CURSO 2015 - 2016

ACTIVIDAD EVALUABLE Y CALIFICABLE

1. Portada con:

Asignatura: 71013035 – Diseño de Software

Año de la práctica:

Centro Asociado al que pertenece:

Tutor que le da asistencia:
(o grupo de tutoría Intercampus al que está adscrito)

Datos personales: Nombre y apellidos:

DNI o número de expediente:

Contacto (teléfono o correo electrónico):

Localidad de residencia:

Datos de coste: Horas de dedicación al estudio de los contenidos:

Nº de actividades no evaluables realizadas y horas de dedicación:

Horas de dedicación para realizar esta actividad:

Juan del Rosal, 16
28040, Madrid

Tel: 91 398 89 10
Fax: 91 398 89 09

swdesign@issi.uned.es

2. El enunciado y planteamiento del caso de estudio.

El dominio del problema es un **subsistema de transacciones simples de una entidad bancaria con sus clientes** (BankBox).

Restricciones y simplificaciones:

- Los clientes son, únicamente, personas físicas. Sólo se consideran los usuarios que son clientes de la entidad; es decir, son titulares de algún producto (cuenta, tarjeta, etc.) El acceso a los servicios requiere la identificación del cliente.
- Aunque las operaciones se podrían realizar a través de una interfaz Web, en cualquier dispositivo informático con esa capacidad, el estudio se reduce a los **cajeros automáticos** de la entidad.
- Los servicios que se van a considerar son:
    - Consultar saldo.
    - Información de movimientos en cuenta.
    - Obtener dinero en metálico.
    - Ingresar dinero en cuenta.
    - Ingresar dinero en tarjeta-monedero.
    - Transferencia entre dos cuentas (la de destino puede ser de otra entidad). La transferencia se realizará en la divisa local del cajero.

En cuanto al alcance del sistema, se pretende que permita operaciones:

- Con distintos tipos de productos (cuentas corrientes, de crédito, de ahorro, etc.); a los que se aplican distintos tipos de comisiones.
- Con distintos tipos de tarjetas (de crédito, de compras, de pago aplazado, monedero, etc.); a las que también se aplican diferentes cargos, según el tipo de tarjeta y la operación realizada.

Supóngase que la arquitectura de este subsistema tiene una estructura en 3 capas:

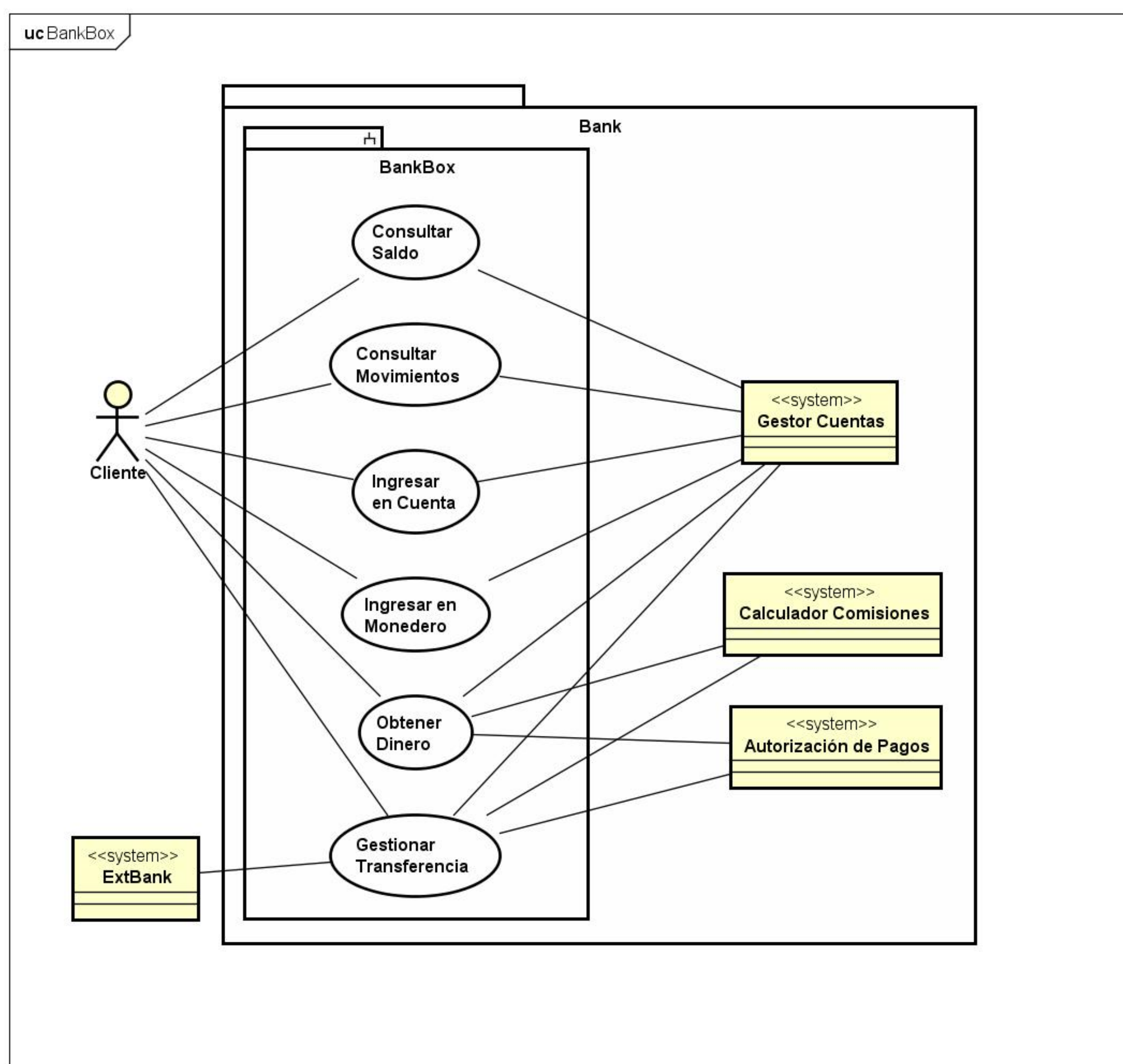
- Capa de presentación.
- Capa de la lógica del negocio.
- Capa del acceso a datos o de los servicios técnicos.

En adelante, en el escenario descrito, el estudio se hará sobre la capa de la lógica y los servicios del negocio. Es decir, aunque exista una capa de presentación, no debe tener en cuenta la interfaz de usuario, la presentación o la captura de eventos. Céntrese en la lógica de la interacción cliente-banco y su funcionamiento.

