Trabajo Final para la Cátedra: Estructuras de Datos en Python

Tema: Cliente de Correo Electrónico (Email Client)

Grupo 52

Integrantes: **Facundo Almada**
Nahuel Escobar

Características Implementadas

Encapsulamiento

- Atributos privados con doble guión bajo (`__`)
- Propiedades (`@property`) para acceso controlado
- Métodos específicos para operaciones seguras

Clases Principales

1. **Mensaje**: Encapsula toda la información de un email
2. **Carpeta**: Organiza mensajes con operaciones de búsqueda
3. **Usuario**: Gestiona carpetas y credenciales
4. **ServidorCorreo**: Núcleo del sistema, gestiona usuarios y envíos

Funcionalidades Clave

Envío y recepción de mensajes

Sistema de carpetas con carpetas predeterminadas

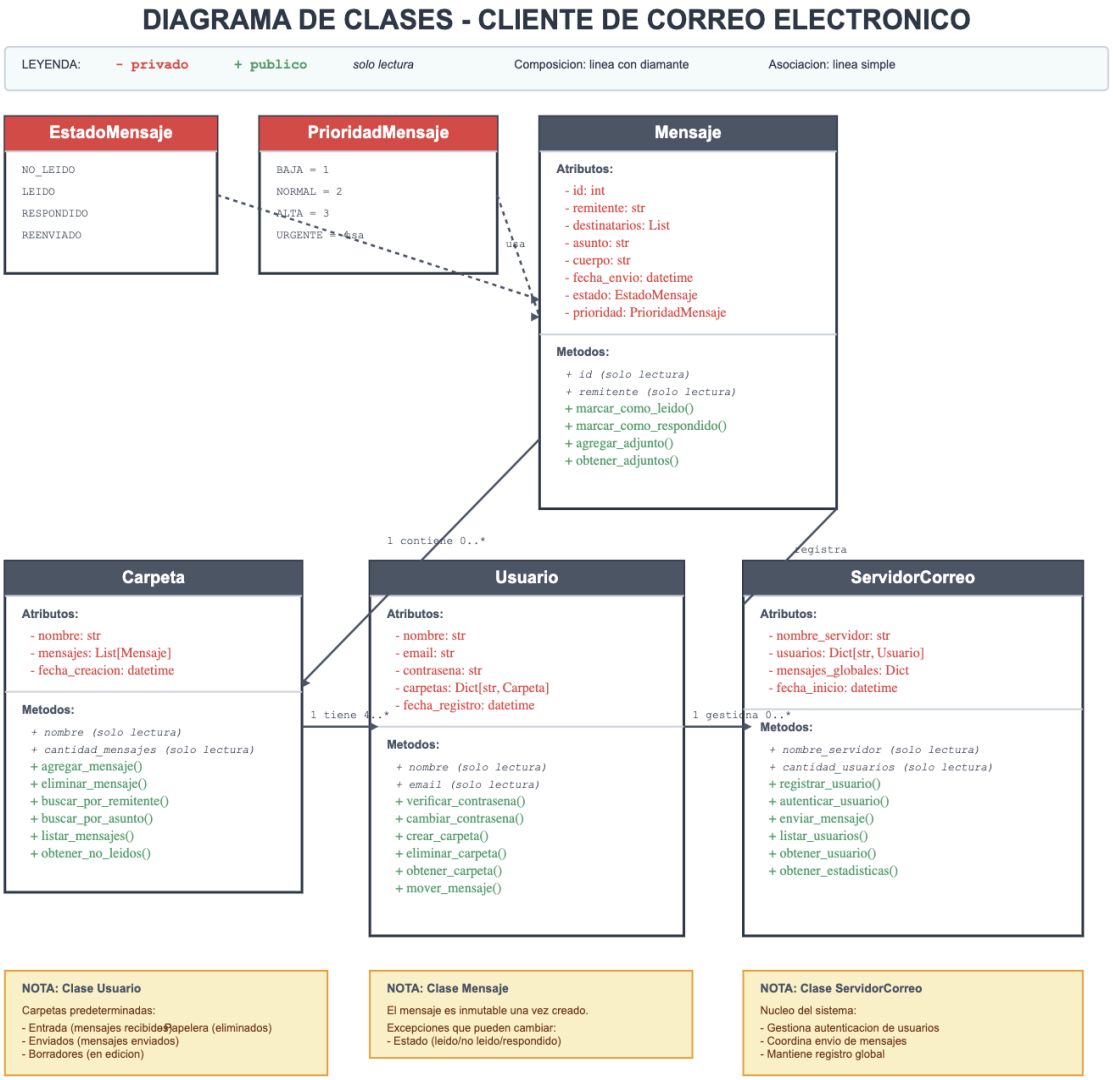
Autenticación de usuarios

Búsqueda por remitente y asunto

Estados de mensajes (leído, no leído, etc.)

Prioridades de mensajes

Movimiento de mensajes entre carpetas



Justificación de Decisiones de Diseño

1. Arquitectura General

1.1 Patrón de Diseño Elegido

Decisión: Implementación basada en el patrón **Composition** con **Single Responsibility Principle**.

Justificación:

- Cada clase tiene una responsabilidad específica y bien definida
- `ServidorCorreo` actúa como **Facade** para coordinar operaciones complejas
- La composición permite flexibilidad y reutilización de componentes
- Facilita el testing unitario y el mantenimiento del código

1.2 Separación de Responsabilidades

Clase	Responsabilidad Principal	Justificación
`Mensaje`	Encapsular datos del email	Representa la entidad fundamental, inmutable por diseño
`Carpeta`	Organizar y gestionar mensajes	Abstrae la lógica de almacenamiento y búsqueda
`Usuario`	Gestionar identidad y carpetas personales	Centraliza la información del usuario y sus datos
`ServidorCorreo`	Coordinar operaciones del sistema	Punto de entrada único, gestiona la lógica de negocio

2. Encapsulamiento y Seguridad

2.1 Atributos Privados

