

Del problema al diseño

Análisis orientado a objetos

1. Introducción

A lo largo del curso aprendimos los principios del diseño orientado a objetos y los patrones de diseño. Pero todavía no respondimos una pregunta fundamental: **¿cómo se pasa de un problema del mundo real a un diseño de clases?**

La respuesta intuitiva de muchos programadores principiantes es “empezar a codificar”. Se lee el enunciado, se imagina una solución, se abre el editor y se empieza a escribir. Este enfoque puede funcionar para problemas pequeños, pero falla estrepitosamente en cuanto la complejidad crece: aparecen clases que no deberían existir, se mezclan responsabilidades, los cambios en un lugar rompen código en otro, y el sistema se vuelve imposible de mantener.

La alternativa profesional es **dedicar tiempo al análisis y al diseño antes de escribir código**. No se trata de hacer un diseño perfecto y congelarlo para siempre, sino de pensar la estructura del sistema con la suficiente claridad como para no construir sobre arena.

Este anexo recorre el proceso completo, desde un documento de requerimientos hasta la implementación en **Python**. Vamos a trabajar con un ejemplo concreto: un sistema de mensajería interna. En el camino usaremos las técnicas de análisis orientado a objetos que vimos en los apuntes previos y algunos diagramas UML para comunicar el diseño.

Al final del documento mencionamos el repositorio del proyecto, donde vive el código completo que surge de este análisis.

2. El documento de requerimientos

Todo sistema de software nace de una necesidad. Esa necesidad se expresa en un **documento de requerimientos**, que describe qué debe hacer el sistema, sin entrar en cómo debe construirse. En un proyecto real, este documento surge de entrevistas con los usuarios y de un análisis detallado del dominio. Aquí partimos de un enunciado simplificado, escrito específicamente para este anexo.

2.1. Enunciado

Se necesita un sistema de mensajería interna para una organización pequeña. El sistema funciona de la siguiente manera:

- Cualquier persona de la organización puede tener una **cuenta** en el sistema. Cada cuenta tiene un nombre de usuario y una contraseña.
- Existen dos tipos de cuentas: **usuarios** y **administradores**. Un usuario puede enviar y recibir mensajes. Un administrador gestiona las cuentas, pero no envía ni recibe mensajes.
- Un **mensaje** tiene un emisor, un receptor, un asunto, un cuerpo y una fecha de envío. Un mensaje puede estar leído o no leído.
- El sistema debe permitir a un usuario **enviar un mensaje** a otro usuario indicando asunto y cuerpo.
- El sistema debe permitir a un usuario **ver los mensajes que recibió y ver los mensajes que envió**.
- El sistema debe permitir a un usuario **leer un mensaje** específico. Al leerlo, el mensaje pasa a estar marcado como leído.
- El sistema debe permitir a un administrador **dar de alta** y **dar de baja** usuarios, y **listar** los usuarios existentes.
- El sistema no se apaga al cerrar sesión. Cada usuario se autentica, realiza sus operaciones y cierra su sesión. Después el sistema queda disponible para el próximo usuario.

A partir de este enunciado vamos a construir el diseño del sistema. Recorreremos el proceso en dos fases: la **fase de análisis**, donde identificamos los elementos del dominio, y la **fase de diseño**, donde organizamos esos elementos en un modelo de clases con relaciones bien definidas.

3. Análisis: identificar clases

La primera tarea consiste en identificar los **conceptos** del dominio que van a convertirse en clases del sistema. Una técnica clásica y muy efectiva consiste en subrayar los **sustantivos y frases nominales** del documento de requerimientos. Cada uno de ellos es un *candidato* a clase, aunque algunos terminarán siendo atributos de otra clase y otros serán descartados.

Nota: los sustantivos son candidatos, no verdades

No todo sustantivo se convierte en clase. La lista que armamos es un punto de partida, no un resultado. A medida que avancemos con el análisis, vamos a decidir cuáles son clases, cuáles son atributos y cuáles se descartan por no ser relevantes para el sistema.

3.1. Lista de candidatos

Extrayendo los sustantivos y frases nominales del enunciado, obtenemos la siguiente lista:

Candidato	Decisión
Organización	Descartado (es el contexto, no una entidad interna).
Sistema	Clase (el orquestador).
Cuenta	Clase abstracta (superclase de Usuario y Admin).
Usuario	Clase (subclase de Cuenta).
Admin	Clase (subclase de Cuenta).
Nombre de usuario	Atributo de Cuenta .
Contraseña	Atributo de Cuenta .
Mensaje	Clase.

Candidato	Decisión
Emisor	Atributo de Mensaje (referencia a Usuario).
Receptor	Atributo de Mensaje (referencia a Usuario).
Asunto	Atributo de Mensaje .
Cuerpo	Atributo de Mensaje .
Fecha de envío	Atributo de Mensaje .
Leído / no leído	Atributo booleano de Mensaje .
Mensajes recibidos	Colección derivada (no es clase, es una consulta).
Mensajes enviados	Colección derivada (no es clase, es una consulta).
Alta / baja de usuario	Operación del Sistema (no es sustantivo).