3. **El enunciado de cada cuestión y las respuestas.** Para cada cuestión, incluirá los desarrollos, listados, diagramas y las argumentaciones que estime necesarios.

Sección 1. Evaluación de ***Casos de Uso***

1. (0'5 puntos) En relación a las operaciones en el cajero automático, identifique al menos 4 casos de uso primarios y sus actores correspondientes. Represente los resultados en un diagrama de casos de uso de UML.



Aunque no queda claro en el enunciado, su intención es realizar el estudio enfocado a los servicios que da el cajero. En todos ellos, el antecedente necesario es la identificación del cliente. Es decir, el acceso a cualquier servicio del cajero pasa por su identificación y autorización previas. Por ello, antes de utilizar cualquier servicio, habría que suponer que el Sistema BankBox crea un controlador de *sesión* (*fachada*) que realiza la identificación y autorización y, en caso positivo, consulta a la entidad la información del cliente. Sin embargo, en este estudio, se va a obviar ese antecedente; aunque no las conclusiones que se derivan de él.

Nótese que, aunque el estilo esencial aconseja independizar los aspectos del comportamiento (análisis) de una tecnología concreta o del '*cómo*', en este punto aparece la necesidad de iniciar la resolución de algunas cuestiones:

- No es razonable que el cajero, ni siquiera la sucursal en la que reside, 'contenga' la información de <u>todos</u> los clientes de la entidad. Tras la autorización del acceso, esa información (de un cliente concreto) debería obtenerse de la entidad bancaria.
- Independientemente de la tecnología utilizada para el acceso (tarjeta, lectura biométrica, etc.), lo más común en este dominio del negocio es que las operaciones asociadas a los servicios disponibles se realicen a través de un medio físico aportado por el cliente. En el enunciado aparecen las tarjetas pero podrían añadirse las '*libretas*' o los dispositivos móviles. Del '*uso*' en este dominio del negocio, también se deriva que, el medio (tarjeta), está asociado unívocamente a una cuenta y su uso restringido a un conjunto de servicios disponibles (y sus correspondientes comisiones). Por tanto, independientemente de qué mecanismo se utilice para el acceso, sí se puede concluir que, en el momento en el que se utilice el medio físico (tarjeta, libreta, etc.), se puede conocer:
    - La cuenta o producto asociado a la tarjeta.
    - Los servicios y operaciones disponibles.

  Y, cuando se seleccione la operación, también se conocerá la comisión aplicable.

2. (1 punto) Escriba el caso de uso <<*Transferencia*>> en un estilo y formato completo, esencial y breve. Incluya tanto el escenario principal de éxito (flujo básico correspondiente a que el cliente –**ya identificado y autorizado**— seleccione la operación, indique los datos necesarios, reciba la autorización –si es pertinente– y el comprobante) como 2 extensiones o flujos alternativos que pudieran ser frecuentes. No escriba un encabezamiento demasiado elaborado del caso de uso (es decir, omita *propósito*, *resumen*...); en su lugar, afronte directamente el transcurso típico de los acontecimientos.

Caso de uso: **Transferencia**

Formato completo (variante 'a dos columnas'), estilo esencial.

Evolución típica de los acontecimientos

Acciones del actor	Respuesta del sistema
1. El caso de uso comienza cuando el cliente selecciona la operación '*Transferencia*'.	2. El sistema solicita los datos para realizar la transferencia: cuenta de origen, cuenta de destino, cantidad de dinero, concepto y datos del destinatario.
3. El cliente facilita los datos pedidos (por ejemplo, rellenando un formulario).	4. El sistema verifica la información y la disponibilidad de recursos.
	5. El sistema presenta un resumen con la información de la operación, los cargos aplicables y solicita confirmación de los datos y de la operación.
6. El cliente ratifica la solicitud.	7. El sistema comprueba la aceptación del cliente.
	8. El sistema realiza la operación, la registra, actualiza la información de los productos implicados (cuenta).
	9. El sistema ofrece un comprobante con el resultado de la operación realizada.
10. El cliente recoge el comprobante y termina la operación.	

Alternativas

- 4 Alguno de los datos introducidos es incorrecto o inviable o no se puede aceptar la operativa en esos términos. Se da opción a volver a 3 o terminar la operación en 10. ¿Se registra la solicitud o sólo el acceso?
- 6 El cliente cancela la solicitud. El sistema termina la operación.
- 7 No se puede verificar la aceptación del cliente (firma, por ejemplo); bien por error en la acreditación del cliente (en cuyo caso se volvería a solicitar dando la opción de cancelar) o por error en el sistema (por ejemplo en las comunicaciones), terminándose la operación.

Este apartado está orientado a pensar:

- **QUÉ** tiene que hacer la aplicación (el '*negocio*').
- Qué **pasos** fundamentales debe dar (el software) **para hacerlo**.

En este proceso ya se empieza a identificar la información que se puede necesitar para que la aplicación realice las operaciones de cada paso, si los datos concretos se pueden obtener o deducir de la información que maneja el sistema o, por el contrario, requiere la intervención de algún actor.

Otra vez, cuando se piensa en cómo se va a realizar esto, se concluye que el software del cajero sólo debe manejar la información que aporte el cliente y la estrictamente necesaria para realizar las operaciones requeridas. Algunas de ellas deben ser extremadamente versátiles o sí necesitan un volumen de información importante; por lo que no parece razonable que se procese localmente en el cajero. Por ejemplo, la conexión a una entidad externa requiere un adaptador que sirva de interfaz para adecuar los formatos de representación de la información y el protocolo de comunicación con cada entidad. No parece viable que dicha colección de adaptadores (polimorfismo) residan en el cajero. Parece más lógico que la conexión se establezca desde el servidor central (o alguno de sus nodos supervisores).

Sección 2. Evaluación del ***Modelado Conceptual***

3. (2 puntos) En relación al caso de uso anterior <<*Transferencia*>>, construya un Modelo de Dominio y represéntelo en notación UML. Represente los objetos conceptuales, las asociaciones y los atributos.

