DISEÑO ORIENTADO A OBJETOS

Prof. Esp. Ing. Agustín Fernandez

Patrones de Diseño (comportamiento)

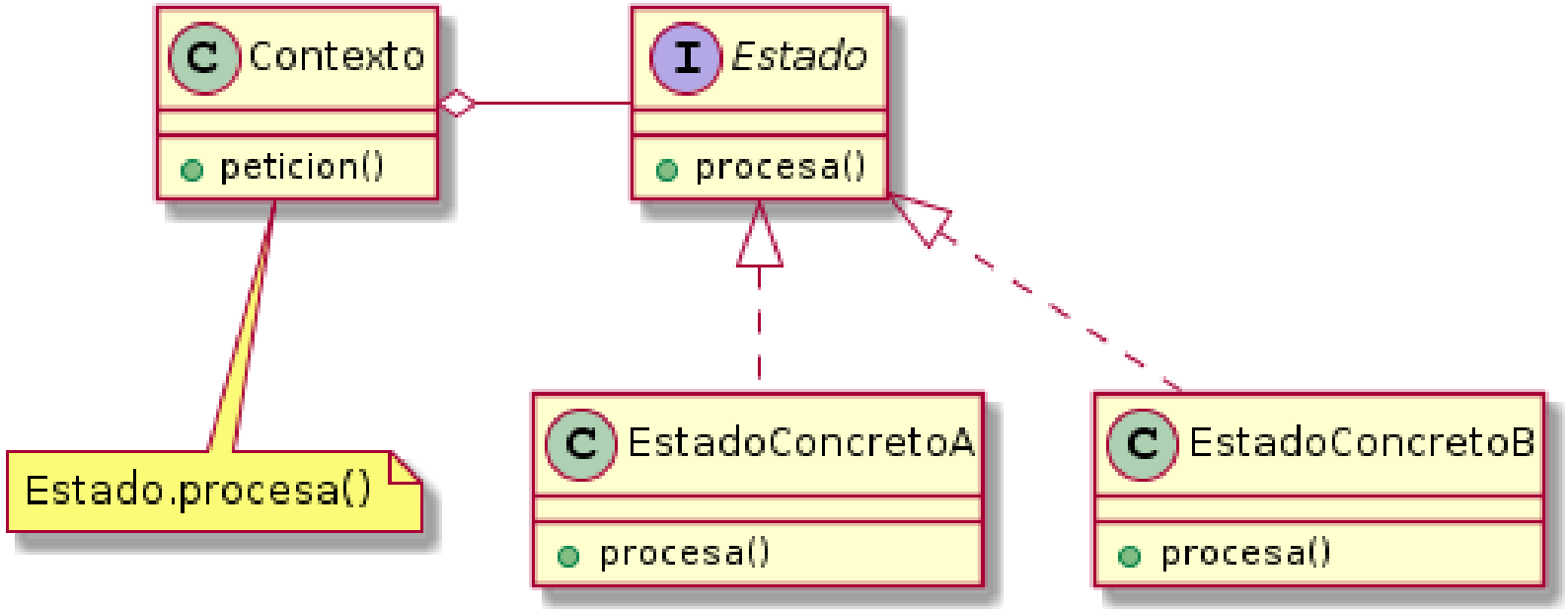
- Patrón State (Estado)
  - Permite que un objeto modifique su comportamiento cada vez que cambia su estado interno.
  - Parecerá que cambia la clase del objeto.
  - También conocido como Estados como Objetos.

Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Debe usarse este patrón cuando:**
  - Este patrón se utiliza cuando un determinado objeto tiene diferentes estados y también distintas responsabilidades según el estado en que se encuentre en un determinado instante.
  - También puede utilizarse para simplificar casos en los que se tiene un complicado y extenso código de decisión que depende del estado del objeto.

Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Estructura UML:**



Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Estructura UML (Cont.):**
  - **Contexto**: mantiene una instancia con el estado actual.
  - **Estado**: define interfaz para el comportamiento asociado a un determinado estado del Contexto.
  - **Estado concreto**: cada subclase implementa el comportamiento asociado con un estado del contexto.

Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Colaboraciones:**
  - El Contexto delega el estado específico al objeto Estado Concreto.
  - Un objeto Contexto puede pasarse a sí mismo como parámetro hacia un objeto Estado. De esta manera la clase Estado puede acceder al contexto si fuese necesario.
  - Contexto es la interfaz principal para el cliente. El cliente puede configurar un contexto con los objetos Estado. Una vez hecho esto los clientes no tendrán que tratar con los objetos Estado directamente.
  - Tanto el objeto Contexto como los objetos de Estado Concreto pueden decidir el cambio de estado.

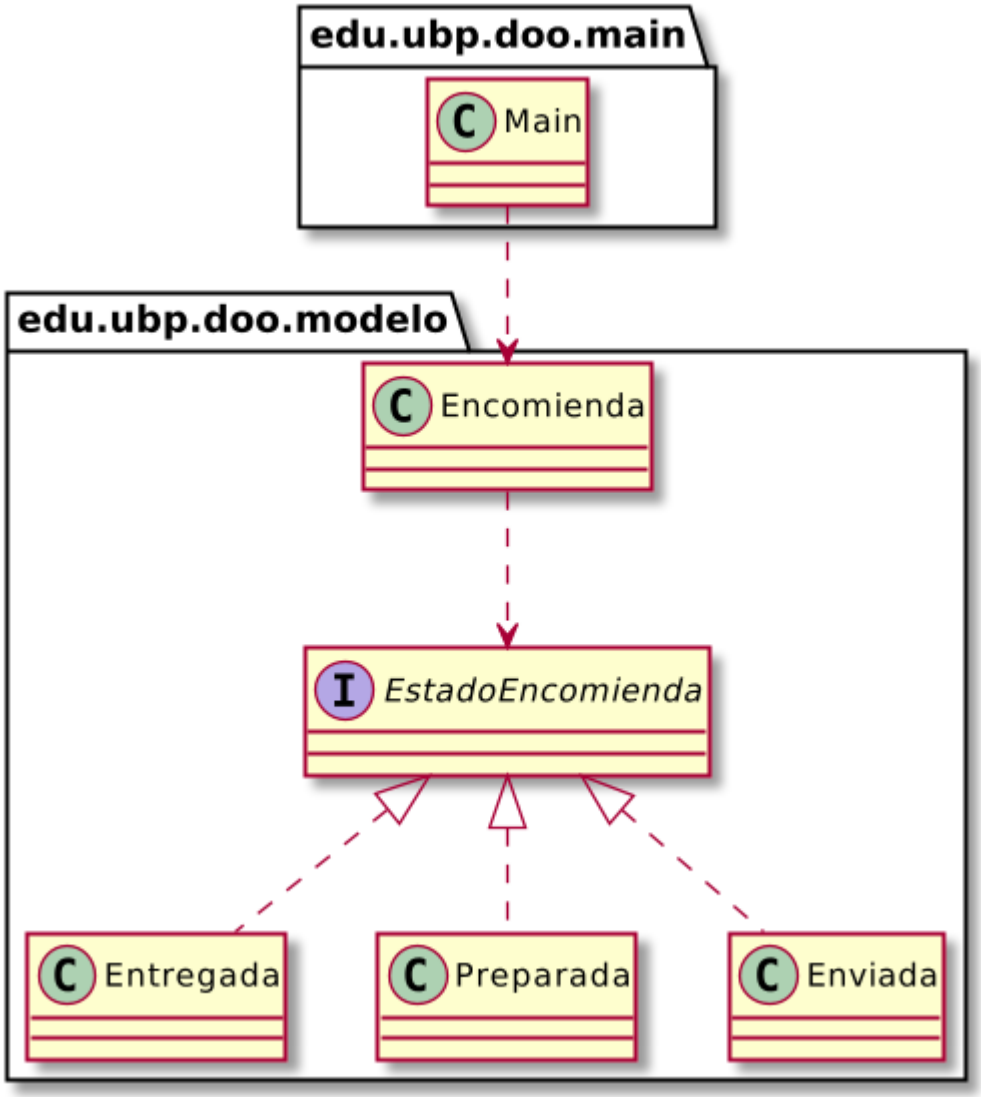
Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Ejemplo:**
  - Imaginemos que debemos modelar los posibles estados y comportamientos por los que debe pasar el envío de una encomienda.
  - Para nosotros dichos posibles estado son: Preparada, Enviada y Entregada.
  - Como breve explicación las transiciones de estados son las siguientes:

  - ¿Cómo modelaríamos esto mediante el patrón state?

Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Ejemplo:**



Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Ejercicio:** Supongamos que deseamos representar los tres estados de un semáforo común y corriente:
  - EstadoVerde
  - EstadoAmarillo
  - EstadoRojo
- Para nuestro caso, cuando el semáforo se inicia por primera vez su estado sera “Rojo”.
- Además se debe pensar que debe ser posible **mostrar** el estado en que se encuentra el semáforo en todo momento.
- También debemos recordar que debe ser posible **cambiar** el estado del semáforo en cualquier momento. Se aclara que, para este caso en particular, no se requiere cumplir con la secuencia correcta entre los cambios de estados del semáforo.
- ¿Como se resolvería esta situación usando este patrón?

Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Cosas a considerar:**
  - El patrón no indica exactamente dónde definir las transiciones de un estado a otro. Tenemos dos soluciones posibles:
    - Definiendo estas transiciones dentro de la clase contexto.
    - La otra es definiendo estas transiciones en las subclases de Estado. Pero hay que tener en cuenta que habrá más dependencia de código entre las subclases.
  - Los estados concretos pueden ser Singleton.

Patrones de Diseño (comportamiento)

- Patrón State (Estado)
- **Consecuencias:**
  - Se localizan fácilmente las responsabilidades de los estados concretos, dado que se encuentran en las clases que corresponden a cada estado.
  - Hace los cambios de estado explícitos.
  - Facilita la ampliación de estados mediante una simple herencia, sin afectar al Contexto.
  - Permite a un objeto cambiar de estado en tiempo de ejecución.
  - Los estados pueden reutilizarse: varios Contexto pueden utilizar los mismos estados siempre y cuando cada estado no tenga dependencia con el contexto.
  - Se incrementa el número de subclases.

Patrones de Diseño (comportamiento)

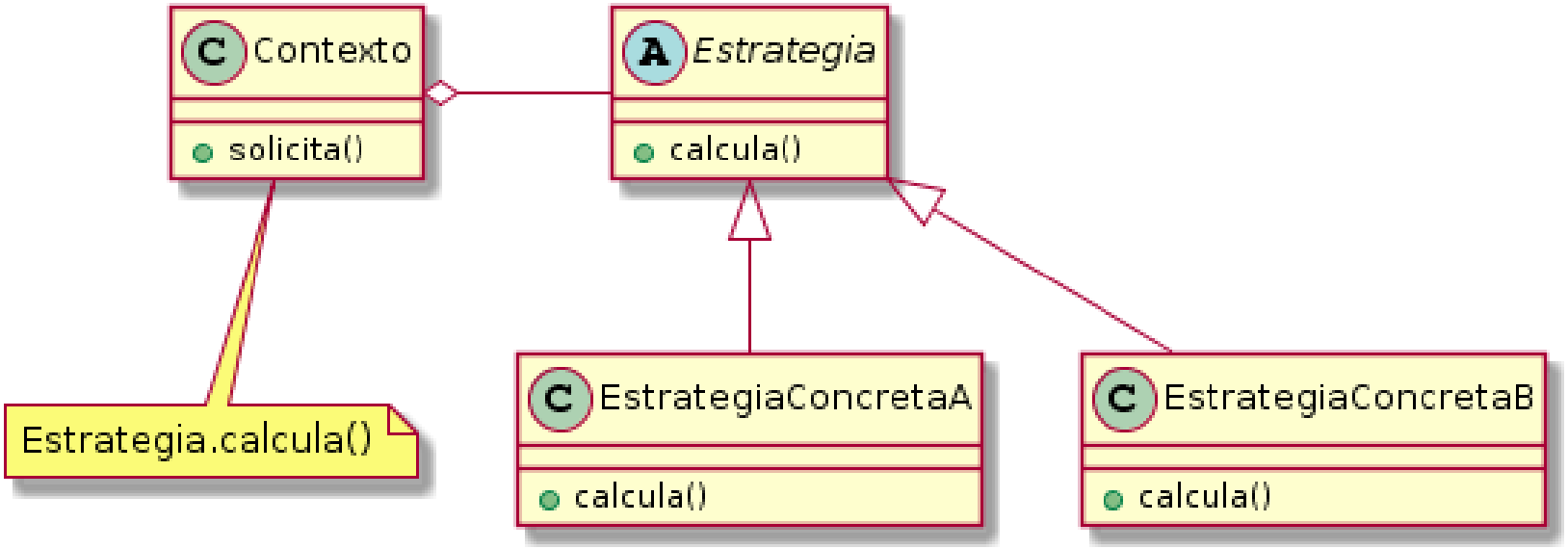
- Patrón Strategy (Estrategia)
  - Define una familia de algoritmos, los encapsula y los hace intercambiables.
  - Permite que un algoritmo varíe independientemente de los clientes que lo usen.
  - En base a un parámetro, que puede ser cualquier objeto, permite a una aplicación decidir en tiempo de ejecución el algoritmo que debe ejecutar.
  - Busca desacoplar bifurcaciones inmensas con algoritmos dificultosos según el camino elegido.
  - También conocido como Policy (Política)

