
# TP 6: SOLID

Programación Orientada a Objetos II
Comisión 2
2º Cuatrimestre de 2024

Nicolás Salvanescki

# 1 Caso 1

1. Diagrama de clases UML del código dado:

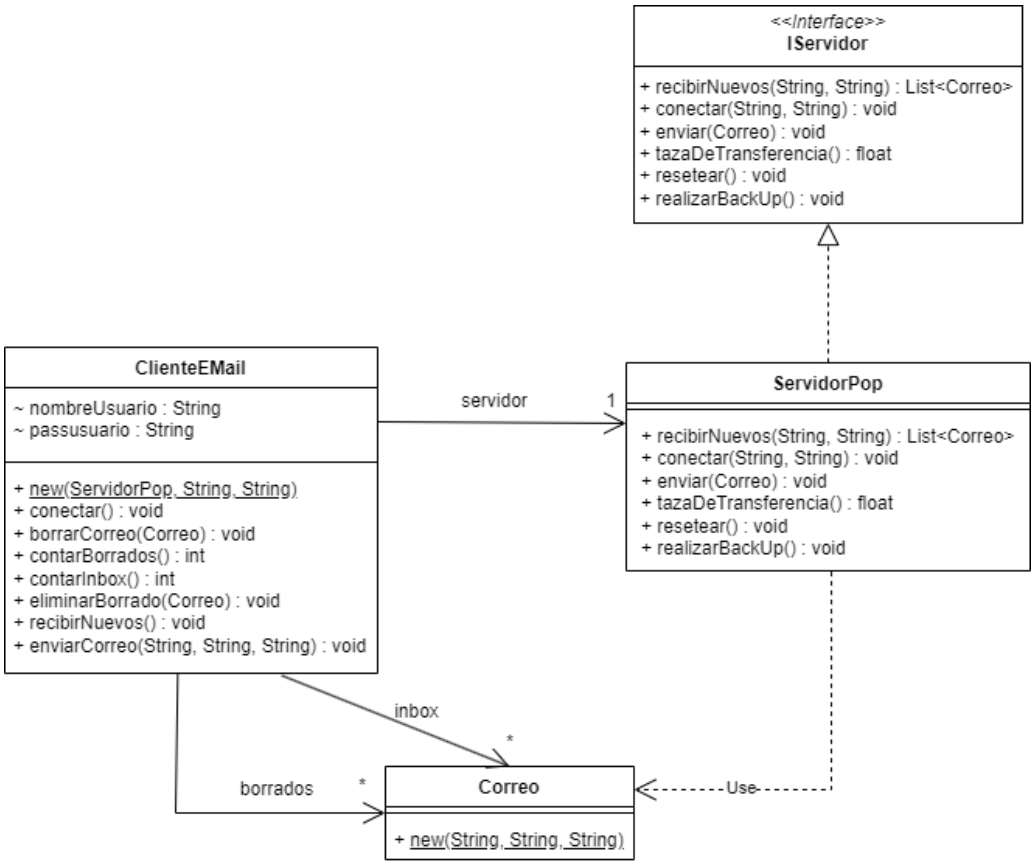


Figure 1: Diagrama de clases UML original

2. Las violaciones que se producen a los principios SOLID son varias:

   - ClienteEMail tiene demasiadas razones de cambio: la conexión con el servidor, la autenticación, el manejo del almacenamiento de correos y la gestión sobre correos individuales. Esto viola claramente el principio de Single Responsability.
   - Además, en vez de conocer a la interfaz IServidor, ClienteEMail utiliza directamente la clase ServidorPop (la cual implementa IServidor). Esto hace que la interfaz no esté cumpliendo exactamente ningún rol, restando polimorfismo al código y violando Dependency Inversion. Además, esto expone la implementación de ServidorPop ya que, si tuviera más protocolo aparte de la interfaz IServidor, ClienteEMail conocería todos esos mensajes aunque no tuviera que hacerlo, ya que solo debería hacer uso de IServidor.
   - El Open-Closed se viola en casi todas las variables de instancia de ClienteEMail. Particularmente: servidor, nombreUsuario, passusuario e inbox, están definidas con visibilidad default (o package), esto significa que

cualquier clase definida en el mismo paquete que ClienteEMail va a poder acceder sin ningún tipo de restricción a dichas variables.

