DISSO 10 – PATRONES DE COMPORTAMIENTO

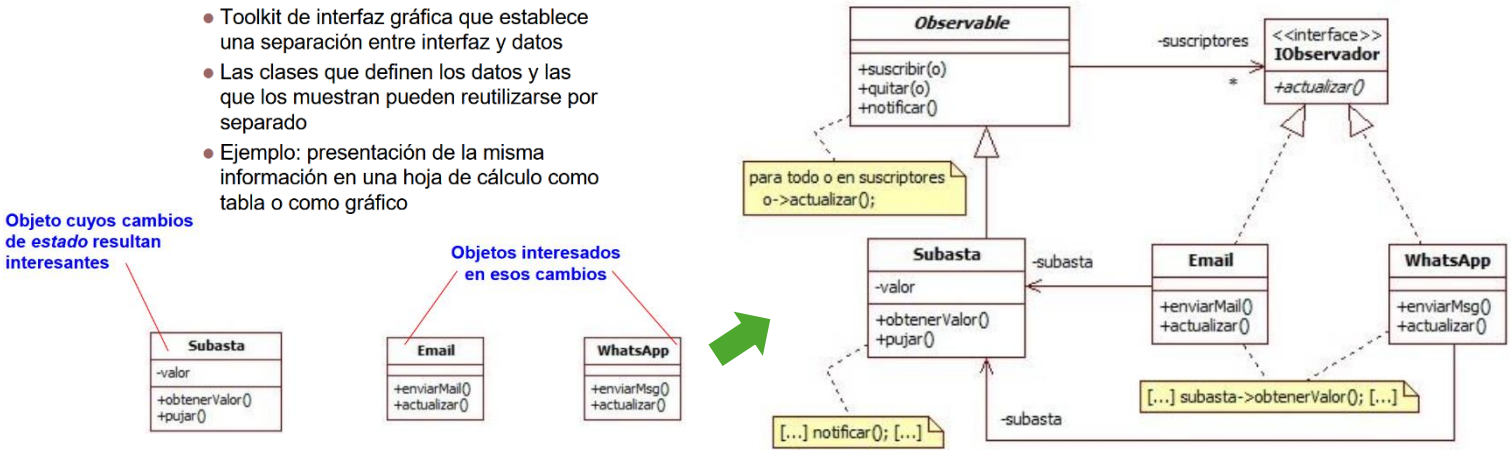
OBSERVER

- OBSERVADOR ⟶ el **cambio de estado** en un objeto (sujeto) se **notifica** a los objetos **suscritos** a él (observadores), que **sincronizan** su estado con el del sujeto.
- Establece una **dependencia de uno a muchos** entre objetos.

Se usa cuando:

- Un cambio en un objeto requiera modificar otros objetos **sin que importe cuántos** son.
- Un objeto deba ser capaz de hacer notificaciones a otros **sin hacer suposiciones sobre quiénes** son exactamente.

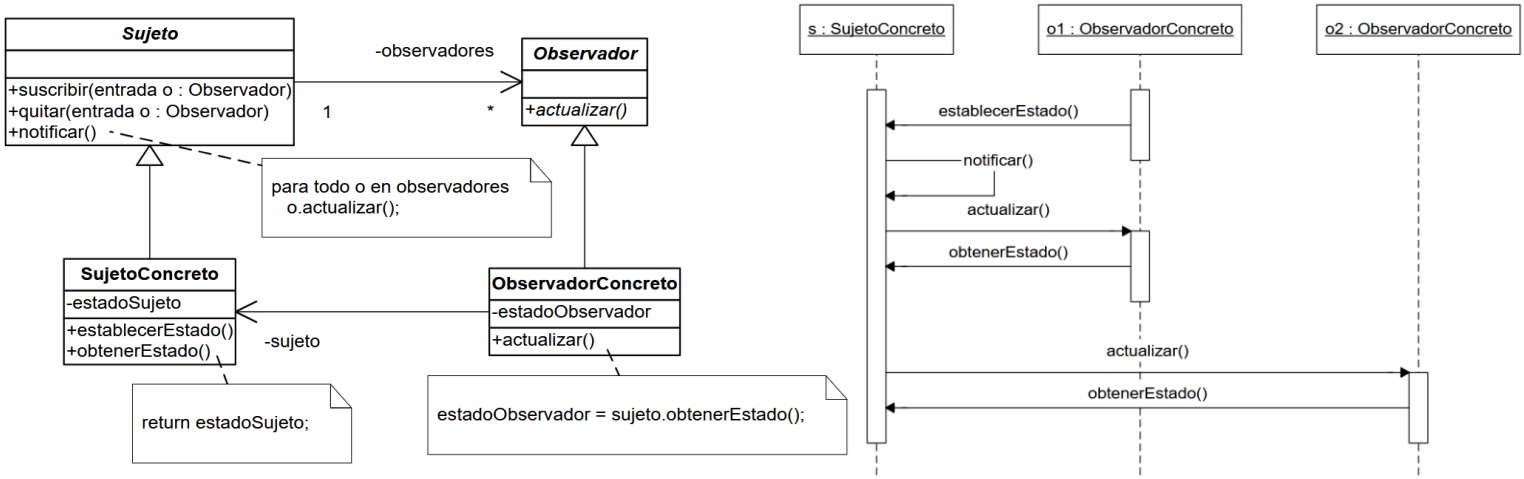
- Toolkit de interfaz gráfica que establece una separación entre interfaz y datos
- Las clases que definen los datos y las que los muestran pueden reutilizarse por separado
- Ejemplo: presentación de la misma información en una hoja de cálculo como tabla o como gráfico