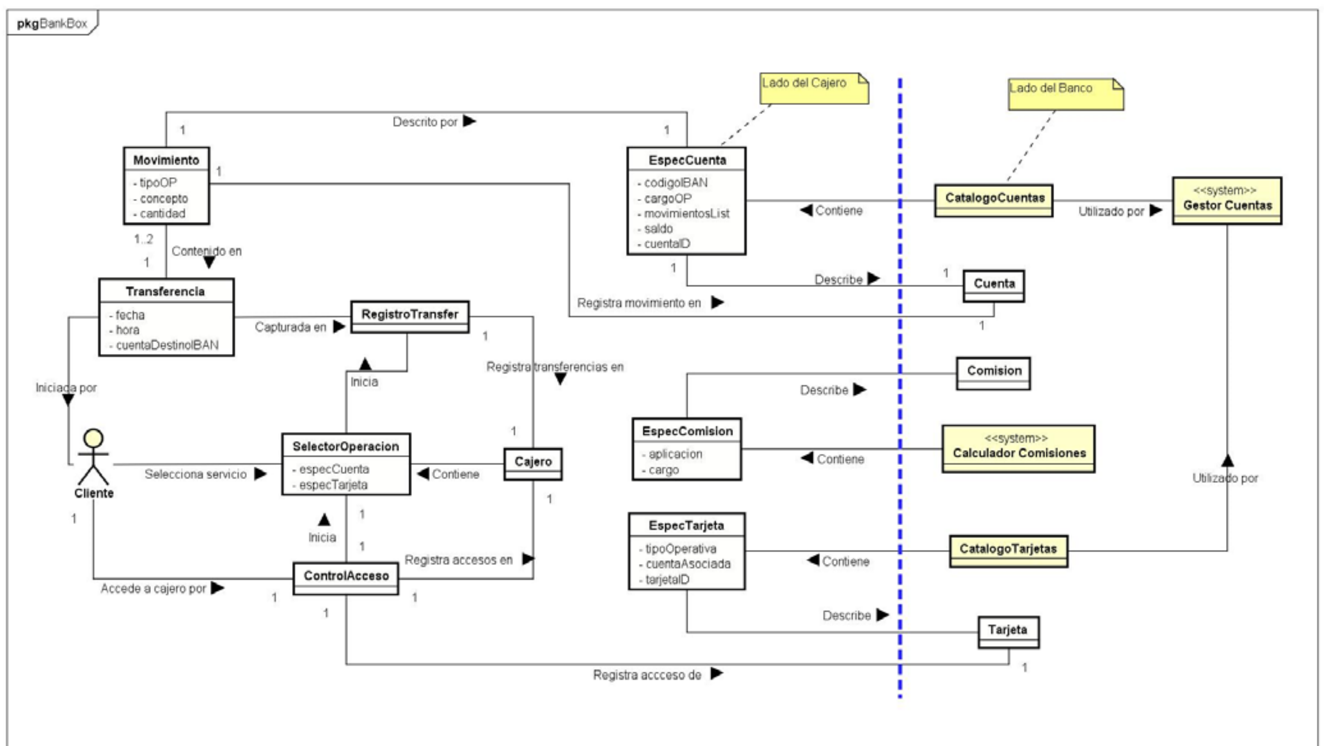
Esta sección es importante porque el Modelo es el antecedente del Diagrama de Clases, casi el objetivo final.

Se trata de asignar, las operaciones identificadas en la escritura del caso de uso, a determinados *elementos* u *objetos*. Como quiera que los principios GRASP (especialmente los 5 primeros) están orientados a mantener una '*privacidad*' en las operaciones, de forma que sólo manejan la información '*local*' (la del objeto en el que están), hay que colocar, la información que se ha identificado como necesaria para una operación, en el mismo objeto en el que está dicha operación. Es la única manera de obtener independencia funcional y, con ello, la flexibilidad que se busca para el funcionamiento del software.

Siguiendo con la idea central de lo anterior, se podría poner el ejemplo PdV del texto de la asignatura. En la escritura del caso de uso de Procesar Venta, se llega a conclusión de que una operación fundamental es registrar los datos de una venta. Por tanto, a la hora de construir el Modelo (y, luego, el diseño de clases) hay que considerar un elemento 'Registro', que recoja el registro de las ventas, y otro, 'Venta', con los datos de la venta. ¿Y cómo Registro captura o registra los datos de una venta? Pues actúa de controlador, recogiendo la información del actor y buscando el resto, y se la provee a Venta para que realice su labor.

Este caso podría ser similar: es necesario registrar los datos de una **transferencia, operación que consiste en cambiar parte de la información contenida en dos cuentas**. Esta es la primera conclusión importante del caso de uso.

Con poco que se profundice en un ejercicio, tiende hacia un planteamiento complicado. Siempre es necesaria una labor de síntesis y abstracción que puede ser difícil al principio. Así, en este caso, aunque el caso de uso está restringido a una situación 'irrealmente' reducida, la aplicación del procedimiento enunciado anteriormente de "*seguir el hilo de dónde proviene la información que se necesita*" lleva a un modelo similar a éste, de ámbito notablemente mayor:



Se ha pretendido hacer una analogía lo más próxima posible al texto y al ejemplo PdV. Aunque, como se verá en el desarrollo del diseño, no es totalmente adecuada. En aquel ejemplo (PdV), el objeto EspecificacionDelProducto es conceptualmente adecuado cuando se

quiere extraer una información ('precio') en una búsqueda en una base de datos (o catálogo) a partir de una clave (ArticuloID). Pero, aquí, se maneja el concepto 'Cuenta' y se utiliza una pequeña parte de la información contenida en ella. Ya se ha concluido que no es razonable mantener una base de datos de clientes en cada cajero (ni siquiera una copia). Por el mismo motivo, tampoco lo es mantener otra de 'tarjetas' ni de 'cuentas'. Asumiendo que, mediante un protocolo de consultas, las búsquedas se realizan remotamente, en el lado del servidor, el software del cajero sólo necesita manejar una pequeña porción de información de cada cuenta, que es la que se ha descrito con el objeto 'EspecCuenta'. No es la descripción de Cuenta si no, más bien, una 'vista' de la cuenta con la información útil.

