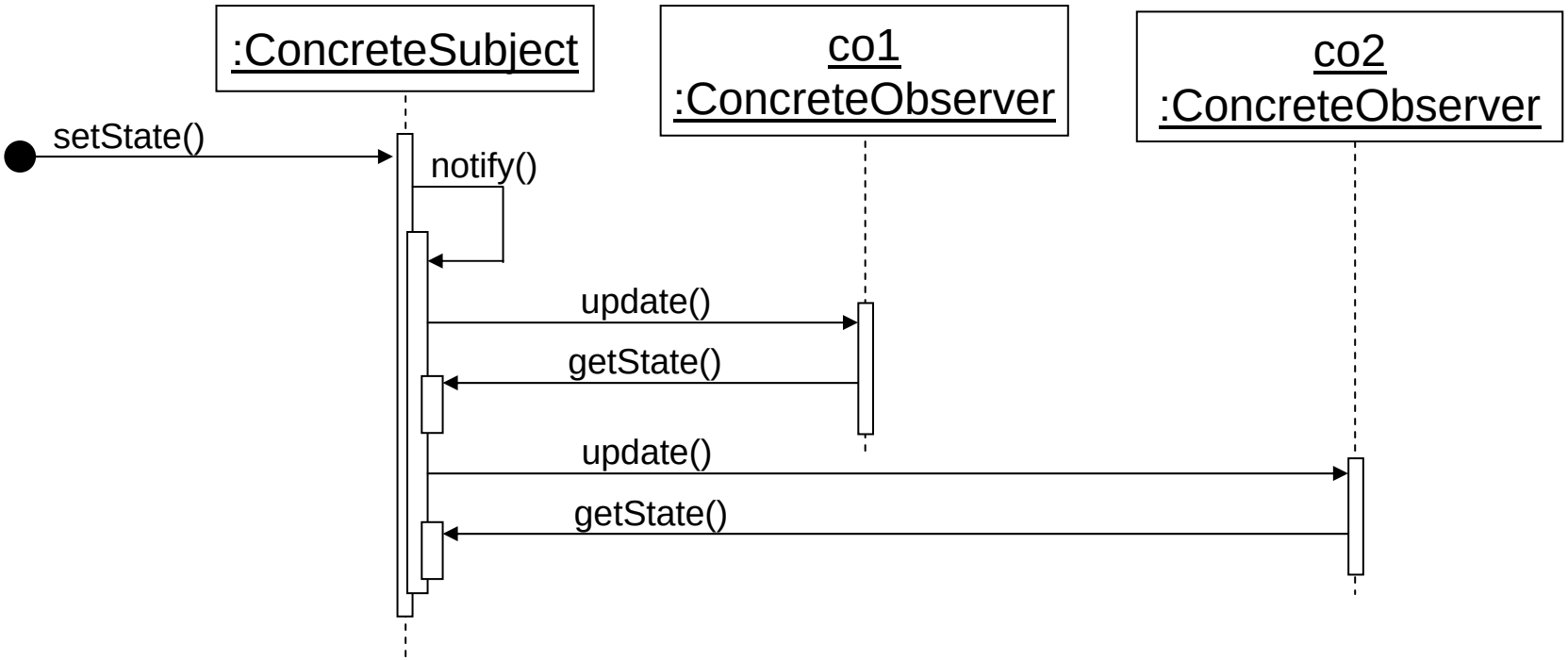
Observer
Estructura



Observer
Participantes

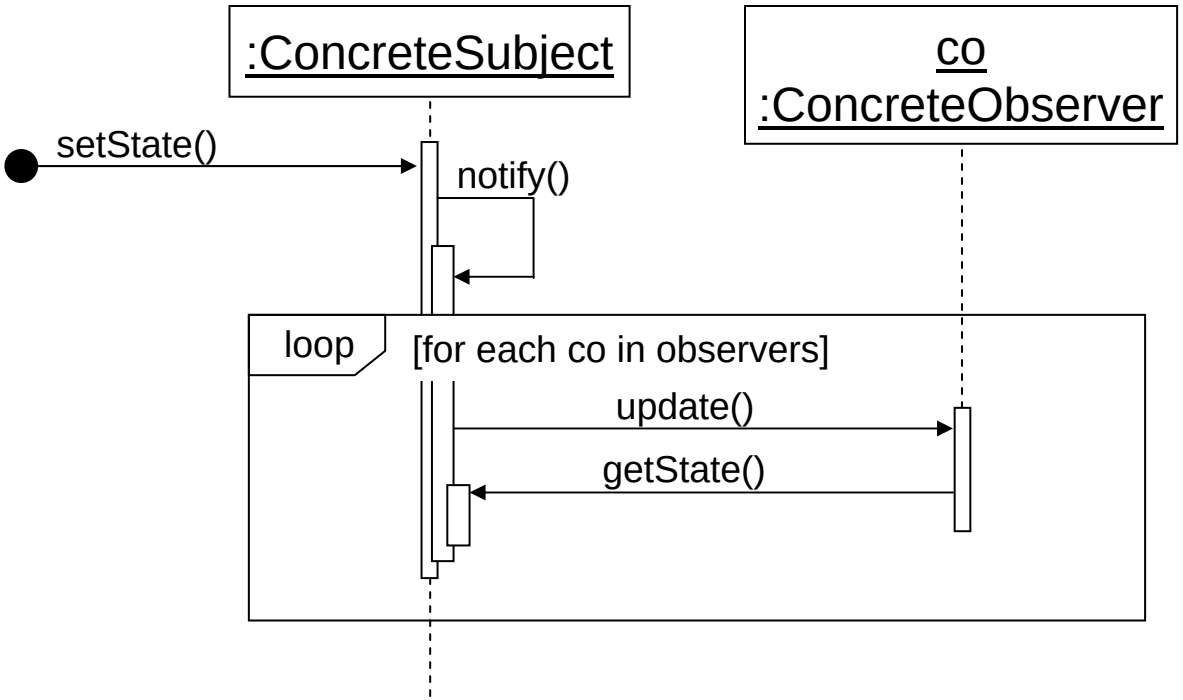
- **Subject**:
  - conoce a sus observadores, que pueden ser un número arbitrario
  - proporciona una interfaz para añadir y quitar objetos observadores
- **Observer**:
  - define la interfaz de los objetos a los que se deben notificar cambios en un sujeto
- **ConcreteSubject***:*
  - almacena el estado de interés para sus observadores
  - envía notificaciones a sus observadores cuando su estado cambia
- **ConcreteObserver**:
  - mantiene una referencia a un *ConcreteSubject*
  - almacena el estado del sujeto que le resulta de interés
  - implementa la interfaz de *Observer* para mantener su estado consistente con el del sujeto

Observer
Colaboraciones



(sujeto con dos observadores)