3.2. Clases resultantes

Aplicando los criterios de decisión, las clases del dominio son:

- **De cuentas:** **Cuenta** (abstracta), **Usuario**, **Admin**.
- **De contenido:** **Mensaje**.
- **Del sistema:** **Sistema**, que orquesta todo el funcionamiento.

Son cinco clases. Todo lo que el enunciado menciona como sustantivo relevante tiene su lugar: los tres tipos de cuenta forman la jerarquía, el mensaje es una entidad con sus propios datos, y el sistema es el punto de entrada que coordina.

4. Análisis: identificar atributos

Una vez identificadas las clases, el siguiente paso es determinar sus **atributos**. Los atributos se extraen de las **palabras y frases descriptivas** que acompañan a cada sustantivo en el documento.

4.1. Atributos por clase

Clase	Atributos
Cuenta	nombreUsuario, contrasenia
Usuario	(hereda los de Cuenta)
Admin	(hereda los de Cuenta)
Mensaje	emisor, receptor, asunto, cuerpo, fechaEnvio, leído
Sistema	cuentas, mensajes

4.2. Observaciones

Algunos detalles que vale la pena notar:

- **emisor** y **receptor** en **Mensaje** son referencias a objetos de la clase **Usuario**, no strings con nombres. Esto permite navegar del mensaje al usuario y viceversa sin ambigüedad.
- **leído** es un atributo booleano que arranca en **false** y se pone en **true** cuando el receptor abre el mensaje.
- El **Sistema** mantiene dos colecciones: la lista de todas las cuentas registradas, y la lista de todos los mensajes enviados. Son los dos conjuntos de datos que el sistema administra.
- El **Sistema** no guarda el usuario actual como atributo. La sesión se maneja como un dato temporal, que existe mientras dura la operación en curso. Guardar “el usuario actual” como estado permanente introduce una dependencia temporal que complica el diseño: obliga a recordar limpiarlo al cerrar sesión, y hace que dos operaciones simultáneas no puedan coexistir.

5. Análisis: identificar métodos

Los **métodos** de cada clase se derivan de los **verbos y frases verbales** del documento. Cuando el enunciado dice “el sistema debe permitir a un usuario enviar un mensaje”, el

verbo “enviar” se convierte en una operación.

5.1. Verbos y sus clases

Frase verbal	Clase responsable
Iniciar sesión	Sistema
Cerrar sesión	Sistema
Enviar un mensaje	Sistema
Ver mensajes recibidos	Sistema
Ver mensajes enviados	Sistema
Leer un mensaje	Sistema
Dar de alta un usuario	Sistema
Dar de baja un usuario	Sistema
Listar usuarios	Sistema
Guardar un mensaje	Sistema
Buscar mensajes recibidos	Sistema
Buscar mensajes enviados	Sistema
Agregar una cuenta	Sistema
Eliminar una cuenta	Sistema
Buscar una cuenta	Sistema
Marcar un mensaje como leído	Mensaje
Verificar credenciales	Cuenta

5.2. Métodos resultantes por clase

- **Sistema:** un único método público `ejecutar()`, que contiene el bucle principal. Todos los demás son privados:
 - `iniciarSesion()`
 - `mostrarMenu(cuenta)`
 - `crearCuenta()`

- `borrarCuenta(cuenta)`
 - `nuevoMensaje(emisor)`
 - `leerMensaje(receptor)`
 - `verEnviados(emisor)`
- **Cuenta:** `comprobarUsuario(username, contrasenia)` y `obtenerOpciones()` (abstracto).
 - **Usuario:** `obtenerOpciones()`.
 - **Admin:** `obtenerOpciones()`.
 - **Mensaje:** `leer()` (marca como leído y devuelve el texto), `estaLeido()`, `esPara(u)`, `esDe(u)`.

Nota: qué exponer como público

No todos los métodos de una clase son públicos. En el **Sistema** solo `ejecutar()` es público: es el único punto de entrada al programa. Todos los demás métodos los invoca internamente el bucle principal.

Regla práctica: **exponer lo mínimo necesario** para que la clase cumpla con su rol en el sistema. Todo lo demás queda oculto, en línea con el principio de encapsulamiento.

6. Análisis: identificar relaciones

Las clases no viven aisladas: se relacionan entre sí. En el diseño orientado a objetos, las relaciones se clasifican en tres grandes grupos:

- **Asociación:** una clase “tiene” o “usa” a otra.
- **Composición:** una clase está formada por otras y su ciclo de vida depende de la primera.
- **Herencia:** una clase es un caso particular de otra.