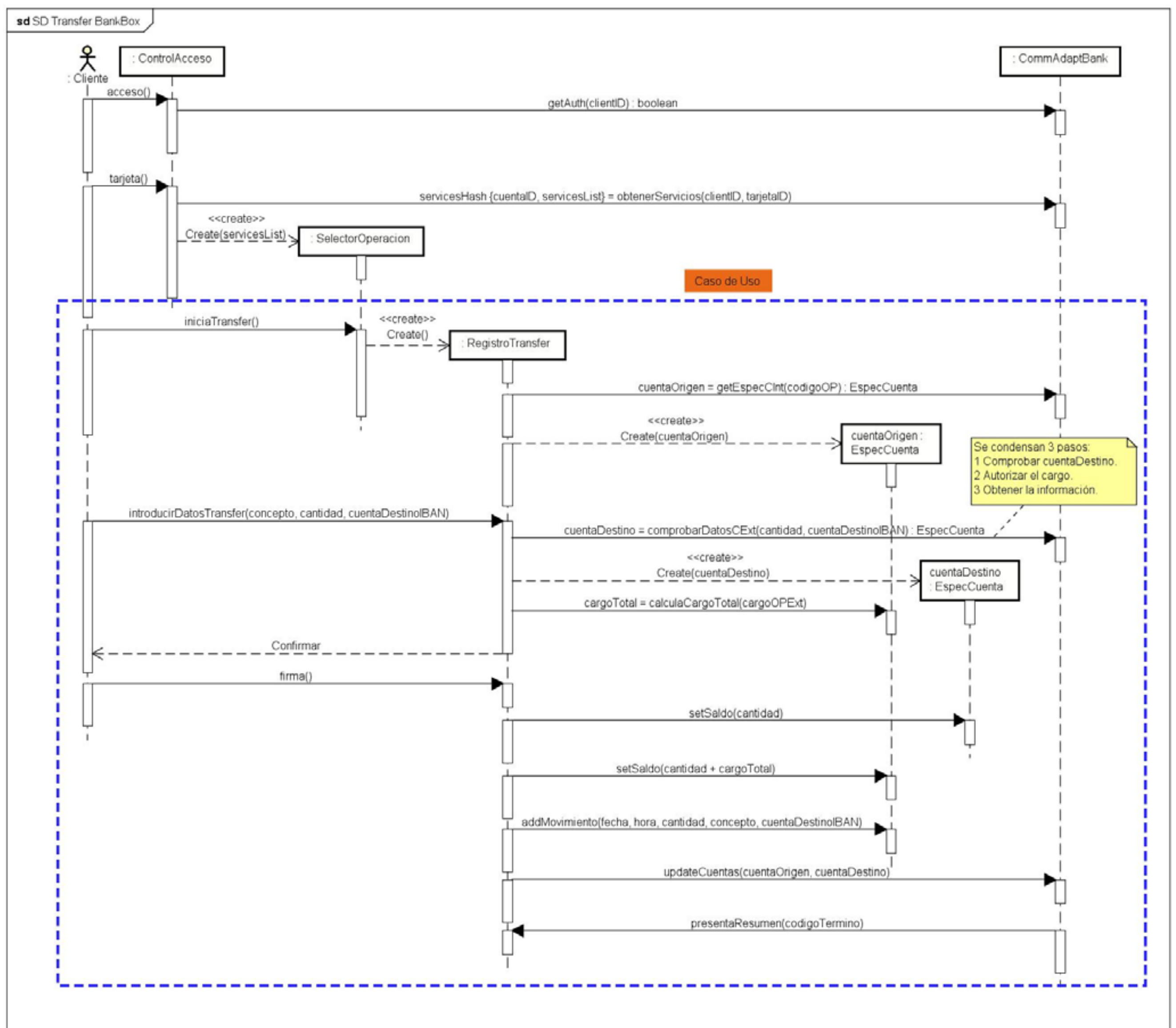
Se prevé un controlador que haga de interfaz entre el cajero (cliente) y el lado del servidor, que 'desacople' los sistemas y facilite la seguridad (línea azul punteada). En el lado del servidor (a la derecha) el procesamiento sí se realiza como en el ejemplo de PdV (objetos 'normalmente' con fondo amarillo). Aunque 'EspecCuenta' aparezca en el lado del cajero, también habrá otra copia en el lado del servidor; con la descripción que sea necesaria para el tratamiento de las cuentas y el uso del catálogo. E, igualmente, para Clientes, Tarjetas, etc.

Para ilustrar el proceso de razonamiento, se va continuar con el desarrollo de este planteamiento macro-sistema; que podría recoger la mayor parte de la operativa del cajero automático. Más adelante se volverá sobre estos pasos, retomando el Modelo ajustado al caso de uso, etc. En realidad, el desarrollo iterativo es así: de refinamiento continuo. Se ha preferido continuar con el modelo ampliado para evidenciar las decisiones que se tomen en las modificaciones del propio sistema.

Sección 3. Evaluación de los ***Eventos del Caso de Uso***

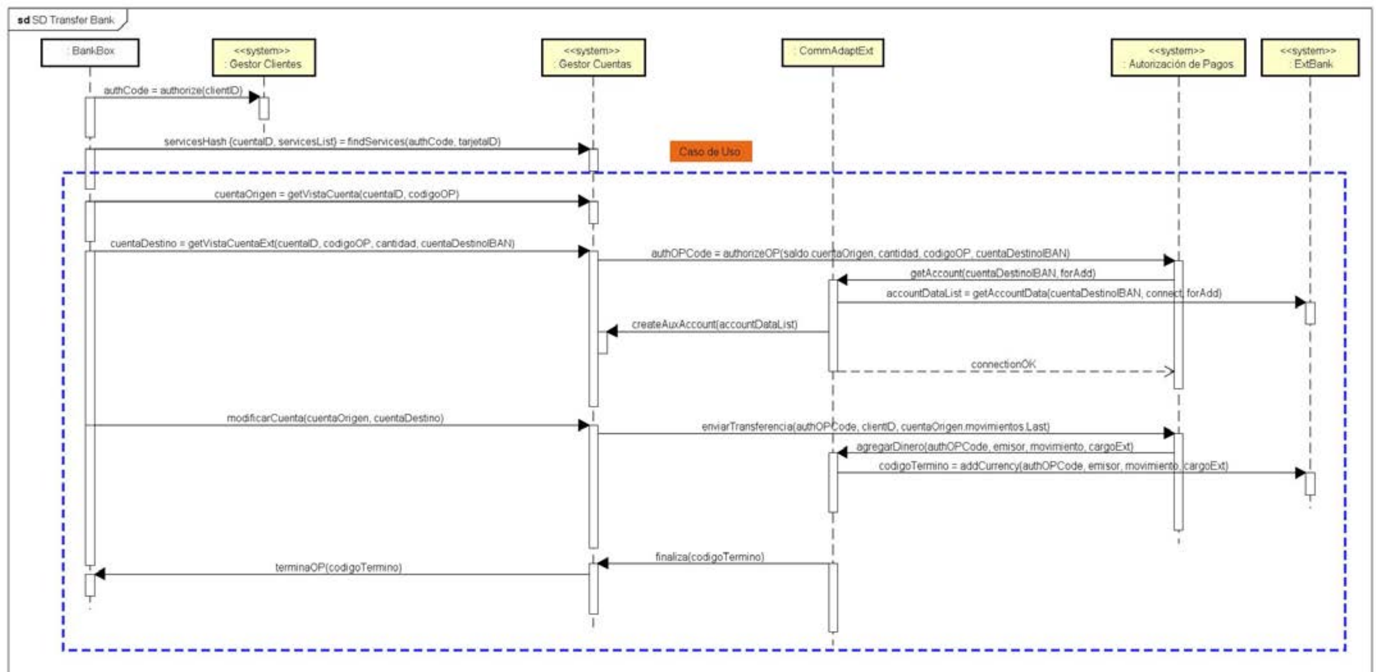
4. (1'5 puntos) Circunscrito al caso de uso anterior <<*Transferencia*>>, construya un Diagrama de Secuencia (diagrama de interacción DS) en UML. Represente los actores y los eventos de los componentes del subsistema.

