ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Práctica 1: Patrones de Diseño

Realizado por
Florín Babusca Voicu

UNIVERSIDAD DE MÁLAGA
MÁLAGA, ABRIL DE 2022

1. Cuestiones

Q1 Adaptador (Wrapper) vs Decorador vs Representante

- Similitudes:
  - El patrón Adaptador y el Decorador son capaces de añadir funcionalidades nuevas a la clase original.
  - Los tres patrones ofrecen una clase con la cual interactuar con otro elemento.
  - Todos estos patrones favorecen la reutilización de código.
- Diferencias:
  - El patrón Adaptador otorga una interfaz con funcionalidades esperadas por sus clientes a un objeto mientras que el Decorador no dota una interfaz.
  - El patrón Decorador dota funciones adicionales dinámicamente a través de composición entre objetos mientras que el patrón Adaptador no extiende el comportamiento por composición sino que lo puede hacer o bien por herencia para clases (pero no sus hijos) o bien con delegación para objetos (clases y sus hijos).
  - El patrón Decorador busca ser una alternativa a la herencia para extender la funcionalidad de una clase mientras que el patrón Adaptador no.
  - El patrón Representante es una clase que actúa exclusivamente de intermediario entre dos objetos para controlar el acceso a un objeto mientras que en el Wrapper y en el Decorador tienen capacidad de añadir nuevas funcionalidades que no existían previamente.
  - El patrón Representante aplaza la creación de objetos pesados hasta que sea estrictamente necesario acceder a ellos mientras que en el Wrapper siempre se crea el objeto en cuestión cuando se usa.

Q2 Estrategia y Estado

- Similitudes:
  - Ambos patrones cambian en cierta medida el comportamiento de una clase.
  - Evitan el uso excesivo de instrucciones condiciones en el código.
- Diferencias:
  - El patrón Estrategia modifica el comportamiento de un método mientras que el Estado modifica un conjunto de métodos los cuales dependen del estado del mismo.
  - Las estrategias del patrón Estrategia son independientes y conocidas por el cliente que y por ello puede cambiarse en tiempo de ejecución por el usuario, mientras que el patrón Estado modifica su comportamiento cuando cambia su estado de manera interna (el usuario está abstraído de dichos estados).

Q3 Mediador y Observador

- Similitudes:
  - Ambos patrones favorecen la distribución del comportamiento y el bajo acoplamiento.
  - Estos patrones tienen como objetivo gestionar a diferentes objetos a la vez.
- Diferencias
  - El patrón Mediador emplea un objeto que encapsula cómo un conjunto de objetos interaccionan entre sí mientras que en el patrón Observador no existe dicho objeto, sino que define una dependencia de uno a muchos entre objetos para que cuando un objeto cambia de estado (clase Observable), todos sus dependientes sean notificados y actualizados automáticamente (clases Observadores).
  - Cuando existe un cambio relevante en cualquier objeto, en el patrón Mediador este debe informar a la clase que actúa como mediador, mientras que en el patrón Observador, solo los objetos que sean Observables podrán notificar dicho cambio.

Fuentes

Diapositivas de la asignatura.

https://www.dofactory.com/net/design-patterns

2. Cliente de correo e-look

Para implementar dicho programa con las características solicitadas por el ejercicio, usaremos el **patrón Estrategia**.

La principal razón por la cual he elegido este patrón es porque nos permite cambiar el comportamiento del método before() durante el tiempo de ejecución y además porque el patrón nos permitirá extender fácilmente los diferentes algoritmos para before().

Durante la implementación del código, se ha creado una variable que contiene el tipo de filtro a aplicar a los emails (el método before()). Esta variable es del tipo **TipoFiltro**, el cual es una interfaz que especifica el método before(). Así, cuando se llame al método before() durante el método sort() , se llamará a la variable mencionada anteriormente.

Como es necesario poder cambiar el tipo de filtro para las diferentes implementaciones de **TipoFiltro**, se ha creado un método público setBefore(), el cual recibe como parámetros un objeto **TipoFiltro**.

Se han creado dos implementaciones de la interfaz **TipoFiltro**, una para ordenar por prioridad (de mayor a menor prioridad) llamada **BeforePorPrioridad**, y otra implementación para ordenar por fecha, de más nuevos a más viejos, llamada **BeforePorFecha**. Como consecuencia de de lo anterior, hemos tenido que cambiar la visibilidad de los atributos implicados de la clase Email a **protected** para poder realizar las comparaciones necesarias.

Finalmente, para comprobar el correcto funcionamiento del programa, se ha creado un test en JUnit donde se emplean ambas estrategias (implementaciones de **TipoInterfaz**). No obstante, debido a la necesidad de implementar dicho test, ha sido necesario cambiar la visibilidad de la estructura de almacenamiento de Emails en la clase **Mailbox** a **protected**.

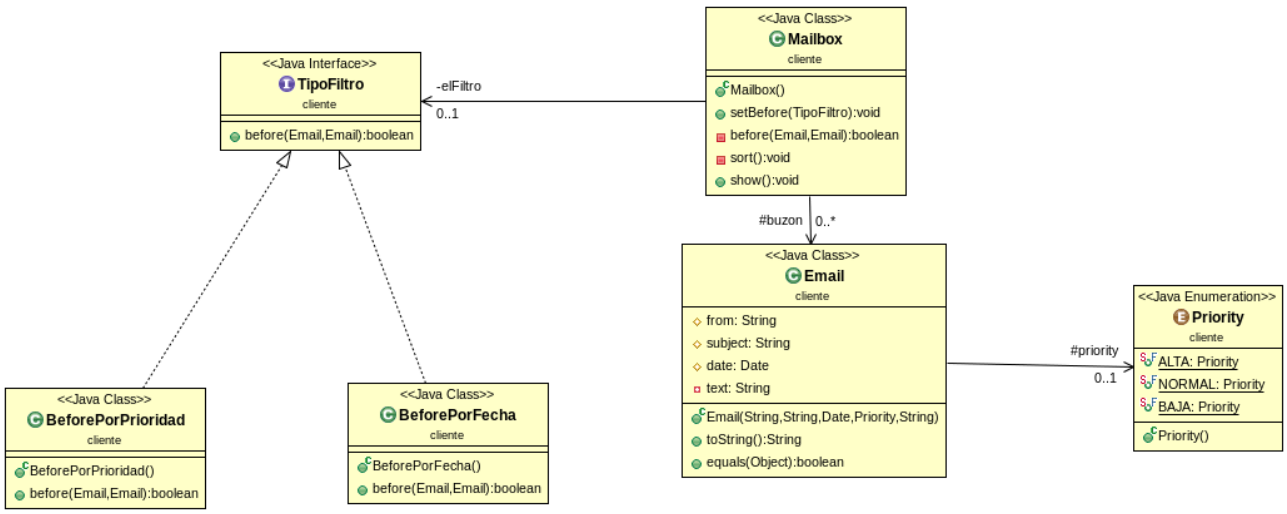

Diagrama UML de la implementación

Enlaces de interés

Github del proyecto: https://github.com/FlorinUMA/e-look