6.1. Relaciones identificadas

Clase A	Relación	Clase B
Usuario	es una	Cuenta
Admin	es una	Cuenta
Sistema	contiene a	Cuenta
Sistema	contiene a	Mensaje
Mensaje	tiene como emisor a	Usuario
Mensaje	tiene como receptor a	Usuario

6.2. La jerarquía de cuentas

Las dos clases **Usuario** y **Admin** tienen en común los atributos **nombreUsuario** y **contrasenia**, además de la operación **comprobarUsuario()**. Esto sugiere modelar una **superclase abstracta** **Cuenta** de la que heredan las dos subclases. Cada subclase agrega su propio comportamiento:

- **Usuario** agrega la capacidad de operar con mensajes.
- **Admin** agrega la capacidad de gestionar cuentas.

Esta jerarquía tiene dos ventajas: elimina la duplicación de los atributos comunes, y permite tratar a cualquier tipo de cuenta de manera polimórfica cuando solo importa la parte común (por ejemplo, durante la autenticación o al mostrar el menú principal).

6.3. Las colecciones del Sistema

El **Sistema** contiene dos colecciones: la lista de cuentas y la lista de mensajes. Se modelan como **composición** (rombo lleno) porque las colecciones se crean junto con el sistema y desaparecen con él. El ciclo de vida está acoplado.

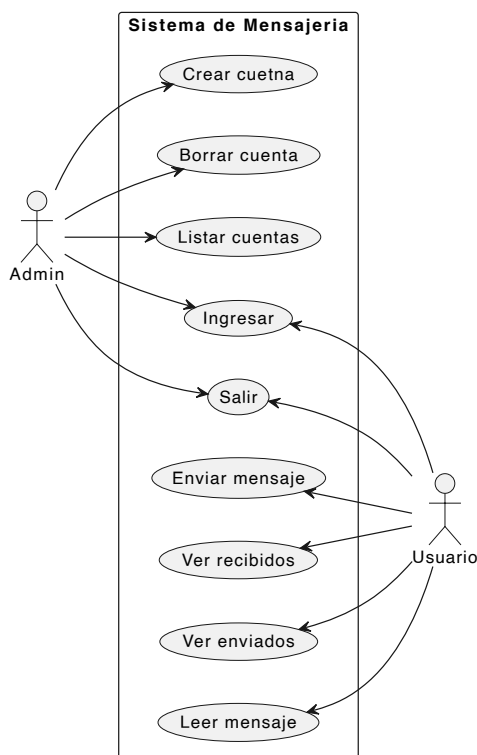
7. Diagrama de casos de uso

Antes de entrar en el diagrama de clases, conviene representar las **interacciones entre los usuarios y el sistema**. Para eso se usa el **diagrama de casos de uso**, que muestra qué actores interactúan con el sistema y qué acciones realizan.

7.1. Actores y casos de uso

Los actores del sistema son dos:

- **Usuario:** ingresa, envía mensajes, ve sus mensajes recibidos, ve sus mensajes enviados, lee un mensaje, sale.
- **Admin:** ingresa, da de alta usuarios, da de baja usuarios, lista usuarios, sale.



7.2. El sistema como servicio

El sistema no tiene un caso de uso "Apagar". Una vez que arranca, corre indefinidamente. Cada cuenta que se autentica pasa por el ciclo ingresar → operar → salir, y cuando sale, el sistema queda esperando a la siguiente cuenta. Nunca deja de estar disponible.

Esta forma de funcionamiento es la habitual en sistemas reales: una aplicación de gestión no se cierra cada vez que un usuario termina su sesión, sino que queda esperando al siguiente. En el diagrama de clases vamos a ver cómo se modela esa idea: el método principal del **Sistema** es un ciclo continuo que en cada vuelta atiende a un usuario distinto.

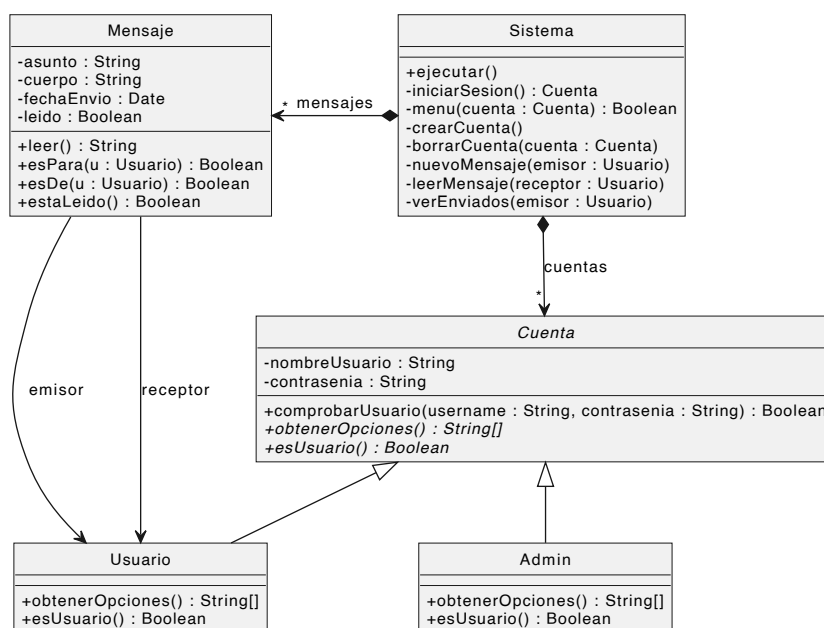
8. Primer diagrama de clases

Con toda la información recolectada en las fases anteriores, construimos el **diagrama de clases** del sistema. Este diagrama muestra las clases, sus atributos, sus métodos, y las relaciones entre ellas.

