Explicación de implementaciones y decisiones

aquí se añadirán las explicaciones y la decisiones que vayamos tomando en cuanto al código

Entrega número 1:

Explicación Carpeta

Esta clase **Carpeta** representa un depósito de mensajes, que va a permitir organizar mensajes de una forma ordenada, asegurando métodos para agregar, eliminar, realizar listas y también buscar mensajes de una manera ordenada, sin alterar una modificación directa de los datos internos. Usamos atributos privados para encapsular el estado interno y así evitar modificaciones accidentales, validamos las entradas para impedir datos incorrectos, también funciones que permiten búsquedas avanzadas y prevenir errores, para proteger la lista original.

Explicación Servidor

En el servidor decidimos, aparte del método de registrar usuario, hacer también los métodos que se pedían en la interfaz (**enviar mensaje, recibir mensaje y listar mensajes**). Para que sea el que gestione todo, y luego la interfaz hacerla en otro módulo aparte, haciendo que importe la clase **ServidorCorreo** y que solamente llame a los métodos que contiene. Creemos que es lo correcto para hacer una interfaz.

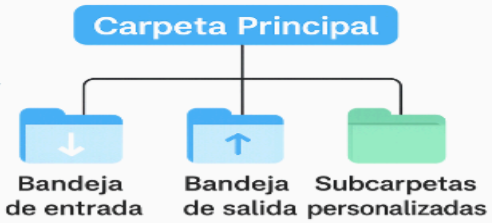
Para solucionar lo de llamar a la interfaz para bandeja de entrada o salida, decidimos hacer dos listar mensajes en **ServidorCorreo**, uno para bandeja de entrada y otro para bandeja de salida, para luego poder llamarlos en interfaz a cada uno por separado.

Corrección: Habíamos confundido una interfaz de menú (que ya removimos) con la interfaz abstracta requerida. Por lo que ya se integró eso en el servidor con todas las modificaciones correspondientes para que funcione el servidor con todo lo requerido.

Entrega número 2 :Infografia

ESTRUCTURA DEL ÁRBOL DE CARPETAS

CLIENTE DE CORREO



Concepto básico

- La clase Carpeta funciona como nodo y como árbol a la vez
- Cada usuario, al crearse, obtiene por defecto:
  - Una carpeta raíz: Carpeta Principal
  - Dos subcarpetas iniciales: Bandeja de Entrada y Bandeja de Salida

Características

- No es un árbol binario → cada nodo puede tener infinitos hijos
- Cada carpeta (nodo) contiene:
  - Una lista de mensajes
  - Un diccionario de subcarpetas (hijos)

INICIALIZACIÓN Y ESTRUCTURA DE LA CLASE CARPETA

La nueva clase Carpeta, implementada como un árbol general, se inicializa por defecto como nodo y árbol a la vez, para garantizar una mayor eficiencia en los métodos de búsqueda e inserción.
Cuenta con un diccionario para almacenar sus hijos, que representan las subcarpetas de la raíz.
Esta raíz se inicializa como "Carpeta principal" al momento de crear un usuario, junto con dos subcarpetas por defecto: "Bandeja de entrada" y "Bandeja de salida".
Cada nodo o subcarpeta tiene una lista donde almacena sus mensajes y un atributo con su nombre para identificarla.

FUNCIÓN DE LOS MÉTODOS

El metodo buscar_mensajes lo hace de forma recursiva con un algoritmo de busqueda en profundidad que primero comienza buscando en la raiz y luego va llamando y "analizando" recursivamente a sus subcarpetas en sus mensajes correspondientes que coincidan con el criterio de busqueda, como se requirio por asunto o remitente. Una vez que los encuentra mediante el metodo .extend() en cada llamada recursiva lo agrega a una lista de resultados.
A la que se retorna finalmente una vez que ya no hay mas recursion (o mensajes que coincidancon el criterio de busqueda).