Observer
Colaboraciones



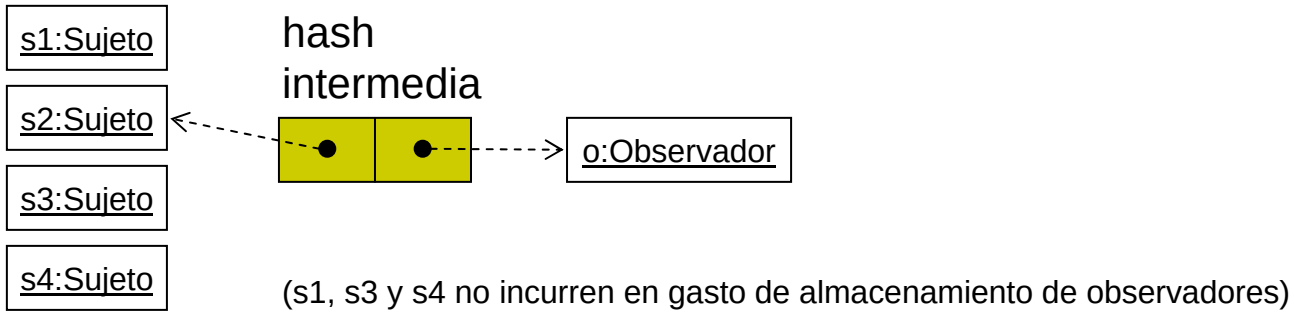
(sujeto con un número arbitrario de observadores)

Observer
Consecuencias

- Permite modificar sujetos y observadores de manera independiente
- Permite reutilizar un sujeto sin reutilizar sus observadores, y viceversa
- Permite añadir observadores sin tener que cambiar el sujeto ni los demás observadores
- Acoplamiento abstracto entre el sujeto y el observador. El sujeto no sabe la clase concreta de sus observadores (acoplamiento mínimo).
- Soporte para *broadcast*. El sujeto envía la notificación a todos los observadores suscritos. Se pueden añadir/quitar observadores.
- Actualizaciones inesperadas. Una operación en el sujeto puede desencadenar una cascada de cambios en sus observadores. El protocolo no ofrece detalles sobre lo que ha cambiado.