Patrones de Diseño (comportamiento)

- Patrón Strategy (Estrategia)
- **Debe usarse este patrón cuando:**
  - Un programa tiene que proporcionar múltiples variantes de un algoritmo o comportamiento.
  - Se deban cambiar o agregar algoritmos, independientemente de la clase que lo utiliza.
  - Un algoritmo utiliza estructuras de datos complejas que no es necesario que los clientes conozcan.
  - Una clase define múltiples comportamientos y estos se representan como múltiples sentencias condicionales en sus operaciones. En vez de tener muchos condicionales podemos encapsularlos en clases Estrategia.

Patrones de Diseño (comportamiento)

- Patrón Strategy (Estrategia)
- **Estructura UML:**



Patrones de Diseño (comportamiento)

- Patrón Strategy (Estrategia)
- **Estructura UML (Cont.):**
  - **Estrategia**: declara una interfaz común a todos los algoritmos soportados.
  - **Estrategia Concreta:** implementa un algoritmo utilizando la interfaz Estrategia. Es la representación de un algoritmo.
  - **Contexto:** mantiene una referencia a Estrategia y según las características del contexto, optará por una estrategia determinada.

Patrones de Diseño (comportamiento)

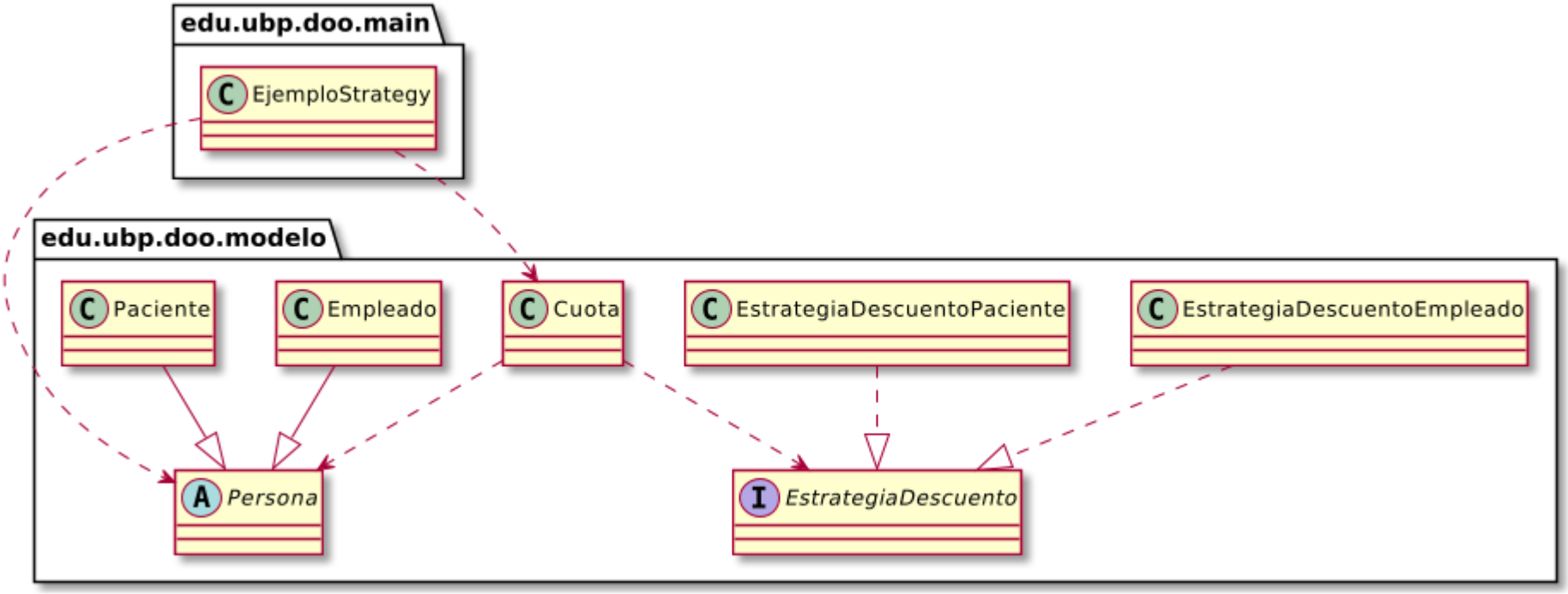
- Patrón Strategy (Estrategia)
- **Colaboraciones:**
  - Estrategia y Contexto colaboran para implementar el algoritmo elegido. Un Contexto puede pasar a la Estrategia todos los parámetros requeridos por el algoritmo cada vez que se llame a éste. Otra alternativa es que el Contexto se pase a sí mismo como argumento.
  - El Contexto redirige las peticiones de los clientes a su Estrategia correspondiente.
  - Los clientes crean normalmente un objeto Estrategias Concreta.

