Programación Modular y Orientación a Objetos - Laboratorio 4

Requisitos previos

El alumnado se desenvuelve correctamente en el entorno Eclipse y maneja adecuadamente la perspectiva Java. Además, es capaz de crear y representar mediante diagramas UML clases con atributos y métodos sencillos, así como relaciones simples entre clases. Por último, debe saber realizar casos y suites de prueba para estas clases con el framework JUnit.

Objetivos

Este laboratorio sirve para profundizar en el uso de Tipos Abstractos de Datos (TADs) y Máquinas Abstractas de Estado (MAEs). Como novedad, se presentará la escritura de mensajes en la consola del sistema (pantalla), y se trabajará con la clase *ArrayList*, utilizando sus métodos más importantes e iteradores sobre dicha clase.

Al finalizar este laboratorio, el alumnado deberá ser capaz de:

- Realizar ejercicios que incluyan clases que implementen tanto TADs como MAEs.
- Estar familiarizado con el uso de listas (*ArrayList*) e iteradores.
- Mostrar mensajes por la consola del sistema.
- Efectuar con soltura pruebas con JUnit.

Motivación

Este laboratorio presentará por primera vez un ejercicio de cierta complejidad que incluye 4 clases interrelacionadas. Se facilita el diagrama de clases con las relaciones entre ellas, ya que entre los objetivos del laboratorio no se incluye de momento el profundizar en cuestiones de diseño. Este objetivo se perseguirá más adelante, especialmente en la asignatura Ingeniería del Software.

Tarea única: PromoBank

Se quiere implementar una aplicación que permita gestionar las transacciones bancarias que los clientes de PromoBank realizan a través de la red de cajeros automáticos.

La figura siguiente muestra el diagrama de clases que se utilizará en esta tarea. Como puede verse, se han identificado dos clases que se van a implementar como MAEs: la lista de clientes y la lista de operaciones.

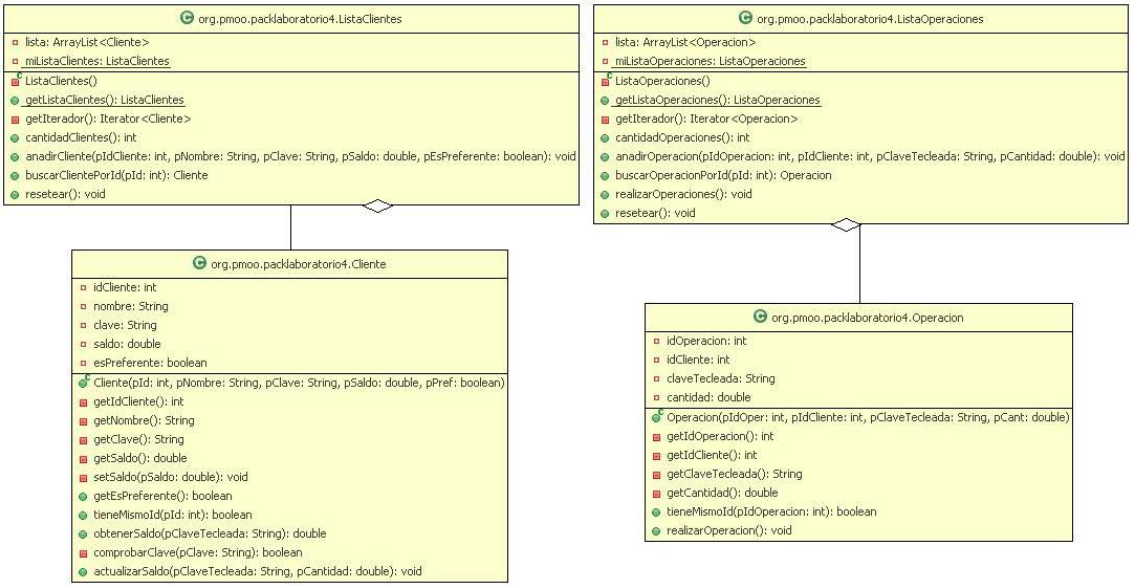

Figura 1 - Diagrama de clases para la tarea PromoBank.