Observer
Implementación

- Correspondencia entre sujetos y observadores
  - Usualmente, el sujeto guarda una referencia a sus observadores
  - Costoso si hay muchos sujetos y pocos observadores. En ese caso:
    - usar tabla hash: menor coste en espacio, pero mayor coste en tiempo



- Observar más de un sujeto (ej. hoja de cálculo con 2 fuentes de datos)
  - Extender la interfaz de actualización para que el observador sepa qué sujeto cambió de estado (por ej. pasar el sujeto en la llamada a *update*).

Observer
Implementación

- ¿Quién dispara la actualización llamando a *notify*?
  - El sujeto desde aquellos métodos que cambian su estado
    - ventaja: las clases cliente no tienen que hacer nada
    - inconveniente: no es óptimo si hay varios cambios de estado seguidos
  - Las clases cliente
    - ventaja: se puede optimizar llamando a *notify* tras varios cambios
    - inconveniente: los clientes tienen la responsabilidad de llamar a *notify*

- Referencias perdidas a sujetos que se han eliminado
  - Se puede evitar notificando la eliminación del sujeto a sus observadores
  - En general, eliminar los observadores del sujeto eliminado no es una opción

Observer
Implementación

- Asegurarse de la consistencia del sujeto antes de una notificación
  - Cuidado con las operaciones heredadas!

```
class ClaseSujetoBase {
   void operacion (int valor) {
      _miVar += valor;        // actualiza el estado
      notify();               // dispara la notificación
   }
}
class MiSujeto extends ClaseSujetoBase {
   void operacion (int valor) {
      super.operacion(valor); // dispara la notificación
      _miVar += valor;        // actualiza el estado (tarde)
   }
}
```

  - Soluciones:
    - documentar los métodos que envían notificaciones
    - patrón de diseño *Template Method*

Observer
Implementación

- Asegurarse de la consistencia del sujeto antes de una notificación
  - Soluciones:
    - documentar los métodos que envían notificaciones
    - patrón de diseño *Template Method*

los clientes llaman a `operacionNotify` en vez de a `operacion`

```
class ClaseSujetoBase {
   void operacion (int valor) {
      _miVar += valor;
   }
   void operacionNotify (int valor) {
      self.operacion(valor);        // ejecuta la operación
      self.notify();                // dispara la notificación
   }
}
class MiSujeto extends ClaseSujetoBase {
   void operacion (int valor) {
      super.operacion(valor);
      _miVar += valor;
   }
}
```