8.1. Estructura

El diagrama incluye:

- La **jerarquía de cuentas**: **Cuenta** como superclase abstracta, y **Usuario**, **Admin** como subclases.
- La clase **Mensaje**, con sus atributos y operaciones básicas.
- La clase **Sistema**, que orquesta todo y contiene las dos colecciones.



8.2. Lectura del diagrama

Algunos detalles de notación que conviene recordar al leer el diagrama:

- Los atributos y métodos con + son públicos, los que tienen - son privados.
- La flecha con triángulo hueco indica herencia: apunta siempre a la superclase.
- El rombo relleno indica composición: la clase del lado del rombo “contiene” a la otra.
- La línea simple con número en los extremos indica asociación con su multiplicidad.
- Los tipos en notación UML son genéricos (**String**, **Integer**, **Boolean**, **Date**), no específicos de **Python**. La traducción a tipos concretos se hace en la etapa de implementación.

8.3. El Sistema como única puerta de entrada

Un detalle importante del diagrama es que el **Sistema** tiene un único método público: **ejecutar()**. Todo lo demás son métodos privados que solo el bucle principal invoca. Esto significa que, desde afuera del sistema, la única operación posible es “arrancalo”.

Esta decisión tiene consecuencias interesantes. Por un lado, hace imposible que código externo altere el estado interno sin pasar por el bucle. Por otro, hace que el **Sistema** cargue con toda la responsabilidad de las operaciones.

9. Una iteración: el Sistema tiene demasiada responsabilidad

El diseño que obtuvimos en la sección anterior funciona. Es coherente con el enunciado, respeta la jerarquía de cuentas, y cada clase tiene su rol. Pero antes de pasar al código conviene detenerse en una observación.

9.1. El problema

Miremos el **Sistema** con atención. Además del bucle principal, contiene siete métodos privados:

- `iniciarSesion()`
- `mostrarMenu(cuenta)`
- `crearCuenta()`
- `borrarCuenta(cuenta)`
- `nuevoMensaje(emisor)`
- `leerMensaje(receptor)`
- `verEnviados(emisor)`

Y dos colecciones como atributos: la lista de cuentas y la lista de mensajes.

Cada uno de esos métodos hace varias cosas. Por ejemplo, `nuevoMensaje` tiene que: obtener la lista de usuarios destinatarios, mostrar un menú, pedir asunto y cuerpo al usuario, crear el mensaje, y guardarlo. Cada una de esas cosas involucra manipular alguna de las dos colecciones.

El **Sistema** no solo orquesta el flujo. También sabe cómo buscar dentro de las listas, cómo filtrarlas, cómo agregar elementos, cómo eliminarlos. Es responsable de demasiadas cosas distintas.

Esto viola el **Principio de Responsabilidad Única** (SRP): una clase debería tener un solo motivo para cambiar. Pero el **Sistema** va a cambiar cada vez que:

- Se agregue una operación al menú.
- Cambie la forma de buscar una cuenta o un mensaje.
- Se modifique la estructura interna de las colecciones.
- Se agregue una validación sobre quién puede hacer qué cosa.

Son demasiados motivos para una sola clase.

9.2. La solución: delegar en gestores

La solución es aplicar un principio que ya conocemos del curso: **si una clase tiene demasiada responsabilidad, se la divide**. En este caso, hay dos responsabilidades que se pueden extraer del **Sistema**:

- La gestión de la colección de cuentas (buscar, agregar, eliminar, listar).
- La gestión de la colección de mensajes (guardar, filtrar por emisor, filtrar por receptor).

Cada una de esas responsabilidades se encapsula en su propia clase: un **GestorDeUsuarios** y un **GestorDeMensajes**. El **Sistema** deja de manipular las listas directamente y les pide a los gestores lo que necesita.

El **Sistema** ya no sabe cómo se guardan las cuentas ni los mensajes. Solo sabe que puede pedirle al gestor una lista de usuarios, y que puede pedirle al otro gestor que guarde un mensaje. Los detalles quedan encapsulados.

9.3. Los gestores como fachadas

Este tipo de clase que agrupa operaciones sobre una colección tiene un nombre en el catálogo de patrones: se llama **Façade**, y ya lo vimos en el apunte de patrones de diseño.

Recordemos qué hace una fachada: ofrece una **interfaz simplificada** para un conjunto complejo de operaciones. En lugar de que el cliente tenga que conocer cómo se guardan los datos, cómo se recorren, cómo se filtran, la fachada le da un puñado de métodos con nombres claros.