Decisión: Uso de doble guión bajo (`__atributo`) para todos los datos sensibles.

Justificación:

- **Seguridad**: Previene modificaciones accidentales o maliciosas
- **Integridad**: Garantiza que los datos se modifiquen solo a través de métodos controlados
- **Mantenibilidad**: Cambios internos no afectan el código cliente

2.2 Propiedades (Properties)

Decisión: Implementación de propiedades con `@property` para acceso controlado.

Justificación:

python

```python
@property
def email(self) -> str:
    return self.__email  # Solo lectura, no se puede modificar

@property
def estado(self) -> EstadoMensaje:
    return self.__estado

@estado.setter
def estado(self, nuevo_estado: EstadoMensaje):
    if isinstance(nuevo_estado, EstadoMensaje):
        self.__estado = nuevo_estado  # Validación antes de asignar
```

Beneficios:

- Validación de datos en tiempo de asignación
- Interfaz limpia y consistente
- Flexibilidad para agregar lógica adicional sin cambiar la API

3. Manejo de Estados y Tipos

3.1 Enumeraciones (Enum)

Decisión: Uso de `EstadoMensaje` y `PrioridadMensaje` como enumeraciones.

Justificación:

- **Type Safety**: Previene valores inválidos
- **Legibilidad**: Códigos más expresivos que números mágicos
- **Mantenibilidad**: Cambios centralizados en un solo lugar
- **IDE Support**: Autocompletado y detección de errores

3.2 Type Hints

Decisión: Anotaciones de tipo completas en toda la API pública.

Justificación:

python

```
def buscar_por_remitente(self, remitente: str) -> List[Mensaje]:
    """Búsqueda tipada y documentada"""
    return [msg for msg in self.__mensajes if remitente.lower() in msg.remitente.lower()]
```

Beneficios:

- Mejor experiencia de desarrollo
- Documentación implícita del código
- Detección temprana de errores
- Facilita refactoring seguro

4. Gestión de Datos y Estructuras

4.1 Inmutabilidad Selectiva

Decisión: Los mensajes son mayormente inmutables después de la creación.