Observer
Implementación

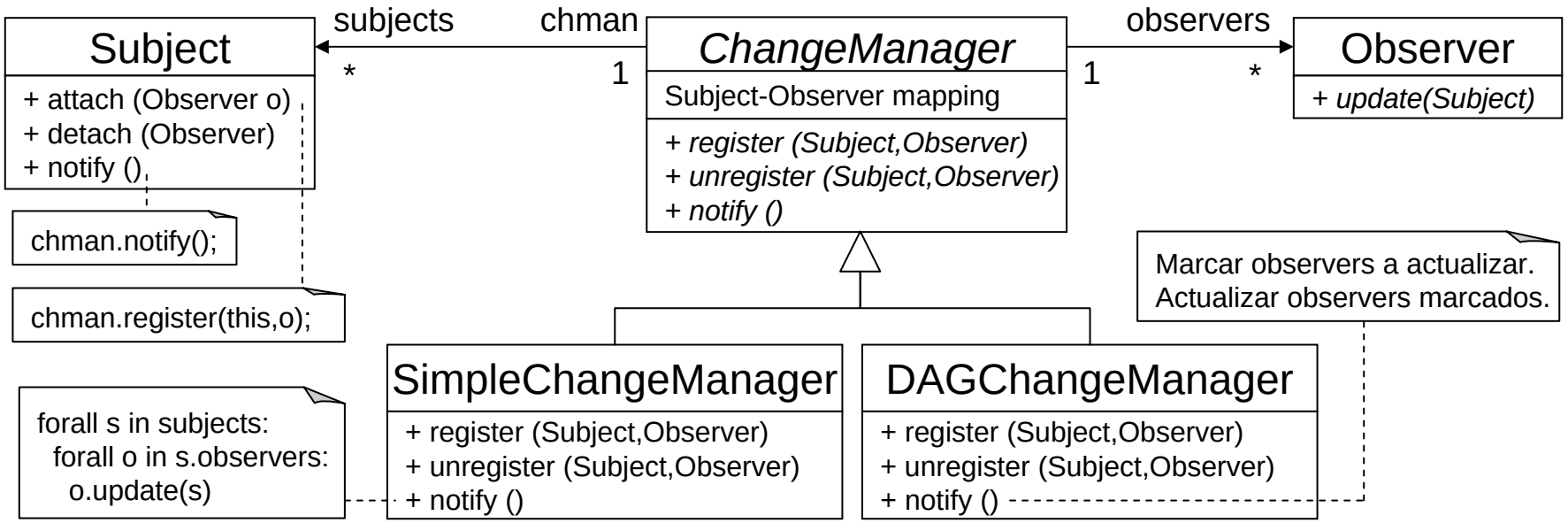
- Evitar protocolos de actualización específicos del observador
  - Modelo *pull*: el sujeto envía lo mínimo, los observadores piden lo necesario
    - inconveniente: puede ser poco eficiente
  - Modelo *push*: el sujeto envía información detallada del cambio, aunque los observadores no la necesiten.
    - inconveniente: observadores menos reutilizables

- Especificar las modificaciones de interés explícitamente
  - Hacer *update* más eficiente, haciendo que los observadores se registren sólo en los eventos que les interesan
  - Los observadores se subscriben a *aspectos* del sujeto

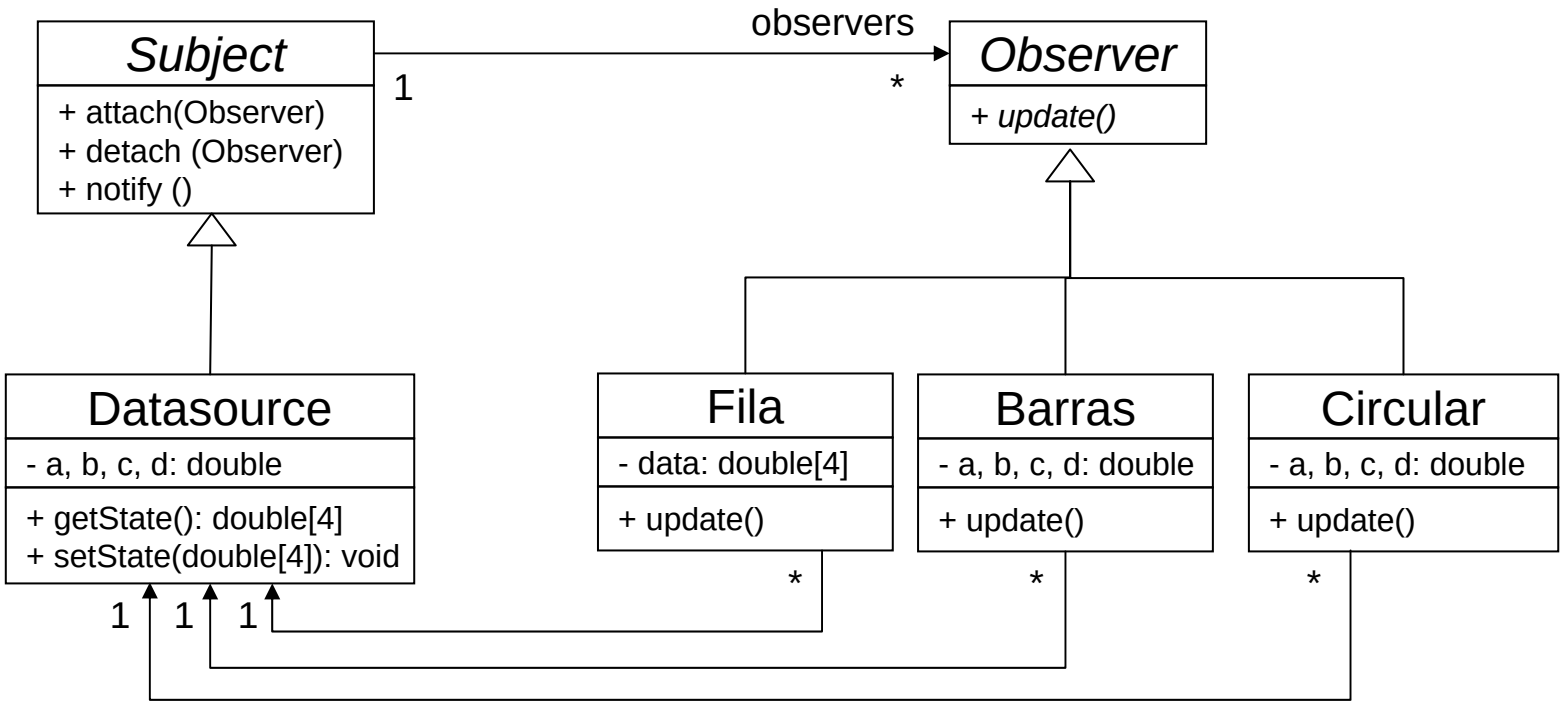
```
class Subject {
   public attach (Observer o, Aspect a) { ... }
}
class Observer {
   public update (Subject s, Aspect a) { ... }
}
```