PARTICIPANTES

- SUJETO ⟶ conoce a sus observadores (pero no sus detalles) y dispone de una **interfaz** para añadirlos y quitarlos.
- OBSERVADOR ⟶ define una **interfaz** para actualizar objetos concretos ante cambios en un sujeto.
- SUJETO CONCRETO ⟶ almacena el **estado de interés** para sus observadores concretos y les **notifica** cambios en ese estado.
- OBSERVADOR CONCRETO ⟶ referencia un sujeto concreto con el que sincroniza su estado.

ESTRUCTURA Y COMPORTAMIENTO



VENTAJAS E INCONVENIENTES

- ✓ Habilita la **modificación independiente** de sujetos y observadores.
- ✓ Hay **mínimo acoplamiento** ⟶ el sujeto no necesita saber la clase concreta de ningún observador.
- ✓ Capacidad de comunicación mediante **difusión**.
- × Los observadores no se conocen entre sí ⟶ la operación sobre el sujeto puede provocar **actualizaciones en cascada** de observadores.
- × El protocolo de **actualización simple** no detalla cambios en el sujeto.

IMPLEMENTACIÓN

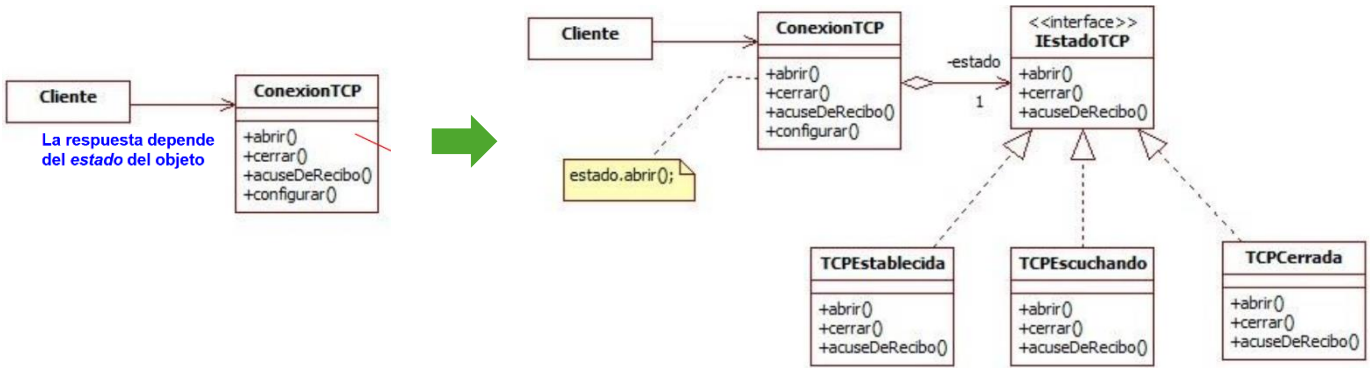
- **Observar varios sujetos** requiere ampliar la operación $actualizar()$.
- Política de actualizaciones:
  - Al cambiar el estado del sujeto.
    - ↳ Es importante documentar qué operaciones de Sujeto disparan notificaciones.
  - Provocadas por los clientes.
- Se debe decidir el **grado de detalle** que los sujetos proporcionan sobre sus cambios.
- Se precisa controlar los **eventos de interés** de cada **observador**.

