PRÁCTICA 1
PATRONES DE DISEÑO

PABLO MOLINA SÁNCHEZ
INGENIERÍA DEL SOFTWARE AVANZADA

CUESTIÓN 1: Consideremos los siguientes patrones de diseño: Adaptador, Decorador y Representante. Identifique las principales semejanzas y diferencias entre cada dos ellos.

- Semejanzas:
  - Se definen como patrones estructurales
  - Decorador y Representante porque tienden a operar en un objeto.
  - Son patrones estructurales
- Diferencias:
  - Adaptador cambia la interfaz de un objeto mientras que Decorador amplía la extensión de funcionalidades de un objeto sin cambiar la interfaz.
  - Decorador cambia las responsabilidades del objeto, mientras que Adaptador no.
  - Adaptador envuelve interfaces mientras que Decorador y Representante operan un objeto.
  - Representante es capaz de crear un objeto si este no existe, mientras que Decorador no.
  - Decorador puede añadir funcionalidades colocando un decorador tras otro mientras que Representante no debe colocar un proxy tras otro.

CUESTIÓN 2: Consideremos los patrones de diseño de comportamiento Estrategia y Estado. Identifique las principales semejanzas y diferencias entre ellos.

- Semejanzas:
    - Estado y Estrategia encapsulan comportamientos.
    - Estado y Estrategia permiten de manera sencilla añadir nuevos estados y estrategias sin afectar el objeto que los usa.
    - Son patrones de comportamiento
- Diferencias:
    - El orden de transición de estado está bien definido en Estado mientras que esto es inexistente en Estrategia.
    - El cambio en la estrategia lo realiza el cliente, pero el cambio en el estado puede hacerlo el propio objeto de estado.
    - Estado envuelve diferentes comportamientos en forma de diferentes objetos de estado y Estrategia envuelve diferentes comportamientos en forma de diferentes objetos de estrategia.
    - Estado responde a las preguntas “qué” y “cuándo” de un objeto, mientras que Estrategia define el “cómo” de un objeto.

CUESTIÓN 3: Consideremos los patrones de diseño de comportamiento Mediador y Observador. Identifique las principales semejanzas y diferencias entre ellos. .

- Semejanzas:
    - Un Mediador puede ser implementado como un tipo de Observador.
    - Son patrones de comportamiento
- Diferencias:
    - Observador distribuye la información, mientras que Mediador la encapsula
    - Mediador reduce las dependencias entre objetos mientras que Observador define una dependencia de uno a muchos entre objetos.
    - En Observador no se conocen los “observadores”, mientras que en Mediador se conocen de antemano los “colegas”.

CLIENTE E-LOOK:

https://github.com/pablomolinasanchez/PatronDise-o1_pablomolinasanchez

En este ejercicio, nos encontramos con un programa que describe el funcionamiento de e-look, un programa cliente con el que el usuario puede gestionar sus mensajes almacenados en un *Mailbox*. Esta clase se sirve de *Email*, con su constructor que crea nuevos emails. En el ejercicio se nos pide que seamos capaces de mejorar el sistema proporcionando herramientas que nos permitan seleccionar estrategias distintas de ordenación de los correos electrónicos. Por tanto, utilizaremos el patrón de diseño **Strategy**, ya que el ejercicio nos pide que el patrón usado debe permitirnos extender fácilmente los criterios de ordenación que se quiera, así como poder cambiar de un criterio de ordenación a otro. Estos dos factores los consideré claves para la elección de este patrón de diseño.

Para utilizar este patrón de diseño crearemos la interfaz *Orden*. Esta interfaz tendrá el método before(Email m1, Email m2) que se será implementado según el criterio que el cliente elija. Para crear las distintas estrategias añadiremos las clases *OrdenFrom, OrdenSubject, OrdenText, OrdenDate* y *OrdenPriority* que implementarán la interfaz creando para cada una la funcionalidad necesaria según el tipo de criterio que solicite el cliente (origen, tema, texto, día y prioridad). En la clase *OrdenPriority* estableceremos el orden de prioridad creando el enumerado *Priority*, contenido por {DESTACADO, IMPORTANTE, RECIBIDO, SPAM}.

Estas estrategias de ordenación las recogemos en la clase Mailbox, que inicialmente como en todos los clientes de correos , vendrá predeterminada según su fecha. Para ello creamos la función getOrden() que devolverá la clase con el orden elegido por el cliente en la función setOrden(Orden definoOrden). Con la función sort() ordenamos los emails que se basa en una operación before() ya que depende del criterio elegido para visualizar el mailbox. Posteriormente, la función show() nos muestra los emails ordenados. Así, llegamos a nuestro objetivo del ejercicio de ordenar según el criterio elegido por el cliente. Para las pruebas hemos implementado dos funciones más para añadir y eliminar emails y ver si el resto de funciones completan su trabajo satisfactoriamente.