- ServidorPop no implementa varios de los métodos de los que IServidor define su contrato. Esto supone una violación al principio de Interface Segregation, ya que se debería separar IServidor en varias interfaces más pequeñas y granulares, para que ServidorPop solo deba implementar las que debe y no todas.

3. Las soluciones que deben llevarse a cabo son:

   - La más sencilla es cambiar la visibilidad de todas las variables de instancia a privadas y, en caso de tener que hacerlo, definir getters para acceder a sus valores.
   - Segregar la interfaz IServidor en varias: una que se encargue de la conexión, otra de las tareas de un servidor POP (enviar y recibir emails) y, por último, el protocolo que no implementa el servidor POP (tazaDeTransferencia, resetear y realizarBackUp).
   - Hacer que los parámetros de los mensajes sean a interfaces abstractas que apliquen el protocolo necesario para dicho mensaje, y no implementaciones de las mismas.
   - Delegar comportamiento en ClienteEMail para que no tenga tantas razones de cambio distintas, en este caso utilicé inyección de dependencias.

4. El diagrama de clases de la solución sería el siguiente

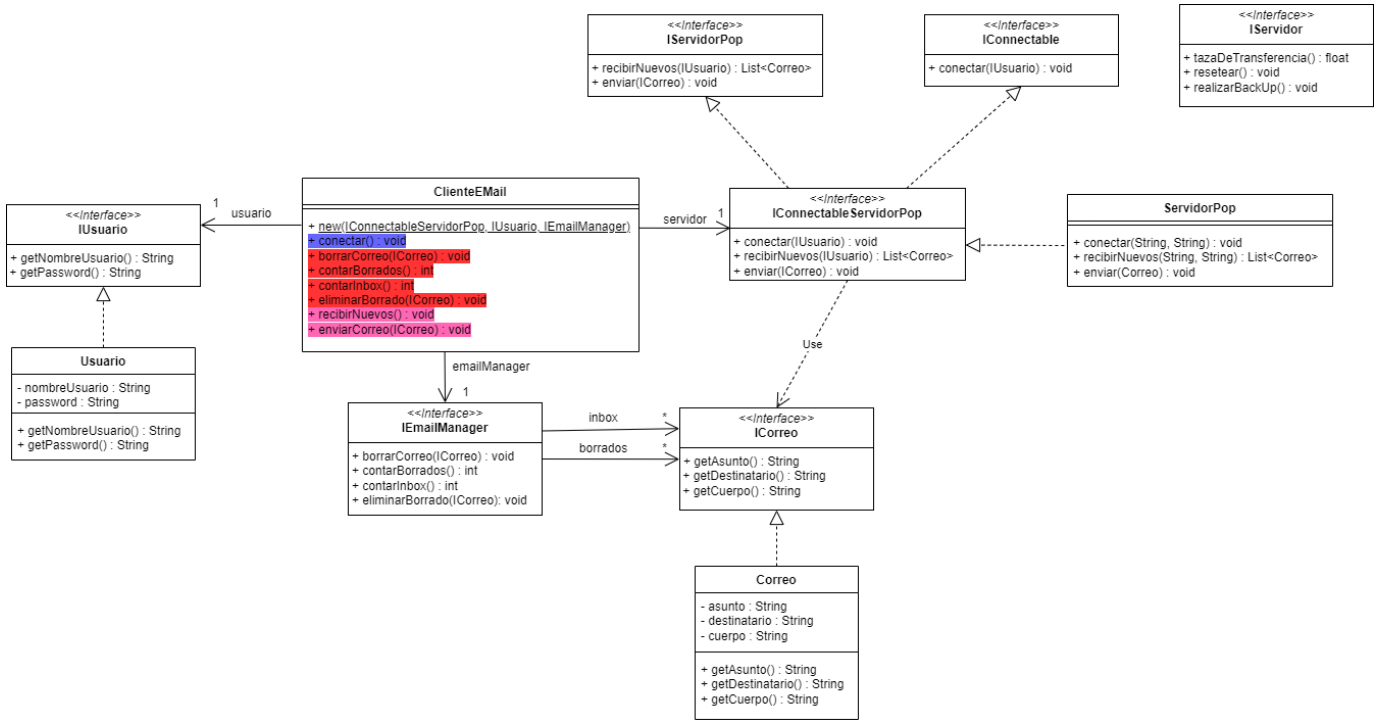


Figure 2: Diagrama de clases UML arreglado

5. La implementación del código en Java está hecha y se encuentra en ar.edu.unq.po2.tpSOLID.caso1

# 2 Banco y préstamos

En este caso, una primera aproximación podría ser la siguiente

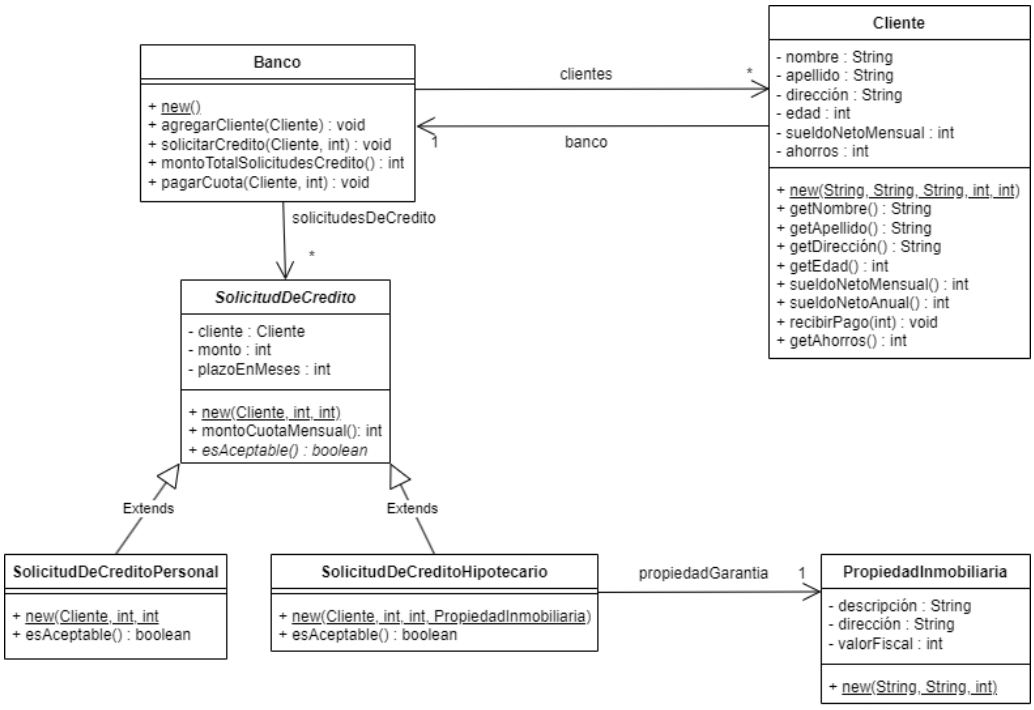


Figure 3: Diagrama de clases UML - Banco

Sin embargo, hay varios problemas en esta solución. En primer lugar, Cliente guarda los datos del cliente y además tiene lógica de Pagable (recibirPago) y financiera (ahorros, sueldo). Una buena solución sería separar estas lógicas.

Banco, a su vez, tiene la lógica de entidad de crédito (solicitarCredito, montoTotalSolicitudes, pagarCuota) y la de guardar clientes. Pueden separarse en EntidadCrediticia y Banco probablemente.

Respecto a las solicitudes de crédito, probablemente debería separarse la lógica de esAceptable en una abstracción aparte para prepararlas ante una futura expansión del código. Pero, no se me ocurre ahora como hacerlo, así que por ahora lo dejo así.