Patrones de Diseño (comportamiento)

- Patrón Strategy (Estrategia)
- **Ejemplo:**
  - Supongamos que debemos desarrollar la “pasarela” de descuentos de un sistema de una clínica privada. En esta la estrategia para calcular los descuentos son realizadas por “complejos” algoritmos sobre el total de la cuota y dependiendo de si se trata de un paciente común o un empleado del nosocomio.
  - Es decir que habrá una política de descuento dependiendo de si trata de un paciente o de un empleado de la clínica.
  - ¿Cómo haríamos esto aplicando este patrón?

Patrones de Diseño (comportamiento)

- Patrón Strategy (Estrategia)
- **Ejemplo:**



Paradigmas de Programación

- Patrón Strategy (Estrategia)
- **Ejercicio:**
  - Supóngase que se tiene una clase OperacionAritmetica que realiza cálculos sobre dos números: Sumar, Restar y Multiplicar.
  - Y para ello se nos ha pedido que tengamos en cuenta que cada una de esas operaciones son llevadas a cabo por algoritmos “complejos” que toman dos números y devuelven un resultado.
  - Además cabe aclarar que la clase OperaciónAritmetica realiza el calculo correspondiente aplicando el patrón Strategy.
  - ¿Cómo lo resolverías usando este patrón?

Patrones de Diseño (comportamiento)

- Patrón Strategy (Estrategia)
- **Consecuencias:**
  - Familias de algoritmos relacionados: Las jerárquias de clases de Estrategia definen una familia de algoritmos o comportamientos reutilizables por los Contextos.
  - Simplifica los Clientes: les reduce responsabilidad para seleccionar comportamientos o implementaciones de comportamientos alternativos. Esto simplifica el código de los objetos Cliente eliminando las expresiones if y switch.
  - Las Estrategias pueden definir diferentes implementaciones de un mismo comportamiento permitiendo elegir al Cliente el más adecuado en tiempo de ejecución.
  - Mayor numero de objetos.

Patrones de Diseño (comportamiento)

- Patrón Strategy (Estrategia)
- **Cosas a considerar:**
  - Strategy define comportamientos independientes de un objeto.
  - State define estados que por lo general son dependientes entre si.
  - Los estados descriptos con un patrón State varían a lo largo del tiempo.
  - Es probable que un comportamiento descripto por Strategy se asigne a un objeto al inicializarlo y luego ya no cambie.
  - En ambos casos lo que se persigue es encapsular el comportamiento e independizarlo del objeto:
    - En el caso de State ese “comportamiento” puede ir variando.
    - Y en el caso de Strategy lo que se busca es proporcionar diferentes alternativas de una misma tarea.