En nuestro caso, el **GestorDeMensajes** expone tres operaciones simples:

- **agregarMensaje(m)**: guarda un mensaje nuevo.
- **mensajesPara(u)**: devuelve los mensajes recibidos por un usuario.
- **mensajesDe(u)**: devuelve los mensajes enviados por un usuario.

El **Sistema** no necesita saber que por debajo hay una lista, que se recorre con un bucle, que se filtra con una condición. Solo necesita saber que puede pedir estas tres cosas.

La ventaja es doble:

1. **Encapsulamiento.** Las colecciones ya no están expuestas. Viven dentro de sus gestores, que controlan cómo se accede a ellas.
2. **Posibilidad de cambiar la implementación.** Si mañana los mensajes se guardarán en un archivo o en una base de datos, alcanzaría con modificar el **GestorDeMensajes**. El **Sistema** no se enteraría del cambio, porque sigue hablando con los mismos métodos.

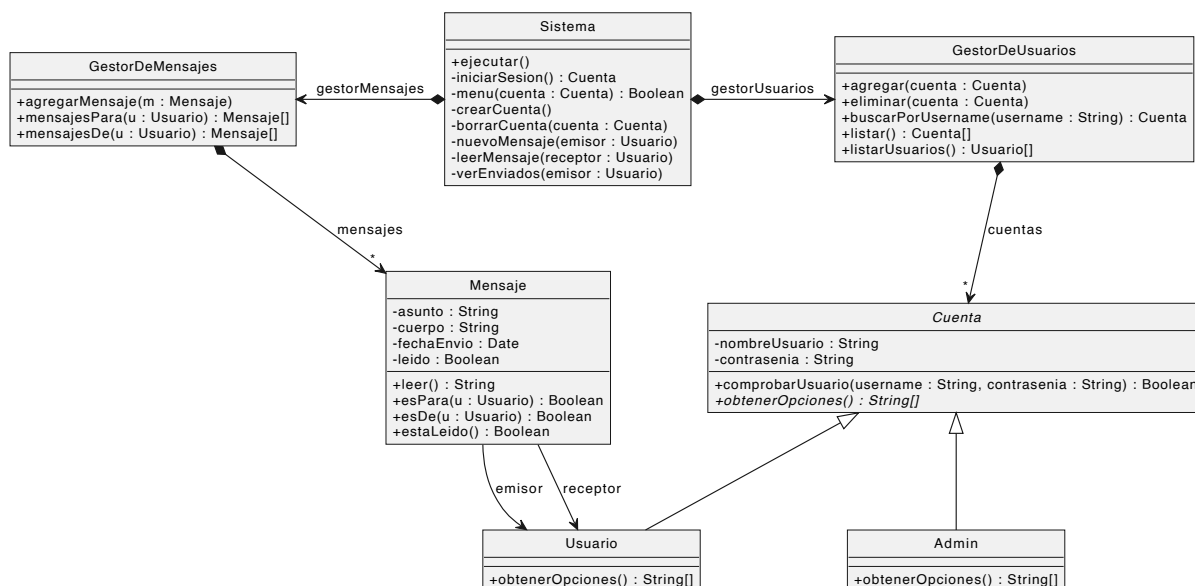
Nota: los gestores no están en el enunciado

Ni el **GestorDeUsuarios** ni el **GestorDeMensajes** aparecen mencionados en el documento de requerimientos. Nadie que use el sistema dice “voy a usar el gestor de mensajes”; dicen “voy a enviar un mensaje”.

Son **decisiones de diseño** que tomamos al modelar. Aparecen porque nos ayudan a organizar mejor el código, no porque existan en el mundo real. Esto es normal.

9.4. Diagrama de clases refinado

Con los dos gestores incorporados, el diagrama de clases queda así:



Los cambios respecto del diagrama anterior son:

- Se agregan dos clases: **GestorDeUsuarios** y **GestorDeMensajes**.
- El **Sistema** ya no tiene las colecciones como atributos. Ahora tiene dos gestores.
- Las colecciones se mueven: la lista de cuentas vive dentro del **GestorDeUsuarios**, y la lista de mensajes dentro del **GestorDeMensajes**.
- Los gestores exponen métodos específicos para operar sobre sus colecciones.

9.5. Lo que aprendimos con esta iteración

Este pequeño *refactor* muestra algo importante sobre el proceso de diseño: **el primer diseño no es el definitivo**. A medida que se avanza y se piensa mejor cada clase, aparecen oportunidades de mejora.

En este caso, la mejora vino de aplicar un principio que ya conocíamos (SRP) y un patrón que ya habíamos estudiado (Façade). No hizo falta inventar nada nuevo: solo reconocer que el **Sistema** estaba haciendo más de lo que le correspondía, y delegar las responsabilidades que se podían separar.

El diseño final sigue haciendo lo mismo que el inicial: un usuario puede enviar mensajes, leerlos, ver los enviados; un admin puede crear y borrar cuentas. Pero ahora cada clase tiene una responsabilidad más acotada y clara.

