Práctica 1. Patrones de Diseño

Sergio Martín Vera https://github.com/smv762e/e-look

Q1. Consideremos los siguientes patrones de diseño: Adaptador, Decorador y Representante. Identifique las principales semejanzas y diferencias entre cada dos ellos (no es suficiente con definirlos, sino describir explícitamente similitudes y semejanzas concretas).

Patrón adaptador:

Conocido como *Wrapper*, es un tipo de patrón estructural con una versión para clases y otra para objetos. Su función es adaptar una interfaz ya existente en una nueva que cumpla los requisitos esperados.

Patrón decorador:

Es un tipo de patrón estructural para objetos. Su función es dotar de funcionalidades de forma dinámica a objetos mediante composición.

Patrón representante:

Conocido como *Proxy,* es un tipo de patrón estructural para objetos. Su función es proporcionar un representante para controlar el acceso a un objeto.

Semejanza principal:

Son patrones estructurales para objetos (Adaptador tiene una versión para clases)

Semejanzas y diferencias entre adaptador y decorador:

Semejanzas	Diferencias
Son capaces de añadir funcionalidades nuevas con respecto a la clase inicial.	Adaptador proporciona una interfaz diferente a su sujeto y Decorador una interfaz mejorada. Adaptador cambia la interfaz de un objeto y Decorador mejora las responsabilidades del objeto. Decorador es más claro para el cliente.

Semejanzas y diferencias entre decorador y representante:

Semejanzas	Diferencias
Tienen propósitos diferentes, pero estructuras similares. Ambos describen un nivel de direccionamiento indirecto a otro objeto, y las implementaciones mantienen referencia al objeto al que reenvían las solicitudes.	Representante proporciona la misma interfaz de un objeto y Decorador una interfaz mejorada.

Semejanzas y diferencias entre representante y adaptador:

Semejanzas	Diferencias
	Adaptador proporciona una interfaz diferente a su sujeto y Representante la misma interfaz que el objeto.

Q2. Consideremos los patrones de diseño de comportamiento Estrategia y Estado. Identifique las principales semejanzas y diferencias entre ellos.

Patrón estrategia:

Es un tipo de patrón de comportamiento de objetos. Su función es definir un conjunto de algoritmos, los agrupa en clases distintas, y los hace intercambiables. Esto permite que el algoritmo cambie, independientemente de lo que utilice el cliente.

Patrón estado:

Es un tipo de patrón de comportamiento de objetos. Su función es permitir que un objeto cambie su comportamiento cuando cambia su estado.

Semejanzas	Diferencias
Ambos son un tipo de patrón de comportamiento de objetos.	Tienen una intención de uso diferentes, es decir, se utilizan para resolver problemas diferentes, a pesar de que tienen implementaciones similares.

Q3. Consideremos los patrones de diseño de comportamiento Mediador y Observador. Identifique las principales semejanzas y diferencias entre ellos.

Patrón mediador:

Es un tipo de patrón de comportamiento de objetos. Su función es definir un objeto como conjunto de objetos que interactúa entre sí.

Patrón observador:

Es un tipo de patrón de comportamiento de objetos. Su función es definir dependencias entre observadores y observables. Se utiliza para mantener la consistencia entre objetos relacionados sin acoplarlos fuertemente para no comprometer su reutilización.

Semejanzas	Diferencias
Ambos son un tipo de patrón de comportamiento de objetos.	Observador distribuye la comunicación introduciendo objetos “observador” y “sujeto”, mientras que un objeto Mediador encapsula la comunicación entre otros objetos.

Cliente de correo e-look:

El patrón de diseño más adecuado para desarrollar este programa es el patrón Estrategia. Gracias a este patrón, podemos cambiar la forma en la que el método before() se ejecuta y podemos extender sus funcionalidades.

Para filtrar los mensajes, se han creado dos tipos de filtros (los que pueden tener más utilidad): BeforeDate() para filtrar por fechas y BeforePriority() para filtrar por prioridad.

Clase Mailbox

Atributos

+ LinkedList<Email> email = new LinkedList<>(); // Lista que guarda los mensaje

- FilterType filter; // Tipo de filtro

Métodos

+ Mailbox() // Constructor de la clase

+ void show() // Para mostrar los mensajes

- void sort() // Para ordenar los mensajes

- Boolean before(Email, Email) // Comparación entre mensajes

+ setBefore(FilterType) // Para seleccionar el tipo de filtro

Clase Email

Atributos

+ String from // Autor del mensaje

+ String subject // Asunto del mensaje

+ Date date // Fecha del mensaje

+ Priority priority // Prioridad del mensaje

- String text // Texto del mensaje

Métodos

+ Email(String, String, Date, Priority, String) // Constructor de la clase

Enum Priority

+ enum Priority { LOW, STANDARD, HIGH } // Los niveles de prioridad

Clases BeforeDate y BeforePriority

Métodos

+ Boolean before(Email, Email) // Para comparar entre mensajes

Ambas clases implementan la interfaz FilterType.

La clase TestEmail, prueba el funcionamiento de este programa.