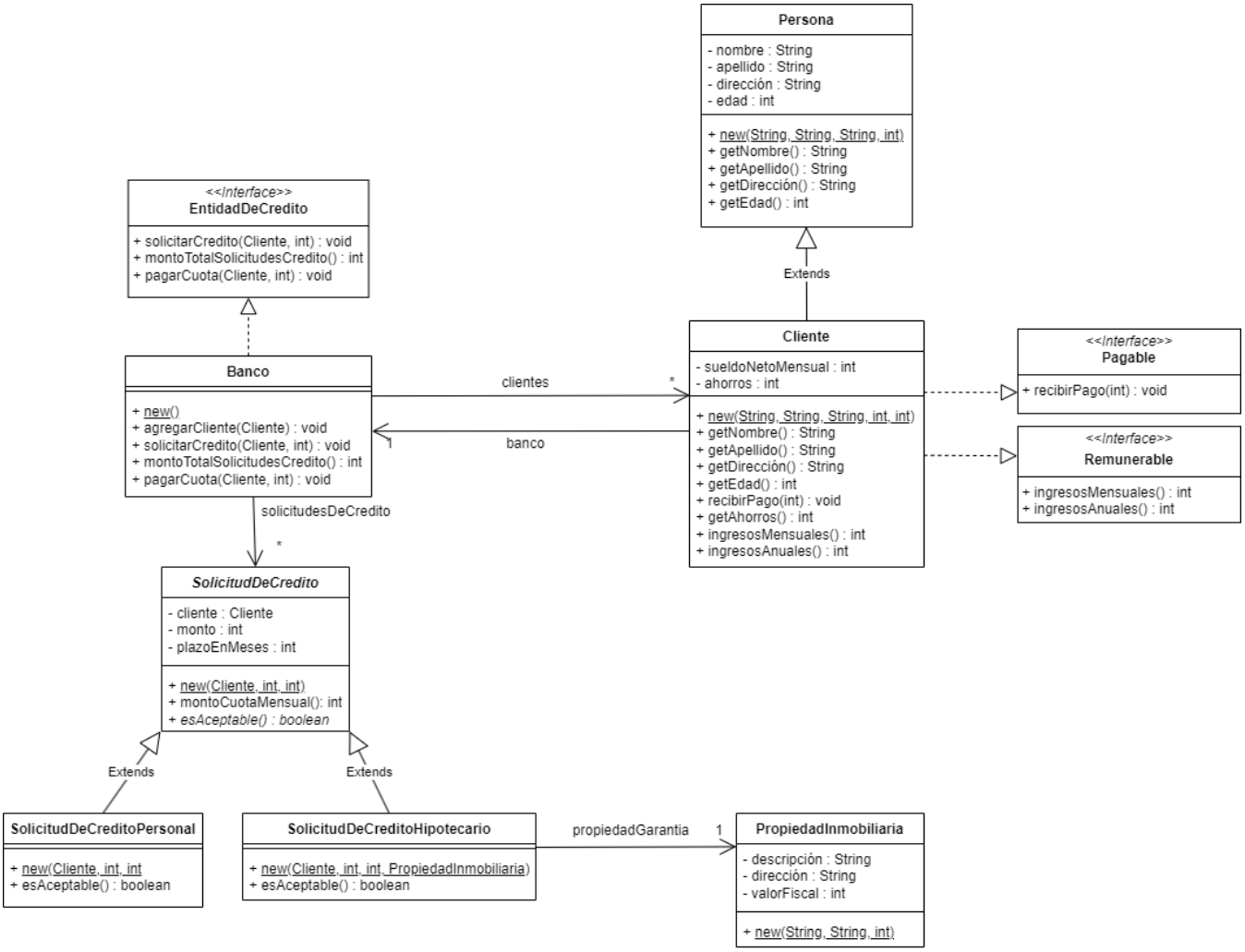


Figure 4: Diagrama de clases UML - Banco (Arreglado)

La implementación en Java usando TDD está en los paquetes de src y test: ar.edu.unq.po2.tpSOLID.banco