A continuación aparece el DS del cajero. Aunque el enunciado restringe el caso de uso a su inicio desde la selección de la operación, se incluyen algunas operaciones del acceso como antecedente y para enmarcar dicho caso de uso. Después, se 'esboza' el DS en el 'lado del servidor' (cómo provee la información al cajero).



Nótese:

- Un actor '*humano*' no puede procesar datos. Sólo puede invocar una acción (en el software) o aportar una información que, en general, no puede ser utilizada directamente por el software. Por ejemplo, un cliente no conoce el código con el que se maneja, internamente, una cuenta.
- Las '*flechas*' entrantes en una clase son invocaciones a una operación. Significa que esa clase tiene esa operación, conoce y maneja la información de los tipos de los argumentos que acompañan a la invocación, así como del valor que devuelve.



Antes de escribir los contratos de las operaciones, vamos a proseguir con el razonamiento.

En esta situación se conoce al cliente, clientID (hay una sesión activa), se conoce la cuenta sobre la que se va a operar, cuentaOrigenID pero no toda la información que interesa, puesto que no se ha seleccionado ninguna operación y no se conocen los cargos aplicables. Por tanto no tiene ningún sentido haber creado ya la 'vista' de la cuentaOrigen.

Se van a utilizar estos criterios:

- Se crearán y utilizarán objetos o recursos con la mínima información y contenidos necesarios.
- La creación o disposición de esos objetos se hará en el instante en el que se necesite utilizarlos.

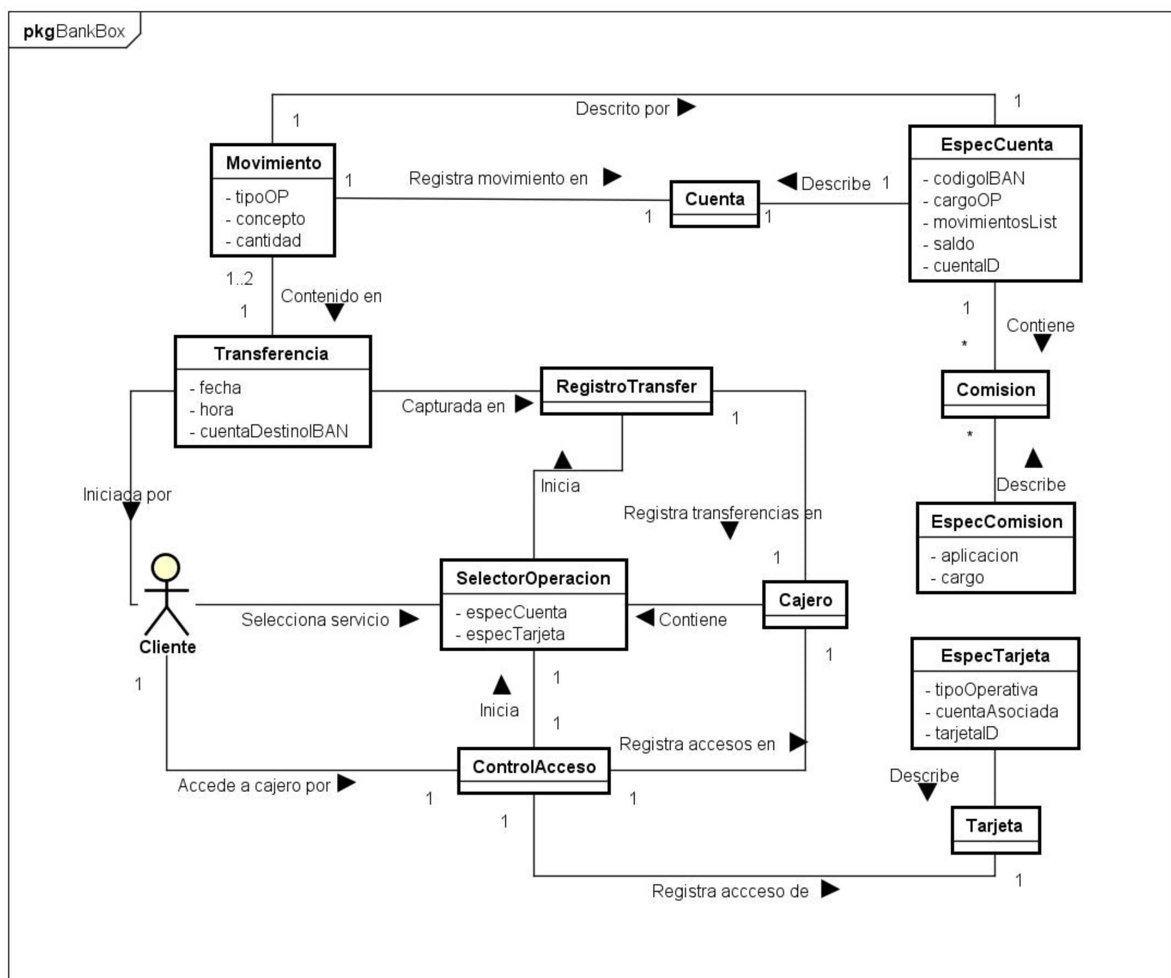
Y esto nos lleva a la pregunta de antes: ¿Qué se necesita?

En el caso de PdV, una Venta es una colección de intercambios: de cada artículo por su precio. Es decir, Venta opera con productos y su precio. Forzando artificialmente la analogía, Transferencia opera con las acciones que se realizan en 2 cuentas (cambio de saldo, anotaciones, etc.) y lo que cuesta hacerlas (los cargos). De todas formas, la analogía no parece adecuada porque, aunque se hayan identificado los elementos con los que operan, la forma en que lo hacen es diferente.