Patrones de Diseño (comportamiento)

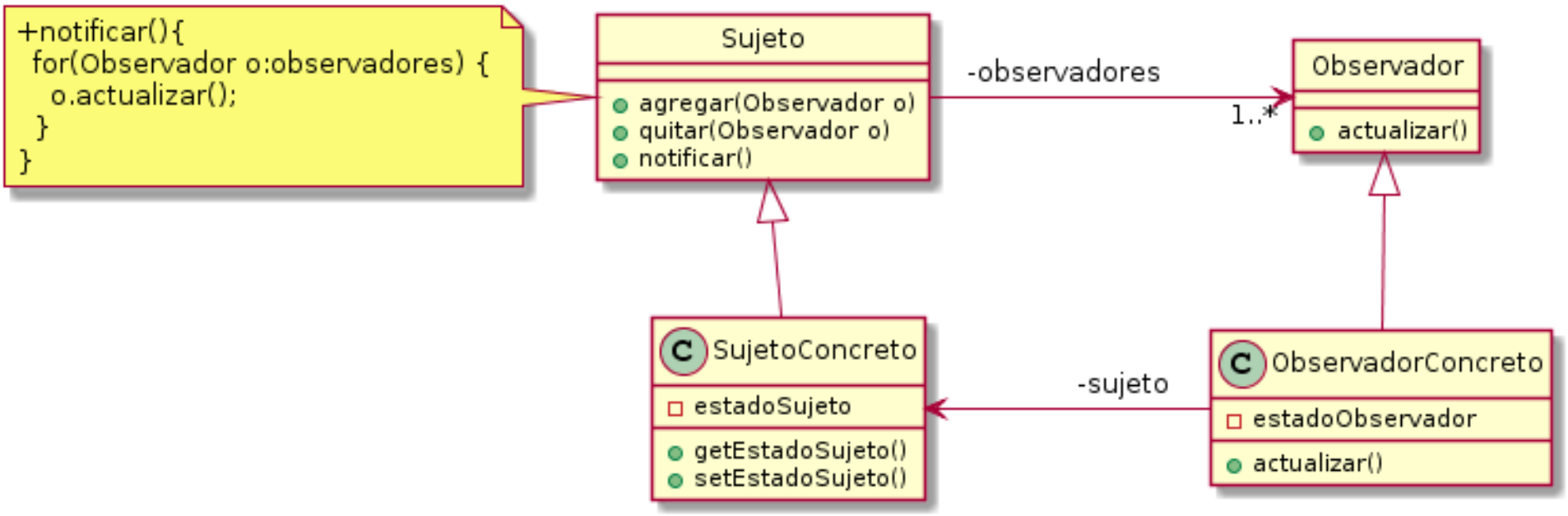
- Patrón Observer (Observador)
  - Define una dependencia uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de el.
  - Este patrón suele observarse en los frameworks de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir listeners a los objetos que pueden disparar eventos.
  - El patrón Observer es la clave del patrón de arquitectura MVC.
  - Conocido como Dependientes (Dependents) o Publicar – Subscribir (Publish - Subscribe)

Patrones de Diseño (comportamiento)

- Patrón Observer (Observador)
- **Debe usarse este patrón cuando:**
  - Una modificación en el estado de un objeto requiere cambios de otros, y no deseamos que se conozca el número de objetos que deben ser cambiados.
  - Queremos que un objeto sea capaz de notificar a otros objetos sin hacer ninguna suposición acerca de los objetos notificados.
  - Una abstracción tiene dos aspectos diferentes, que dependen uno del otro; si encapsulamos estos aspectos en objetos separados permitiremos su variación y reutilización de modo independiente.

Patrones de Diseño (comportamiento)

- Patrón Observer (Observador)
- **Estructura UML:**



Patrones de Diseño (comportamiento)

- Patrón Observer (Observador)
- **Estructura UML (Cont.):**
  - **Subject**: conoce a sus observadores y ofrece la posibilidad de añadir y eliminar observadores. Posee un método llamado attach() y otro detach() que sirven para agregar o remover observadores en tiempo de ejecución.
  - **Observer**: define la interfaz que sirve para notificar a los observadores los cambios realizados en el **Subject**.
  - **SubjectConcreto**: almacena el estado de interés para los objetos **ObserverConcreto** y envía un mensaje a sus observadores cuando su estado cambia.
  - **ObserverConcreto**: mantiene una referencia a un **SubjectConcreto**. Guarda un estado que debería ser consistente con el del **Subject**. Implementa la interfaz de actualización de **Observer** para mantener su estado consistente con el del **Subject**.