Para el método mover_mensaje(), optamos por hacer tres métodos auxiliares antes para que sea más limpio el mover mensajes.
Los métodos auxiliares son: agregar_mensaje(), eliminar_mensaje(), que verifican que el mensaje a remover o eliminar sea una instancia de Mensaje, y mediante .append() y .remove(), agregan y eliminan el mensaje deseado.
Y el tercer método auxiliar, obtener_carpeta(), es para devolver la carpeta de origen donde está el mensaje a mover, para luego poder aplicar el método eliminar_mensaje() y reubicarlo en el destino deseado, o sea, moverlo.
Para encontrar la subcarpeta aplicamos la recursión, similar al método buscar_mensajes().

Para el método mover_mensaje() dimos por entendido que era necesario buscar recursivamente la carpeta de origen, pero también dimos por entendido que el usuario de antemano iba a proporcionar la carpeta de destino.
Sino, es imposible saber dónde quiere mover el mensaje.
Por lo tanto, la pasamos como parámetro junto al mensaje a mover en los parámetros del método mover_mensaje().
En el método mover_mensaje() primero verificamos que todos los datos sean instancias.
Después, obtenemos la carpeta de origen con obtener_carpeta(), luego eliminamos el mensaje de la lista de mensajes de la carpeta de origen y agregamos el mensaje al destino con agregar_mensaje(), en el destino proporcionado por el usuario.

Entrega número 3

Explicación nuevos métodos

Nuevo método en módulo Carpeta: `obtener_carpeta_padre()`

Lo que hace este método es primero mirar en las subcarpetas de la raíz; si no encuentra nada, llama recursivamente a los valores de las subcarpetas.
Busca la carpeta padre recursivamente; si encuentra algo que no es `None`, la retorna en la primera línea inmediatamente. Si recorre todos los subniveles y no encuentra nada, retorna `None`. En el caso límite, que no se encuentre, retorna `None`.

La recursión se ejecuta una y otra vez hasta encontrar el resultado. En el peor de los casos, recorre todo y no lo encuentra.
La complejidad es de **O(1)** en el mejor de los casos y **O(n)** en el peor, siendo *n* la cantidad de subcarpetas, ya que se recorre todo sin encontrar nada.

Utilizamos este método para poder implementar el método de crear una subcarpeta anidada en el módulo **Usuario** con el método: `crear_subcarpeta_anidada()`.
En este método, lo que se hace es primero verificar si la carpeta padre es la raíz, y luego, si no lo es, utilizar el método `obtener_carpeta_padre()` para encontrar la carpeta padre en el árbol, y ahí usar el método `agregar_subcarpeta()` e insertar la carpeta nueva en el lugar deseado.

Justificación y costo — Cola de prioridades

Elegimos utilizar el módulo **heapq** para implementar la cola de prioridades porque, a comparación de una lista común, al implementar un **heap binario** nos aseguramos de que las operaciones de inserción y extracción de mensajes tengan un costo/complejidad algorítmica de **O(log N)**.
Esto significa que si la lista es muy grande y sigue creciendo, el costo de las operaciones crece, pero a un valor muchísimo más bajo que lo haría con una lista normal, que tendría un costo mucho más elevado **O(N)**.

Para poder implementar la cola de prioridades en vez de una lista normal de mensajes en cada carpeta, en la clase **Mensaje** tuvimos que agregarle un parámetro/atributo por defecto que es la **prioridad**. Si es `0` (por defecto), es normal, y si es `1`, es urgente.
Esto hace que los mensajes urgentes se ubiquen más cerca del índice 0 en la nueva lista (cola de prioridad).

Después, en la clase **Servidor**, se determina que si el asunto del mensaje contiene la palabra “urgente”, se le cambia la prioridad al mensaje pasando a ser `prioridad = 1`; y si no contiene la palabra “urgente”, sigue como normal con `prioridad = 0`.
Con esto, sin importar en qué carpeta esté, si es urgente, siempre va a tener prioridad a la hora de ser ordenado en la cola de prioridades.

Después del filtrado, el mensaje va a ir a una carpeta específica según las reglas definidas por el usuario, y si no coincide con ninguna regla, va por defecto a la **bandeja de entrada**, donde igualmente seguirá ordenado porque es una instancia de **Carpeta**.