STATE

- ESTADO → permite que un objeto **modifique su comportamiento** cada vez que cambia su estado.
- **Parecerá** que **cambia la clase** de ese objeto.

Se usa cuando:

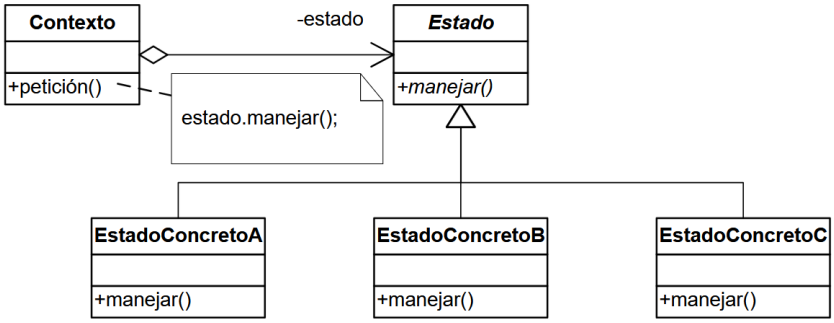
- El comportamiento de un objeto **depende de su estado** y cambia en **tiempo de ejecución**.
- Algunos métodos presentan estructuras **condicionales con múltiples ramas** formadas por bloques extensos de código.



PARTICIPANTES

- CONTEXTO → define una **interfaz** de interés para los clientes y dispone de **una instancia** de un Estado Concreto que determina su estado.
- ESTADO → declara una **interfaz** que encapsula el comportamiento asociado a los distintos estados del Contexto.
- ESTADO CONCRETO → implementa el comportamiento asociado a un estado particular del Contexto.

ESTRUCTURA Y COMPORTAMIENTO



- El **Contexto delega** las peticiones que dependen de su estado en el objeto Estado Concreto actual.
- El **Contexto** puede pasarse a **sí mismo** como **parámetro**.
- El **Contexto** o cualquier **Estado Concreto** pueden decidir qué estado sigue a otro y bajo qué circunstancias.

VENTAJAS E INCONVENIENTES

- Sitúa en **un solo objeto** todo el comportamiento asociado a un estado:
  - ✓ Fácil **adición** de **estados** y **transiciones**.
  - ✓ Simplificación de estructuras **condicionales**.
  - × Incrementa el **número de clases**.
- ✓ Explicita **transacciones** → protege frente a estados inconsistentes.
- ✓ Los **estados** pueden **compartirse** entre diferentes contextos si no tienen atributos.

IMPLEMENTACIÓN

? ¿Quién define las transiciones?
- **Contexto** → si los criterios son fijos.
- **Cada estado** concreto designa su sucesor → mayor flexibilidad y acoplamiento.

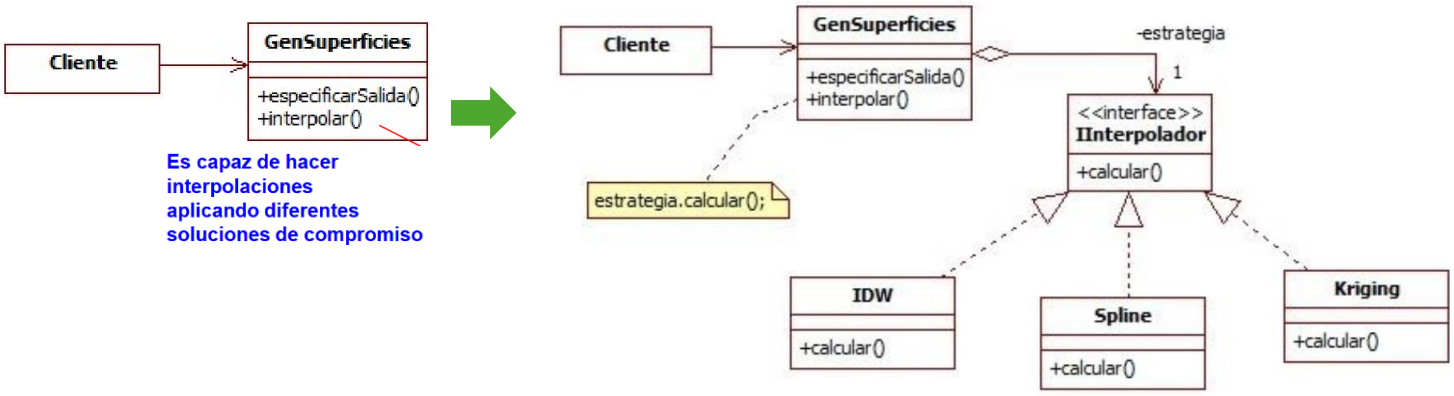
? ¿Cómo se crean/destruyen estados?
- Se crean sólo **si se necesitan** y **se destruyen** después.
- Se crean **al principio** y **no se destruyen**.