Patrones de Diseño (comportamiento)

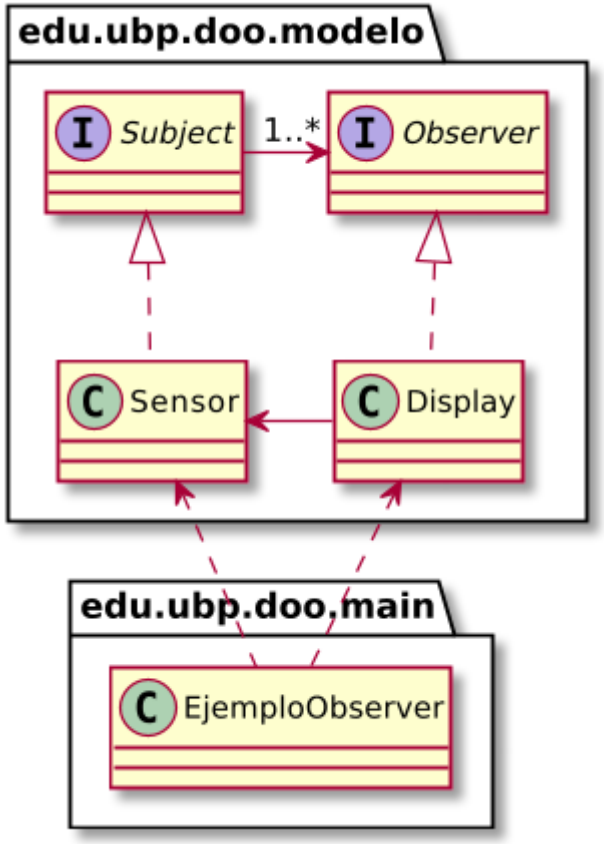
- Patrón Observer (Observador)
- **Colaboraciones:**
  - El objeto observado notifica a sus observadores cada vez que ocurre un cambio.
  - Después de ser informado de un cambio en el objeto observado, cada observador concreto puede pedirle la información que necesita para reconciliar su estado con el de aquél.

Patrones de Diseño (comportamiento)

- Patrón Observer (Observador)
- **Ejemplo:**
  - Imaginemos que tenemos un sensor de temperatura ambiente y que existen un Objeto “Dysplay” que esta interesado en que se lo notifique cada vez que el valor del sensor de temperatura cambia.
  - Cabe aclara que en nuestro caso haremos nosotros mismo, de manera explicita, el cambio en el sensor de temperatura.
  - ¿Como resuelve este problema este patrón?

Patrones de Diseño (comportamiento)

- Patrón Observer (Observador)
- **Ejemplo:**



Patrones de Diseño (comportamiento)

- Patrón Observer (Observador)
- **Ejercicio:** Imaginemos que tenemos un reloj y que existen diferentes Objetos “Vistas” que esta interesado en que se lo notifique cada vez que el valor del reloj se actualice permitiéndoles mostrar, a los mismos, la hora actual.
- Los tipos de los objetos vistas son:
  - VistaDigital
  - VistaAnalogica
- Cabe aclara que en nuestro caso haremos nosotros mismo, de manera explicita, el cambio de la hora en nuestro reloj.
- ¿Como se resolvería esta situación usando este patrón?

Patrones de Diseño (comportamiento)

- Patrón Observer (Observador)
- **Cosas a considerar:**
  - Los observadores pueden observar a varios objetos subject (Observable) a la vez:
    - Si pero será necesario ampliar el servicio update() para permitir conocer a un objeto observer dado cuál de los objetos subject (Observable) que observa le ha enviado el mensaje de notificación.
    - Una forma de implementarlo es añadiendo un parámetro al servicio update() que sea el objeto subject (Observable) que envía la notificación (el remitente).

Patrones de Diseño (comportamiento)

- Patrón Observer (Observador)
- **Consecuencias:**
  - Permite modificar las clases subjects (Observables) y las observers independientemente.
  - Permite añadir nuevos observadores en tiempo de ejecución, sin que esto afecte a ningún otro observador.
  - Permite comunicación broadcast, es decir, un objeto subject (Observable) envía su notificación a todos los observers sin enviárselo a ningún observer en concreto (el mensaje no tiene un destinatario concreto). Todos los observers reciben el mensaje y deciden si hacerle caso o ignorarlo.
  - La comunicación entre los objetos subject y sus observadores es limitada: el evento siempre significa que se ha producido algún cambio en el estado del objeto y el mensaje no indica el destinatario.