Implementación en Carpeta

En la clase **Carpeta**, en el método `agregar_mensaje()`, el mensaje se almacena como una tupla con su prioridad y el mensaje.
Hacemos que si la prioridad es `1`, al momento de agregar el mensaje a la heap binaria sea `-1`, para que esté más cerca de la raíz, ya que el módulo **heapq** ordena a los hijos con prioridad más baja más cerca de la raíz.
Esto lo implementamos con **`heapq.heappush()`**, que realiza ese "filtrado" de mensaje con menor prioridad (numéricamente) hacia arriba en la raíz.
Como nosotros pusimos `-1` en los urgentes, esos quedan más arriba, y los que no son urgentes (`0`, que es mayor a `-1`) quedan más abajo.

En el método **listar_mensajes()**, primero se hace una copia de la lista de mensajes. Luego se itera sobre esa lista, y en cada iteración se aplica el algoritmo **`heappop()`**, que tiene un costo de **O(N log N)**.
Este algoritmo "saca" y coloca en la lista a mostrar el mensaje de mayor prioridad (`-1`, urgente).
Cada vez que lo hace, reorganiza el heap para colocar como raíz al hijo de mayor prioridad, lo saca, vuelve a ordenar, y así sucesivamente, con un costo de **O(log N)** por cada operación.

Por último, se muestra la lista de mensajes ordenada por prioridad.
N es la cantidad de veces que "saca y reemplaza" el algoritmo, y **log N** es el costo de cada operación individualmente.

Explicación filtros

El objetivo de implementar el filtrado de mensajes que vamos a hacer es el siguiente:
Que el usuario reciba un mensaje y, si contiene una palabra clave en el asunto, vaya a una carpeta específica o no.
Y si no va a una carpeta en específico, se va directamente a la **bandeja de entrada por defecto**.
Todo esto siempre manteniendo la **prioridad** de un mensaje en la lista de mensajes de cada subcarpeta si es "urgente", ya que la **cola de prioridades** está implementada para todas las listas de mensajes de las subcarpetas que son objetos de la clase **Carpeta**.

La clase **Usuario** va a tener una lista vacía en el constructor, donde se van a guardar las tuplas que contienen *(asunto, destino)*.
Es decir, si el mensaje tiene determinada palabra en el asunto, va a esa carpeta.
En el **caso límite**, si la carpeta no existe o el mensaje recibido no aplica a ningún filtro, simplemente va a la **bandeja de entrada por defecto**.

Para crear la regla de filtrado hicimos el método **`creacion_regla_de_filtro()`**, que toma como parámetro una palabra “clave” del asunto y una carpeta de destino.
Se guarda como tupla y se almacena en la lista de filtros.

Para aplicar el filtro, hicimos un método auxiliar (**`aplicar_filtro()`**) que, si el mensaje a filtrar coincide con una regla de filtrado, **retorna el nombre** de la carpeta de destino para que otro método se encargue de la lógica de movimiento.
Si no coincide, **devuelve None** para que el otro método lo mueva por defecto a la bandeja de entrada.
Para hacer esto, se **itera sobre la lista de tuplas** (reglas de filtrado).

El **análisis de complejidad** de `aplicar_filtro()` es de **O(n)**, ya que para obtener la carpeta de destino dependiendo de la palabra clave en el asunto, depende de la cantidad de tuplas a recorrer que haya en la lista de reglas de filtrado.

Para filtrar el mensaje, en el método **`filtrar_mensaje()`** primero se aplica el método auxiliar `aplicar_filtro()` para que devuelva el nombre de la carpeta de destino o `None`.
Si devuelve el nombre de la carpeta de destino (previamente definida en la regla de filtrado), se aplica la **lógica de movimiento del mensaje**;
y si devuelve `None`, por defecto se guarda el mensaje en la **bandeja de entrada**.

También cambiamos en el **ServidorCorreo** el método **`enviar_mensaje()`**.
El único cambio que hicimos es que, para el destinatario, se aplique el filtro cuando recibe el mensaje.
Antes, lo recibía directamente en la bandeja de entrada.