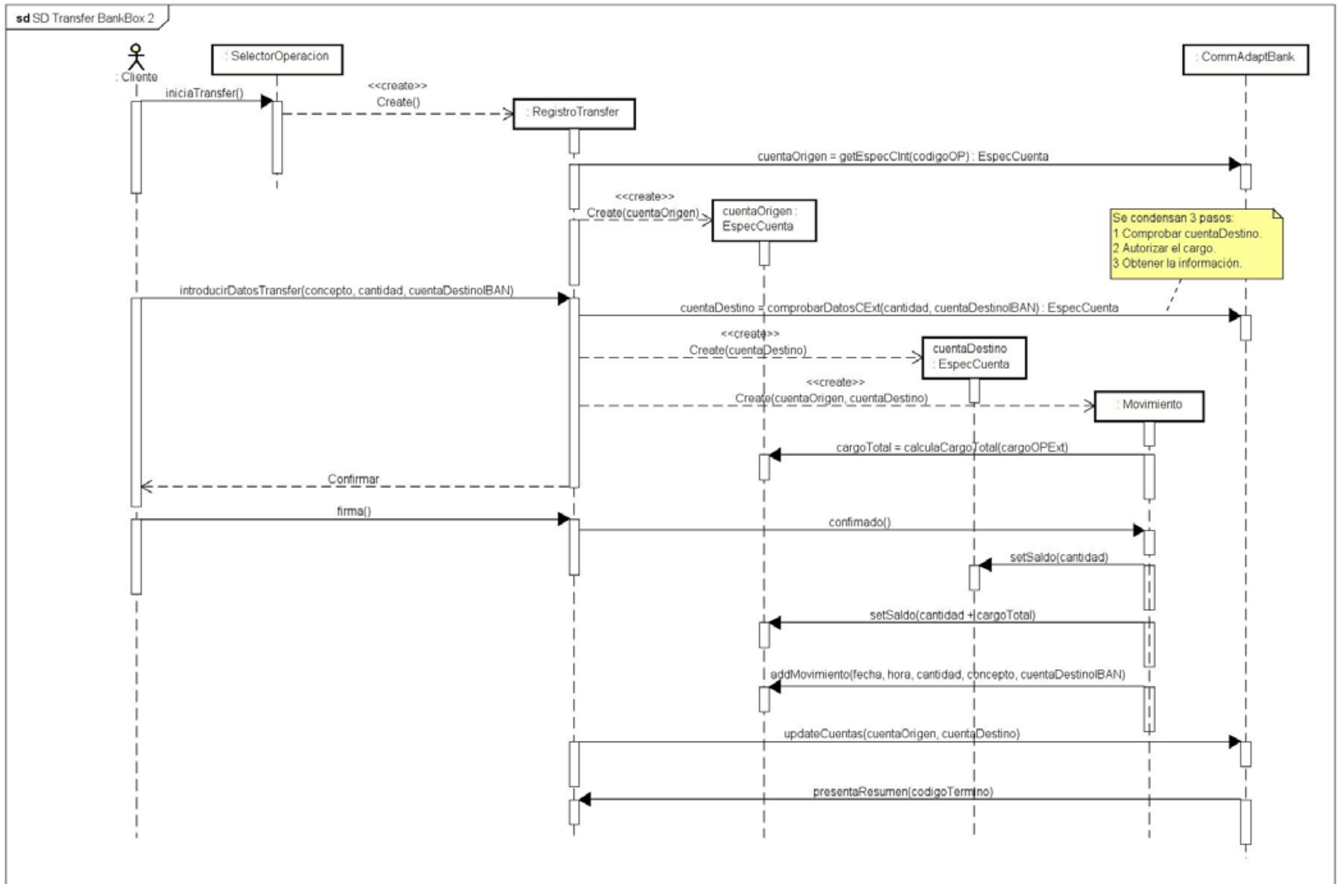
También se puede ver de esta forma: Transferencia opera con cuentas (es evidente; aunque, mejor, con la 'vista' de las cuentas EspecCuenta) y cada acción se va a llamar 'Movimiento' en cuenta.

Como se ve en el DS anterior, aunque los mensajes tienen coherencia y la operación de transferencia podría llegarse a realizar, el no entender cuál es el papel de Transferencia frente a las cuentas, lleva a que sea el Registro el que haga la mayor parte de las operaciones. La incorporación de Transferencia y Movimiento en el modelo, permiten generalizar Registro y desacoplarlo.

Ahora, se puede plantear otro Modelo de Dominio, ajustado al caso de uso:



Y, aplicando las últimas conclusiones al cómo, a la asignación de responsabilidades, se obtiene el siguiente DS:



A partir de este DS, especifique los contratos de **dos** operaciones principales: *<<operación A>>* y *<<operación B>>*.

Contrato CO1: seleccionarTransferencia

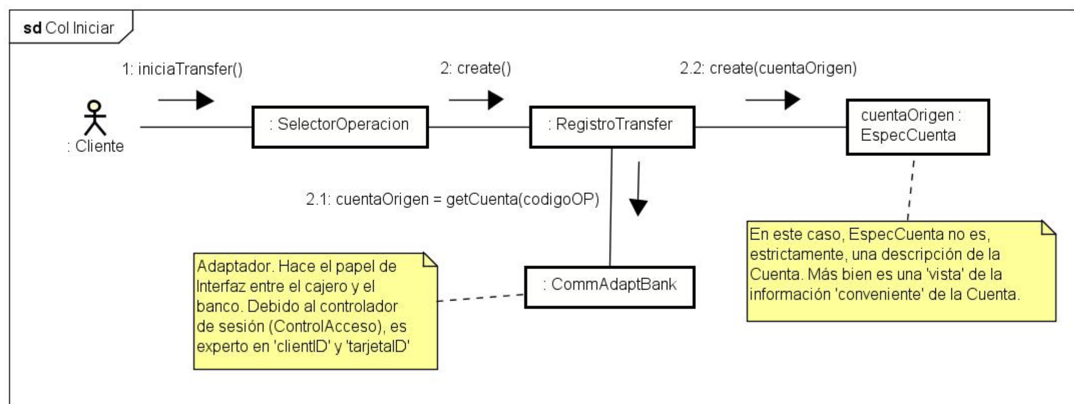
Operación:	iniciaTransfer()
Referencias cruzadas:	Caso de Uso: Transferencia.
Precondiciones:	– Hay una sesión activa, **registroCajero**. – Hay una instancia de CommAdaptBank, **catalogosBanco**. – Hay una instancia de cliente, clienteID, de tarjeta, tarjetaID y se ha creado una lista de servicios, **servicesList**, a los que tiene acceso. – Hay un SelectorOperacion, **selectorOP**, que maneja la selección en **servicesList**.
Postcondiciones:	– Se creó una instancia de RegistroOP, **registroTransfer**. – Se creó una instancia de Transferencia, **transferencia**, asociada a **registroTransfer**. – **catalogosBanco** se asoció a **registroTransfer**. – **codigoOP**, atributo de **registroTrasnfer**, pasó a valer **TRF**, en virtud de la selección. – Se creó una instancia de **EspecCuenta**, **cuentaOrigen**, en virtud de una búsqueda en **catalogosBanco**.