La lista de clientes contiene la información relativa a todos los clientes del banco. Por cada uno de ellos se almacena su identificador, su nombre, su clave, su saldo y un atributo booleano que indica si se trata de un cliente preferente.

La lista de operaciones guarda en tiempo real la información de las transacciones que hay pendientes de realizar. Una operación se caracteriza por su identificador, el identificador del cliente al que afecta, la clave que el cliente ha introducido al identificarse en el cajero automático y el importe asociado a la operación. Para no complicar el ejercicio, se va a suponer que todas las operaciones son de cobro, esto es, transacciones en las que el cliente quiere sacar dinero de su cuenta. Por lo tanto, una operación consistirá en restar el importe al saldo del cliente. Adicionalmente, estas transacciones de cobro aplican una comisión del 0.1% sobre el importe a aquellos clientes que no son preferentes.

Se quiere implementar un método (*realizarOperaciones()*)que efectúe todas las transacciones que hay en la lista de operaciones. Antes de efectuar una operación, es necesario verificar que la clave que el cliente ha introducido al identificarse se corresponde con la clave del cliente al que afecta la transacción, esto es, que ambas contraseñas (la del cliente y la que el cliente ha tecleado en el cajero) son iguales. Si esto no ocurriera, se mostraría un mensaje de alerta por pantalla y se cancelaría el proceso de realización de la transacción, continuando con la siguiente transacción de la lista.

Por cuestiones de seguridad, en ningún momento un objeto de tipo Cliente va a proporcionar su propia clave. Por este motivo, el método *getClave()* de la clase Cliente se ha definido como privado. Adicionalmente, se debe utilizar el método privado *comprobarClave()*,

que devuelve un booleano indicando si la clave del usuario es igual a la clave que se le pasa como parámetro. De este modo, los métodos públicos *ObtenerSaldo()* y *actualizarSaldo()* comprobarán si la clave que reciben como parámetro es correcta, y solamente en ese caso realizarán su función.

Se pide:

- dibujar el **diagrama de secuencia** relativo al método *realizarOperaciones()*, encargado de la realización de las operaciones de la lista de operaciones.
- **implementar las clases** siguientes, cuyos atributos y métodos se describen a continuación, así como las **pruebas unitarias** que permitan verificar su corrección.

- **Clase Cliente**
  - **Atributos:** un identificador (int), el nombre (String), la clave (String), el saldo (double) y si es preferente o no (boolean).
  - **Método constructor:** recibe como parámetro los valores con los que inicializar los atributos.
  - **getters y setters:**
    - *getIdCliente()*, *getNombre()*, *getClave()*, *getSaldo()* y *getEsPreferente()* devuelven respectivamente el valor de los atributos idCliente, nombre, clave, saldo y esPreferente. Todos ellos son privados, excepto el último, que se va a utilizar desde fuera de la clase Cliente.
    - *setSaldo()* modifica el valor del saldo del cliente.
  - **Otros métodos:**
    - *tieneMismoId()* devuelve un booleano indicando si el identificador que recibe como parámetro es igual al del cliente.
    - *comprobarClave()*, como se ha argumentado anteriormente, devuelve un booleano indicando si la clave que recibe como parámetro es igual a la del cliente.
    - *obtenerSaldo()* en primer lugar comprueba si la clave que recibe como parámetro es la del cliente, y en caso afirmativo devuelve su saldo. Por el contrario, si la clave no es correcta, entonces se devolverá el valor 0.0.
    - *actualizarSaldo()* en primer lugar comprueba si la clave que recibe como parámetro es igual a la del cliente, y en caso afirmativo resta al saldo del cliente la cantidad que recibe como parámetro, excepto si dicha cantidad es negativa o superior al saldo, en cuyo caso no se modificaría el saldo. Si la clave recibida como parámetro no fuera correcta, no se modificaría el saldo del cliente. En cualquier caso se muestra un mensaje en la consola del sistema: si la operación se ha realizado correctamente, se indica el nuevo saldo del cliente, y si no se ha podido actualizar el saldo, se avisa de tal circunstancia indicando el motivo.