Observer
Implementación

- Encapsular la semántica de actualizaciones complejas
  - Cuando la relación de dependencia entre sujetos y observadores es compleja, se puede usar un objeto intermedio para la gestión de cambios
  - Minimiza el trabajo de reflejar cambios de los sujetos en los observadores
  - Ej.: si se actualizan varios sujetos, hay que asegurar que los observadores se actualizan sólo después del cambio en el último sujeto



Observer
Ejemplo



Observer
Código de ejemplo


```
public abstract class Subject {
   protected List<Observer> _observers;

   public Subject() {
      _observers =
      new LinkedList<Observer>();
   }

   public void attach(Observer o) {
      _observers.add(o);
   }
   public void detach(Observer o) {
      _observers.remove(o);
   }
   public void notify() {
      Iterator<Observer> it;
      it = _observers.iterator();
      while (it.hasNext())
         it.next().update();
   }
}
```

```
public class Datasource
   extends Subject {
   private double _a, _b, _c, _d;

   public double[] getState () {
      double[] d = new double[4];
      d[0] = _a;
      d[1] = _b;
      d[2] = _c;
      d[3] = _d;
      return d;
   }

   public void setState(double[] d){
      _a = d[0];
      _b = d[1];
      _c = d[2];
      _d = d[3];
      this.notify();
   }
}
```

Observer
Código de ejemplo


```
public abstract class Observer {
   protected Subject _subject;

   public Observer (Subject s) {
      _subject = s;
      _subject.attach(this);
   }

   public abstract void update ();
}
```

```
public class Fila extends Observer {
   private double[] _data;

   public Fila (Subject s) {
      super(s);
      _data = new double[4];
   }

   public void update () {
      double[4] data;
      data =
        ((Datasource)_subject).getState();
      for (int i=0; i<4; i++)
         _data[i] = data[i];
      this.redraw();
   }

   public void redraw () { ... }
}
```

Observer
Código de ejemplo


```
public abstract class Observer {
   public abstract void update ();
}
```

```
public class Fila extends Observer {
   private Datasource _subject;
   private double[] _data;

   public Fila (Datasource s) {
      _subject = s;
      _subject.attach(this);
      _data = new double[4];
   }

   public void update () {
      double[4] data;
      data = _subject.getState();
      for (int i=0; i<4; i++)
         _data[i] = data[i];
      this.redraw();
   }

   public void redraw () { ... }
}
```

Observer
En java…

- La interfaz `java.util.Observer`
  - `void update (Observable o, Object arg)`

- La clase `java.util.Observable`
  - `Observable()`
  - `void addObserver(Observer)`
  - `int countObservers()`
  - `void deleteObserver(Observer o)`
  - `void deleteObservers()`
  - `void notifyObservers()`
  - `void notifyObservers(Object arg)`
  - `boolean hasChanged()`
  - `void clearChanged()`
  - `void setChanged()`