Contrato CO2: introducirDatosTransferencia

Operación:	introducirDatosTransfer(concepto, cantidad, cuentaDestinoIBAN)
Referencias cruzadas:	Caso de Uso: Transferencia.
Precondiciones:	– Hay un **registroTransfer** activo, con una instancia **transferencia** asociada a él. – Hay una instancia de **EspecCuenta**, **cuentaOrigen**. – Se han aportado los datos **concepto**, **cantidad** y **cuentaDestinoIBAN**.
Postcondiciones:	– Se creó una instancia de **EspecCuenta**, **cuentaDest**, en virtud de una búsqueda en **catalogosBanco**. – Se crearon 2 instancias de Movimiento. – Se modificaron los atributos de **cargo**, **saldo** y **movimientosList** en **cuentaDest** y **cuentaOrigen**.

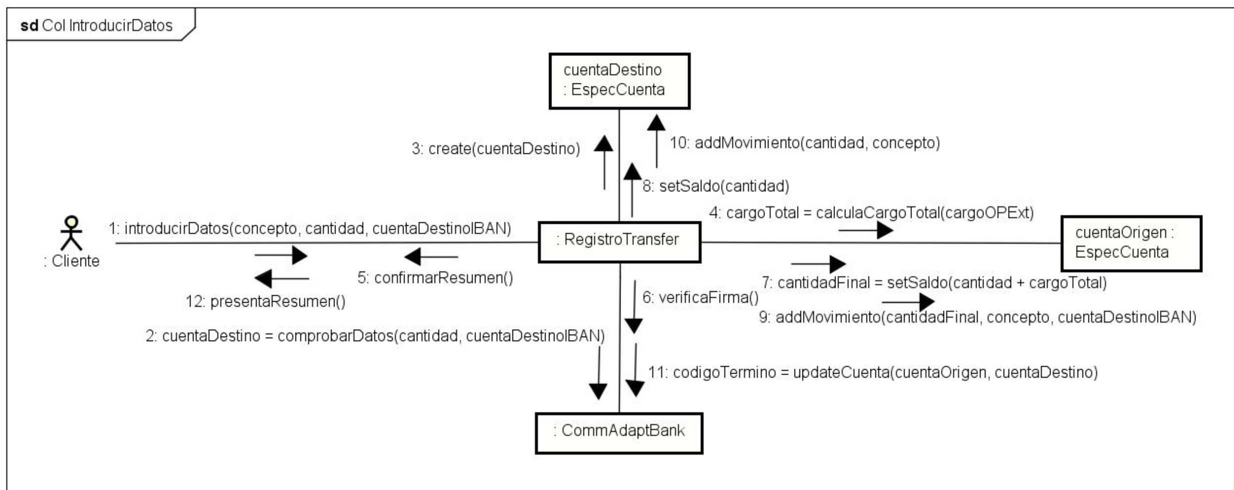
Sección 4. Evaluación de la ***Asignación de Responsabilidades*** y ***Diseño de Colaboraciones***

5. (2 puntos) A partir del contrato de la operación <<*iniciaTransfer*>> que haya indicado en el punto 4, complete el diagrama de colaboración en UML. Consigne cada mensaje con los patrones GRASP (Experto, Creador, etc.) o cualquier otro que lo justifique. Si añade responsabilidades no explicitadas en el contrato (porque crea que es importante señalarlas), explíquelas brevemente.

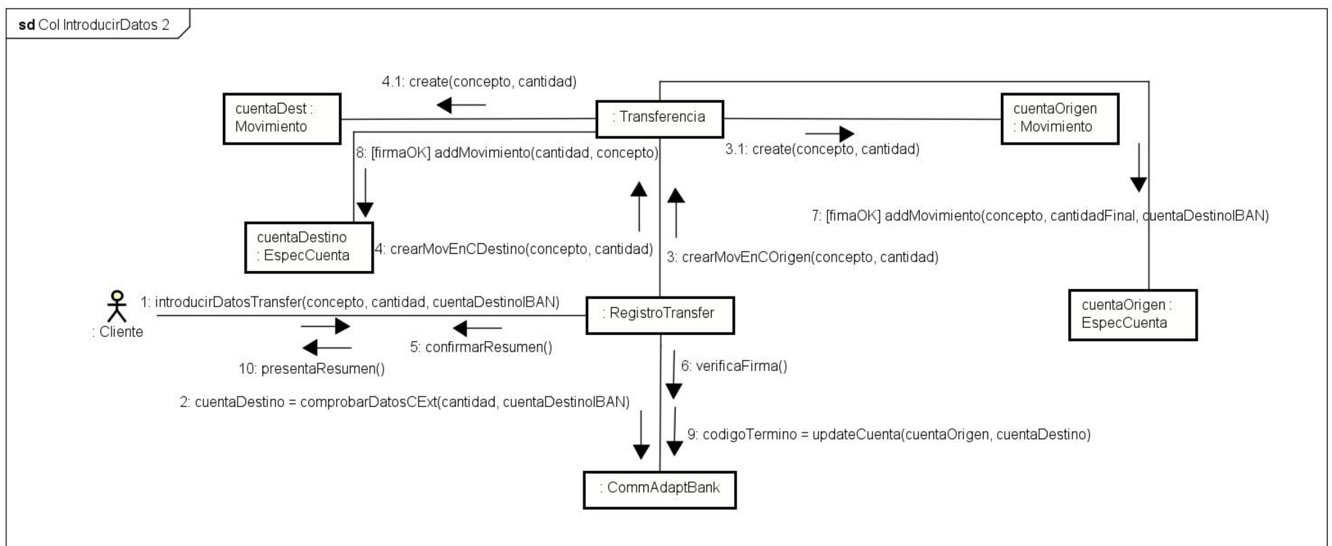


6. (2 puntos) A partir del contrato de la operación <<*introducirDatosTransfer*>> que haya indicado en el punto 4, complete el diagrama de colaboración en UML. Consigne cada mensaje con los patrones GRASP (Experto, Creador, etc.) o cualquier otro que lo justifique. Si añade responsabilidades no explicitadas en el contrato (porque crea que es importante señalarlas), explíquelas brevemente.