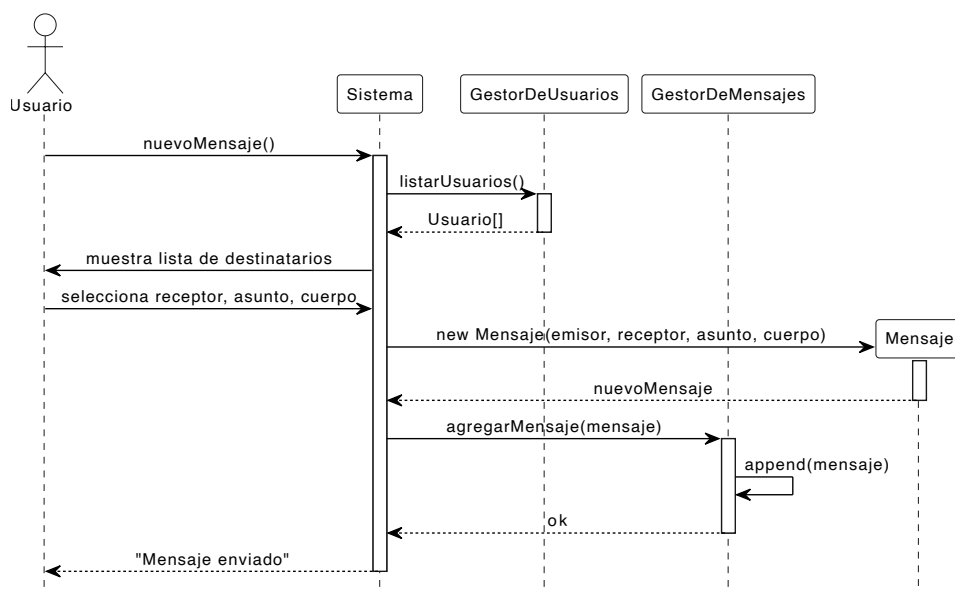
10. Diagrama de secuencia

Un **diagrama de secuencia** muestra el orden temporal de los mensajes entre objetos durante la ejecución de una operación concreta. Es útil para visualizar cómo colaboran los objetos en un caso específico, paso a paso.

10.1. Caso: enviar un mensaje

El caso de uso “enviar mensaje” involucra al usuario emisor, al sistema, al gestor de usuarios, al gestor de mensajes y al nuevo mensaje que se crea. La secuencia es la siguiente:

1. El usuario elige “Nuevo mensaje” del menú.
2. El **Sistema** le pide al **GestorDeUsuarios** la lista de destinatarios posibles.
3. El **Sistema** le muestra esa lista al usuario.
4. El usuario elige un destinatario e ingresa asunto y cuerpo.
5. El **Sistema** crea el **Mensaje** con esos datos.
6. El **Sistema** le pide al **GestorDeMensajes** que lo agregue.
7. El gestor lo guarda en su colección interna.
8. El **Sistema** confirma la operación al usuario.



10.2. Lectura del diagrama

En el diagrama de secuencia, cada objeto tiene una línea de vida vertical. Las flechas horizontales representan mensajes entre objetos, y el tiempo transcurre de arriba hacia abajo. Las barras verticales delgadas sobre cada línea de vida indican el tiempo durante el cual el objeto está ejecutando alguna operación.

Este diagrama es especialmente útil para detectar dependencias ocultas. En este caso, casi todas las flechas salen del **Sistema** hacia las demás clases. La concentración de mensajes en un solo emisor es una señal: el sistema sigue siendo el que conoce el flujo completo de las operaciones, y eso es esperable. El sistema es el orquestador.

11. Colaboración entre objetos

Un diagrama de clases muestra la estructura del sistema, y un diagrama de secuencia muestra el orden temporal de un caso concreto. Pero conviene además tener una **tabla de colaboraciones**, que resume todas las interacciones entre objetos del sistema en una sola vista.

11.1. Colaboraciones principales

Emisor	Mensaje	Receptor
Sistema	comprobarUsuario(username, contraseña)	Cuenta
Sistema	obtenerOpciones()	Cuenta
Sistema	agregar(cuenta)	GestorDeUsuarios
Sistema	eliminar(cuenta)	GestorDeUsuarios
Sistema	listarUsuarios()	GestorDeUsuarios
Sistema	agregarMensaje(m)	GestorDeMensajes
Sistema	mensajesPara(u)	GestorDeMensajes
Sistema	mensajesDe(u)	GestorDeMensajes
Sistema	leer()	Mensaje
GestorDeMensajes	esPara(u)	Mensaje
GestorDeMensajes	esDe(u)	Mensaje

11.2. Lectura de la tabla

El **Sistema** es el emisor de la mayoría de las colaboraciones. Esto es consecuencia directa de la decisión de diseño: el sistema es el orquestador, el único que conoce el flujo completo de las operaciones. Las dos excepciones son las filas del **GestorDeMensajes**, que consulta a cada **Mensaje** para saber si es para un usuario o si es de un usuario.