STRATEGY

- ESTRATEGIA → define una **familia de algoritmos** y los hace **intercambiables**, encapsulando los algoritmos en **objetos** en los que se **delegan peticiones**.
- Permite configurar una clase con **un comportamiento** de entre **varios posibles**.
- Permite que un algoritmo **varíe** con **independencia de sus clientes**.

Se usa cuando:

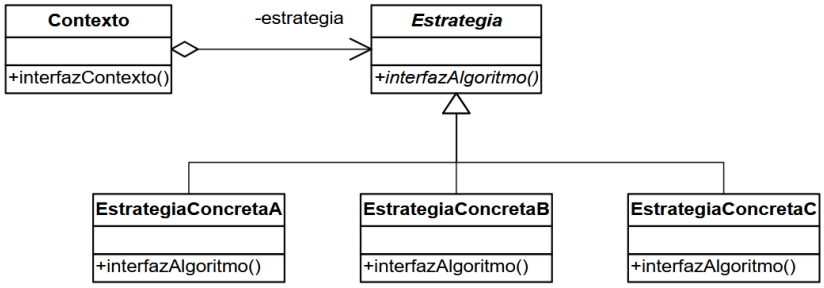
- Existen varias clases relacionadas que **difieren sólo** en su **comportamiento**.
- Se necesitan **variantes de un algoritmo**.
- Un algoritmo usa **datos** que los **clientes no** necesitan **conocer**.
- Una clase implementa comportamientos complejos en las diferentes **ramas** de largas **estructuras condicionales**.



PARTICIPANTES

- ESTRATEGIA → declara la **interfaz** común a todo algoritmo, utilizada por Contexto para ejecutar un algoritmo concreto.
- ESTRATEGIA CONCRETA → implementa un algoritmo en particular.
- CONTEXTO → referencia a una estrategia y se configura con una estrategia concreta.
  - ↳ Puede redefinir una **interfaz** para que la estrategia acceda a sus datos.

ESTRUCTURA Y COMPORTAMIENTO



- **Estrategia** y **Contexto** interactúan para implementar el algoritmo.
- **Contexto** puede pasar a su estrategia todos los datos requeridos por el algoritmo o incluso pasarse **a sí mismo**.
- **Contexto delega** en su estrategia determinadas peticiones recibidas.
- **Otros objetos** crean la **Estrategia Concreta** y se la pasan al contexto.

VENTAJAS E INCONVENIENTES

- ✓ Define una familia de algoritmos intercambiables ligados a diferentes **soluciones de compromiso**.
- ✓ Permite **modificar** cualquier algoritmo con **independencia de su contexto**.
- ✓ Elimina las **estructuras condicionales**.
- × Las estrategias más simples pueden **no usar toda la información** recibida.
- × **Aumenta** el **número de objetos**.

IMPLEMENTACIÓN

- La **estrategia** puede **referenciar** a su **contexto**, evitando el paso de datos innecesarios.
- Si no dispone de estrategia asociada, el **contexto** puede implementar el **comportamiento predeterminado**.

COMPARACIÓN ENTRE PATRONES

Encapsulación de lo que **varía**.

- Varios patrones **encapsulan** un **aspecto variable** de una clase **en otra clase**.
- Definen una **clase abstracta** para el objeto encapsulado, de la que toman su nombre (p. e. Strategy).

Comunicación **centralizada** vs comunicación **distribuida**.

- Mediator y Observer son rivales:
  - **Observer distribuye** comunicación introduciendo objetos observador y sujeto poco acoplados → mayor facilidad de **reutilización**.
  - **Mediator centraliza** comunicación en el mediador → patrón de comunicación **más sencillo**.