- **Clase ListaClientes (MAE)**
  - **Atributos:** una lista de clientes, implementada en este caso como ArrayList, y la instancia (única) de ListaClientes.
  - **Método constructor:** inicializa el atributo de tipo lista de la clase.
  - **getters y setters:**
    - *getListaClientes()* devuelve la única instancia de ListaClientes para que pueda ser visible desde cualquier otra clase.
  - **Otros métodos:**
    - *getIterador()* devuelve un elemento de tipo Iterator para que los métodos de la clase ListaClientes puedan recorrer la lista con él.
    - *buscarClientePorId()* devuelve el cliente que tiene como identificador el valor que recibe como parámetro. Si no existe tal usuario, devolverá el valor *null*.
    - *anadirCliente()* añade a la lista el cliente que recibe como parámetro, excepto si ya existe un cliente con el mismo identificador, en cuyo caso no se añadiría y se mostraría un mensaje por la consola del sistema avisando de tal circunstancia.
    - *resetear()* restaura el valor inicial de la lista de clientes, o lo que es lo mismo, la vacía. La utilidad de este método vendrá a la hora de realizar las pruebas unitarias, ya que en principio no debería existir en una aplicación real, y menos si su visibilidad es pública.
    - *cantidadClientes()* devuelve el número de clientes que hay almacenados en la lista.

- **Clase Operacion**
  - **Atributos:** un identificador (int), el identificador del cliente al que afecta la operación (int), la clave que ha tecleado el cliente en el cajero (String) y la cantidad asociada a la operación (double).
  - **Método constructor:** recibe como parámetro los valores con los que inicializar los atributos.
  - **getters y setters:**
    - *getIdOperacion()*, *getIdCliente(), getClaveTecleada(),* y *getCantidad()* devuelven respectivamente el valor de los atributos idOperacion, idCliente, claveTecleada y cantidad. Dado que no se van a utilizar fuera de la clase Operacion, se definen como métodos privados.
  - **Otros métodos:**
    - *tieneMismoId()* devuelve un booleano indicando si el identificador que recibe como parámetro es igual al de la operación.
    - *realizarOperacion()* se encarga de comprobar que existe un cliente cuyo identificador es el que afecta a la operación. Si lo encuentra, entonces actualiza su saldo en función de la cantidad asociada a la

operación y de si se trata o no de un cliente preferente. Por el contrario, si no existiera tal cliente, mostraría un mensaje por la consola del sistema indicando tal circunstancia.

- **Clase ListaOperaciones (MAE)**
  - **Atributos:** una lista de operaciones, implementada en este caso como ArrayList, y la instancia (única) de ListaOperaciones.
  - **Método constructor:** inicializa el atributo de tipo lista de la clase.
  - **getters y setters:**
    - *getListaOperaciones()* devuelve la única instancia de ListaOperaciones para que pueda ser visible desde cualquier otra clase.
  - **Otros métodos:**
    - *getIterador()* devuelve un elemento de tipo Iterator para que los métodos de la clase ListaOperaciones puedan recorrer la lista con él.
    - *buscarOperaciónPorId()* devuelve la operación cuyo identificador es el valor que recibe como parámetro. Si no existe tal operación, devolverá el valor *null*. Este método debe definirse como privado, pues no se usa fuera de la clase ListaOperaciones.
    - *anadirOperacion()* añade a la lista la operación que recibe como parámetro, excepto si ya existe una operación con el mismo identificador, en cuyo caso no se añadiría y se mostraría un mensaje por la consola del sistema avisando de tal circunstancia.
    - *resetear()* restaura el valor inicial de la lista de operaciones, o lo que es lo mismo, la vacía. La utilidad de este método vendrá a la hora de realizar las pruebas unitarias, ya que en principio no debería existir en una aplicación real, y menos si su visibilidad es pública.
    - *cantidadOperaciones()* devuelve el número de clientes que hay almacenados en la lista.
    - *realizarOperaciones()* realiza todas las operaciones que hay en la lista en ese momento. Este método no modifica la lista de operaciones, sino que la deja en el mismo estado en el que se encontraba al principio.