Ninguna de las clases colaboradoras sabe qué es un “flujo completo de la aplicación”.

Solo conocen su parte: “guardo mensajes”, “administro cuentas”, “me autentico”, “tengo un estado de lectura”. El conocimiento del flujo vive en el **Sistema**.

12. Del diseño al código

Llegó el momento de escribir el código en **Python** que refleja el modelo que construimos. No vamos a mostrar el código completo, pero sí las partes más importantes para que se vea cómo se traduce cada pieza del diseño.

12.1. Las clases del dominio

La clase **Mensaje** traduce directo del diagrama: sus atributos son referencias a usuarios y valores simples, y sus métodos implementan las operaciones que definimos.

```
1  from datetime import date
2  class Mensaje:
3      def __init__(self, emisor: Usuario, receptor: Usuario,
4                  asunto: str, cuerpo: str) -> None:
5          self.__emisor = emisor
6          self.__receptor = receptor
7          self.__asunto = asunto
8          self.__cuerpo = cuerpo
9          self.__leido: bool = False
10         self.__fecha_envio: date = date.today()
11
12     def leer(self) -> str:
13         self.__leido = True
14         return (f"De: {self.__emisor}\n"
15               f"Para: {self.__receptor}\n"
16               f"Fecha: {self.__fecha_envio}\n"
17               f"Asunto: {self.__asunto}\n\n"
18               f"{self.__cuerpo}")
19
20     def es_para(self, u: Usuario) -> bool:
21         return self.__receptor == u
22
23     def es_de(self, u: Usuario) -> bool:
24         return self.__emisor == u
```

El doble guion bajo en los atributos marca que son privados, en línea con el principio de encapsulamiento que estudiamos.

La jerarquía de cuentas se implementa con una clase abstracta:

```
1 class Cuenta(ABC):
2     def __init__(self, username: str, contrasenia: str) -> None:
3         self.__username = username
4         self.__contrasenia = contrasenia
5
6     def comprobar_usuario(self, username: str,
7                           contrasenia: str) -> bool:
8         return (username == self.__username
9               and contrasenia == self.__contrasenia)
10
11    def __str__(self) -> str:
12        return self.__username
13
14    @abstractmethod
15    def obtener_opciones(self) -> list[str]:
16        ...
```

Las subclases **Usuario** y **Admin** implementan **obtener_opciones** devolviendo las etiquetas del menú que les corresponden:

```
1 class Admin(Cuenta):
2     def obtener_opciones(self) -> list[str]:
3         return [const.CREAR_CUENTA, const.BORRAR_CUENTA,
4               const.SALIR]
5
6 class Usuario(Cuenta):
7     def obtener_opciones(self) -> list[str]:
8         return [const.NUEVO_MENSAJE, const.LEER_MENSAJE,
9               const.VER_ENVIADOS, const.SALIR]
```

Cada subclase sabe qué opciones le corresponden a su rol. Esa es la parte polimórfica de la jerarquía.

12.2. Los gestores

Los gestores son las fachadas que encapsulan las colecciones. El de mensajes:

```
1 class GestorDeMensajes:
2     def __init__(self) -> None:
3         self.__mensajes: list[Mensaje] = []
4
5     def agregar_mensaje(self, m: Mensaje) -> None:
6         self.__mensajes.append(m)
7
8     def mensajes_para(self, u: Usuario) -> list[Mensaje]:
9         return [m for m in self.__mensajes if m.es_para(u)]
10
```

```
11     def mensajes_de(self, u: Usuario) -> list[Mensaje]:  
12         return [m for m in self.__mensajes if m.es_de(u)]
```

Y el de usuarios, simétrico:

```
1  class GestorDeUsuarios:  
2      def __init__(self) -> None:  
3          self.__cuentas: list[Cuenta] = [Admin('admin', 'admin')]  
4  
5      def agregar(self, cuenta: Cuenta) -> None:  
6          self.__cuentas.append(cuenta)  
7  
8      def eliminar(self, cuenta: Cuenta) -> None:  
9          self.__cuentas.remove(cuenta)  
10  
11     def buscar_por_username(self, username: str) -> Cuenta|None:  
12         for c in self.__cuentas:  
13             if str(c) == username:  
14                 return c  
15         return None  
16  
17     def listar(self) -> list[Cuenta]:  
18         return list(self.__cuentas)  
19  
20     def listar_usuarios(self) -> list[Usuario]:  
21         return [c for c in self.__cuentas if c.es_usuario()]
```

12.3. El Sistema

El **Sistema** es el punto de entrada. Su constructor crea los dos gestores:

```
1  class Sistema:  
2      def __init__(self) -> None:  
3          self.__gestor_usuarios = GestorDeUsuarios()  
4          self.__gestor_mensajes = GestorDeMensajes()  
5  
6      def ejecutar(self) -> None:  
7          ...
```

El bucle principal de **ejecutar** es el que orquesta todo: pide credenciales, muestra el menú correspondiente a la cuenta activa, y ejecuta la operación elegida. Veamos las partes más importantes.