Desacoplamiento entre emisores y receptores.

- Los objetos que se refieren unos a otros explícitamente se vuelven dependientes entre sí.
- **Command**, **Observer**, **Mediator** y **Chain of Responsibility** tratan el problema de cómo desacoplar emisores y receptores.

EPÍLOGO

- Los patrones de diseño proporcionan un **vocabulario común** ⟶ elevan la abstracción y reducen la complejidad del sistema.
- Ayudan a **documentar** sistemas existentes y facilitan el **aprendizaje** de técnicas OO.
- **Complementan** métodos de **diseño OO** (que no reflejan la experiencia de los expertos).
- Facilitan la **transición** del **análisis** al **diseño**.
- Previenen y guían la **refabricación** de un desarrollo software reutilizable.

REFLEXIÓN FINAL

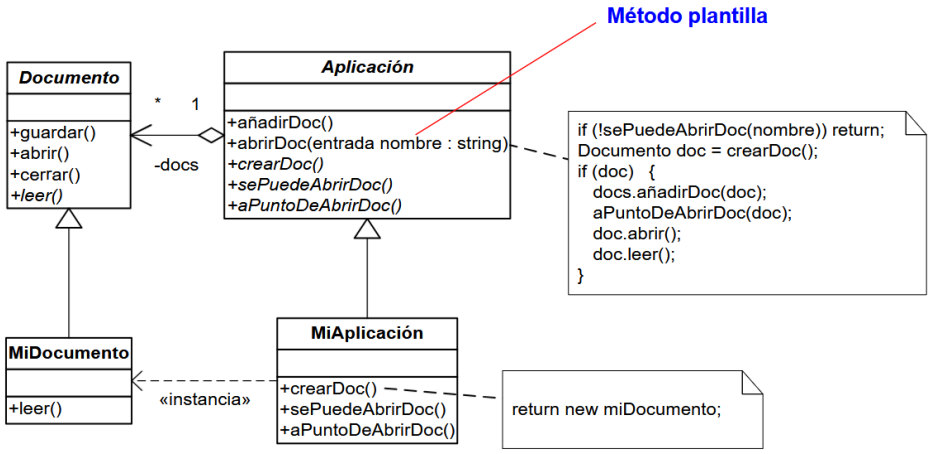
- Los mejores diseños combinan diversos patrones que se superponen para producir un mejor resultado.

TEMPLATE METHOD

- MÉTODO PLANTILLA ⟶ define en **esqueleto de un algoritmo** dejando algunos de sus pasos en manos de las subclases.
- Permite **redefinir ciertos pasos** del algoritmo sin cambiar su estructura.
- Cada paso invoca a una operación abstracta o a una operación concreta.
- Las subclases completan el algoritmo implementando operaciones abstractas.

Aplicación:

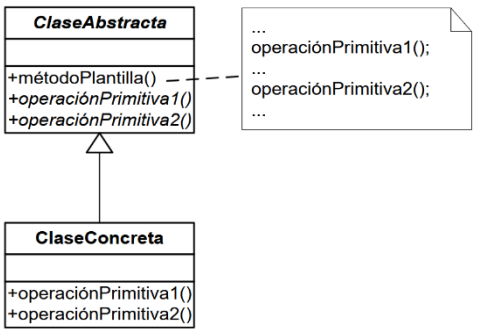
- Para implementar partes de un algoritmo que no cambian y dejar que subclases definan el comportamiento variable.
- Cuando el comportamiento repetido deba ser factorizado en una clase común para evitar código duplicado.
- Para controlar extensiones de subclases mediante operaciones de **enganche**.



PARTICIPANTES

- CLASE ABSTRACTA:
  - Declara operaciones primitivas abstractas que son definidas por las subclases para implementar los pasos de un algoritmo.
  - Implementa un método plantilla que define el **esqueleto** del algoritmo.
- CLASE CONCRETA ⟶ implementa operaciones primitivas para realizar los pasos.

ESTRUCTURA



OPERACIONES INVOCADAS POR EL MP

- Primitivas: abstractas en Clase Abstracta (concretas en Clase Concreta).
- Concretas en Clase Abstracta.
- Operaciones de enganche (concretas en Clase Abstracta):
  - Métodos vacíos en el padre.
  - **Pueden** ser redefinidas (vs. Operaciones abstractas, que **deben** ser redefinidas).