Justificación:

- **Integridad**: Un email enviado no debería poder alterarse
- **Auditabilidad**: Preserva el historial de comunicaciones
- **Concurrencia**: Reduce problemas de concurrencia futura
- **Excepción controlada**: Solo `estado` y `adjuntos` son modificables por razones de UX

4.2 Copias Defensivas

Decisión: Los métodos que retornan listas devuelven copias, no referencias.

Justificación:

python
```
def listar_mensajes(self) -> List[Mensaje]:
    return self.__mensajes.copy()  # Evita modificaciones externas

@property
def destinatarios(self) -> List[str]:
    return self.__destinatarios.copy()  # Lista inmutable desde exterior
```

Beneficios:

- Previene modificaciones accidentales
- Mantiene la integridad del estado interno
- Sigue el principio de menor privilegio

5. Arquitectura de Carpetas

5.1 Carpetas Predeterminadas

Decisión: Creación automática de carpetas básicas (`Entrada`, `Enviados`, `Borradores`, `Papelera`).

Justificación:

- **Usabilidad**: Experiencia familiar para usuarios de email
- **Funcionalidad**: Separación lógica de tipos de mensajes
- **Extensibilidad**: Base para funcionalidades futuras (filtros, reglas)

5.2 Protección de Carpetas del Sistema

Decisión: Las carpetas predeterminadas no pueden eliminarse.

Justificación:

python
```
def eliminar_carpeta(self, nombre: str) -> bool:
    carpetas_protegidas = ["Entrada", "Enviados", "Borradores", "Papelera"]
    if nombre in self.__carpetas and nombre not in carpetas_protegidas:
        del self.__carpetas[nombre]
        return True
    return False
```

Beneficios:

- API predecible y consistente
- Fácil verificación de operaciones
- No requiere manejo de excepciones para casos normales

7.2 Validación de Tipos

Decisión: Validación explícita en métodos críticos.

Justificación:

python

```
def agregar_mensaje(self, mensaje: Mensaje):
    if isinstance(mensaje, Mensaje):
        self.__mensajes.append(mensaje)
    else:
        raise ValueError("Solo se pueden agregar objetos de tipo Mensaje")
```

Beneficios:

- Falla rápido con errores claros
- Previene corrupción de datos
- Facilita debugging

8. Consideraciones de Performance

8.1 Búsquedas Lineales

Decisión: Búsquedas O(n) en listas para la Entrega 1.

Justificación:

- **Simplicidad**: Implementación directa y comprensible
- **Suficiencia**: Adecuado para volúmenes pequeños de desarrollo
- **Evolución**: Base para optimizaciones futuras (índices, hash maps)

8.2 Almacenamiento en Memoria

Decisión: Toda la información se mantiene en memoria.

Justificación:

- **Enfoque Académico**: Prioriza estructura OOP sobre persistencia
- **Simplicidad**: Evita complejidad de base de datos
- **Prototipado Rápido**: Permite iteración rápida en diseño

9. Extensibilidad y Mantenimiento

9.1 Interfaces Consistentes

Decisión: Métodos con nombres y patrones similares entre clases.

Beneficios:

- `buscar_por_*()` en `Carpeta`
- `obtener_*()` para recuperación de datos
- `listar_*()` para colecciones

9.2 Separación de Concerns

Decisión: Lógica de presentación separada de lógica de negocio.

Preparación para:

- Múltiples interfaces (CLI, GUI, Web)
- Testing automatizado
- Integración con sistemas externos

Conclusión

El diseño prioriza **claridad**, **mantenibilidad** y **extensibilidad** mientras demuestra sólidos principios de programación orientada a objetos. La arquitectura está preparada para evolucionar en las siguientes entregas con estructuras de datos más complejas y algoritmos avanzados.