12.3.1. El despacho de operaciones

Aquí aparece un detalle de implementación interesante. Para vincular cada menú con el método que la ejecuta, se decidió que el Sistema mantenga un **diccionario de acciones**:

```
1 self.__acciones: dict[str, Any] = {
2     const.CREAR_CUENTA: self.__crear_cuenta,
3     const.BORRAR_CUENTA: self.__borrar_cuenta,
4     const.NUEVO_MENSAJE: self.__nuevo_mensaje,
5     const.LEER_MENSAJE: self.__leer_mensaje,
6     const.VER_ENVIADOS: self.__ver_enviados,
7     const.SALIR: self.__cerrar_sesion }
```

Cuando el usuario elige una opción, el sistema busca la etiqueta en el diccionario y ejecuta el método asociado:

```
1 def __menu(self, ca: Cuenta) -> bool:
2     opciones = ca.obtener_opciones()
3     op = io.menu(opciones, 'Menú principal') - 1
4     return self.__acciones[opciones[op]](ca)
```

Esta solución funciona: el vínculo entre la etiqueta que ve el usuario y el método que se ejecuta está concentrado en un solo lugar. Agregar una operación nueva implica agregar una entrada al diccionario y modificar las listas de opciones de las cuentas correspondientes.

12.3.2. Un ejemplo de operación

Veamos cómo está implementada una de las operaciones, `nuevoMensaje`:

```
1 def __nuevo_mensaje(self, ua: Cuenta) -> bool:
2     usuarios = self.__gestor_usuarios.listar_usuarios()
3     if not usuarios:
4         print("No hay usuarios registrados.")
5         return False
6     nombres = [str(u) for u in usuarios]
7     nombres.append("Cancelar")
8     op = io.menu(nombres, "Seleccione usuario destinatario") - 1
9     if 0 <= op < len(usuarios):
10        receptor = usuarios[op]
11        asunto = input("Ingrese asunto: ")
12        cuerpo = input("Ingrese mensaje: ")
13        m = Mensaje(ua, receptor, asunto, cuerpo)
14        self.__gestor_mensajes.agregar_mensaje(m)
15        print("Mensaje enviado.")
16    return False
```

El método hace cinco cosas: le pide al gestor de usuarios la lista de destinatarios, muestra un menú, pide los datos al usuario, crea el mensaje, y le pide al gestor de mensajes que lo guarde. El sistema coordina, los gestores ejecutan.

13. Una pregunta abierta

El diseño que construimos funciona y está bien organizado. Los gestores encapsulan las colecciones, la jerarquía de cuentas aprovecha el polimorfismo, y cada clase tiene una responsabilidad clara.

Sin embargo, hay una observación que vale la pena hacer antes de cerrar. Miremos con atención el **Sistema** y su diccionario de acciones:

- Cada vez que se agrega una operación nueva, hay que agregar una entrada al diccionario, agregar el método correspondiente al **Sistema**, y modificar las listas de opciones de las cuentas.
- Todos los métodos de negocio viven en la misma clase. El **Sistema** sabe cómo crear cuentas, cómo borrarlas, cómo enviar mensajes, cómo leerlos, cómo ver los enviados. Es responsable de muchas cosas distintas.

Estas observaciones no son un error del diseño. Son la consecuencia natural de haber seguido el camino más directo desde el análisis del enunciado hasta el código. El diseño cumple con lo que se propuso hacer: modelar los conceptos del dominio, sus relaciones y sus operaciones.

Pero queda abierta una pregunta: **¿qué pasaría si mañana quisiéramos agregar una nueva operación al menú?** ¿Qué archivos habría que tocar? ¿Cuánto del código actual se vería afectado? ¿Hay una forma de diseñar el sistema para que agregar operaciones nuevas no implique modificar el **Sistema**?

Esas preguntas no las vamos a responder ahora. Pero conviene dejarlas planteadas, porque son el tipo de pregunta que un buen diseño de software debería poder responder con facilidad.

Sobre el repositorio

El código completo que implementa este diseño está en el repositorio del proyecto. Allí se encuentran las clases en **Python**, los tests, los diagramas en formato PlantUML y las versiones compiladas en PNG. El repositorio es la fuente de verdad del sistema: si hay una diferencia entre el código y este anexo, la versión actualizada es siempre el repositorio.

<https://github.com/tute-avalos/mensajeria-anexo-a>

14. Referencias

- Deitel, P. & Deitel, H. (2012). *Java: How to Program*. Pearson. El caso de estudio del ATM que sirvió de inspiración para la estructura de este anexo.
- Booch, G., Rumbaugh, J. & Jacobson, I. (2005). *The Unified Modeling Language User Guide*. Addison-Wesley.
- Object Management Group. *Unified Modeling Language 2.5.1*. <https://www.omg.org/spec/UML/>
- Refactoring Guru: <https://refactoring.guru/es/>