Con el planteamiento inicial, este podría ser el diagrama de colaboración:

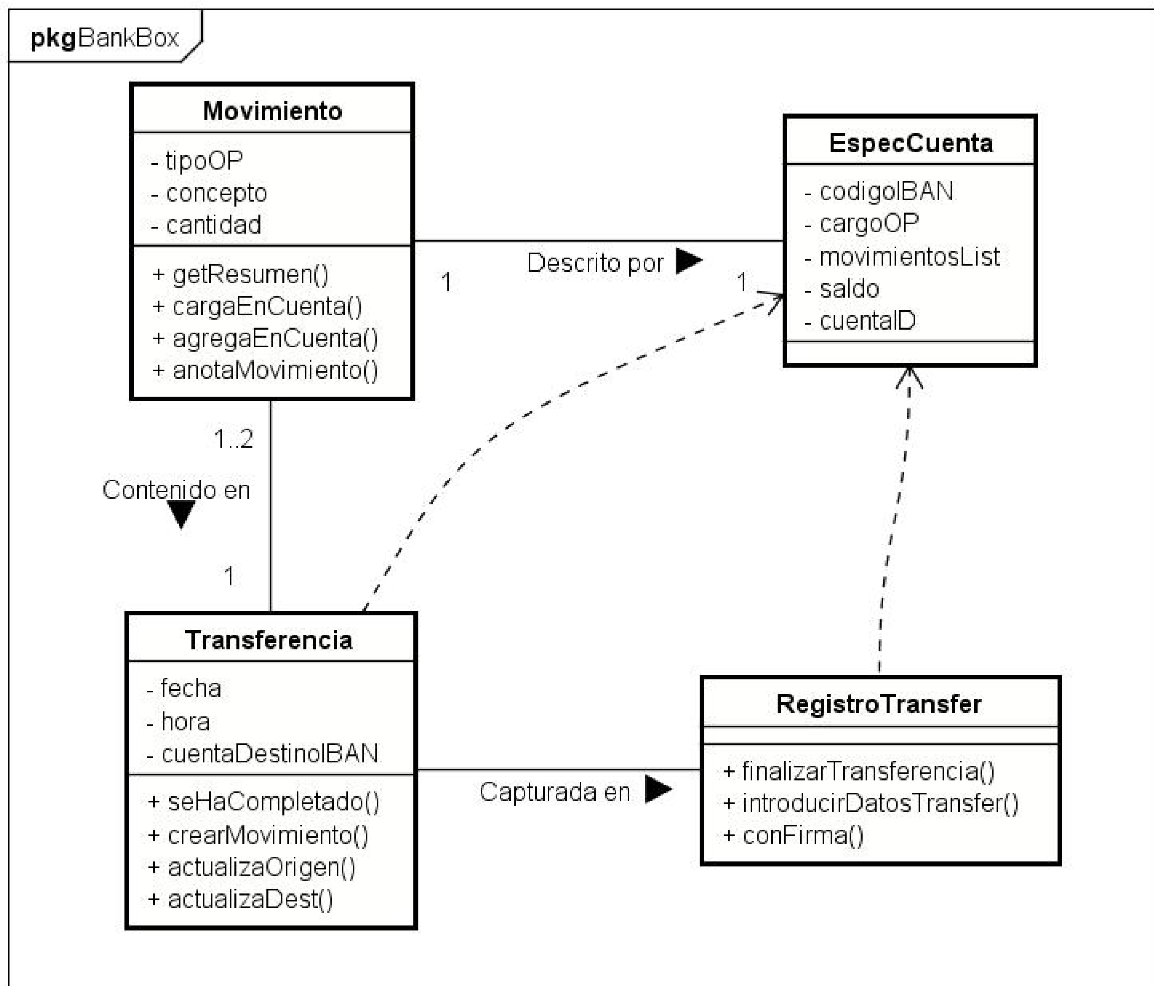


Y con el último:



Sección 5. Evaluación de los ***Diagramas de Clases*** de diseño

7. (0'5 puntos) Elabore un diagrama de clases para el caso de uso que se está tratando <<*Transferencia*>> (DCD), centrado en la clase cuya responsabilidad es registrar los datos de esa operación. Represente los nombres de todos sus atributos, asociaciones (con la navegabilidad) y métodos.



Sección 6. Evaluación de la ***Transformación del Diseño en Código***

8. (0'5 puntos) A partir de los anteriores diagramas de clases y colaboraciones, elabore y defina la clase que haya definido, en el desarrollo anterior, como responsable de <u>registrar</u> los datos de la operación *<<TransferirDinero>>*. Incluya las definiciones de todas las variables que la componen (miembros), pero escriba solamente la definición completa del cuerpo para el método (o métodos) principal o más significativo: *<<se omite el método>>*. Ignore los pequeños detalles de sintaxis -el objetivo es evaluar la capacidad fundamental para transformar el diseño en código-. Utilice la sintaxis de Java.

Registro de operación en la sesión:

Clase RegistroTransfer

```
public class RegistroTransfer extends RegistroOP
{
    private CommAdaptBank catalogosBanco;
    private CuentaID cuentaOrigen;
    private List logEvents = new ArrayList();
    private CodigoOP codigoTransf = new CodigoOP(TRF);

    private EspecCuenta especOrigen, especDest;
    private Transferencia transfer;

    public RegistroTransfer (CommAdaptBank catalogosBanco, CuentaID cuentaOrigen List
    logEvents)
    {
        this.catalogosBanco = catalogosBanco;
        this.cuentaOrigen = cuentaOrigen;
        this.logEvents = logEvents;
        transfer = new Transferencia();

    }
```

```
public void finalizarTransferencia ()
{
    private List resumen = new ArrayList();
    private Map event = new HashMap();

    resumen = transfer.presentaResumen();
    SignCode firma = conFirmaOP(resumen);
    // Comprueba firma … catalogosBanco.comprobarFirma(firma);
    // Si firma OK…
    event.put( firma, resumen );
    logEvents.add( event );

    especOrigen = transfer.actualizaOrigen();
    especDest = transfer.actualizaDest();
    catalogosBanco.updateCuentas(especOrigen, especDest);

    transfer.seHaCompletado();
    resumen = transfer.presentaResumen();
}

public SignCode conFirmaOP(resumen)
{
    // Por ejemplo…
    private Scanner sign = new Scanner(System.in);
    private String firma;

    // Imprime resumen…

    firma = sign.next();
    return new SignCode(firma);
}

public void introducirDatosTransfer(String concepto, Dinero cantidad, CuentaID cuentaDestinoIBAN)
{
    especOrigen = catalogosBanco.getEspecCInt(cuentaOrigen, codigoTransf);
    especDest = catalogosBanco.comprobarDatosCExt
                            (cuentaOrigen, cantidad, cuentaDestinoIBAN, codigoTransf);
    // Calcula cargos
    especOrigen.addCargo( especDest.getCargo() );

    transfer.crearMovimiento(especOrigen, codigoTransf, concepto, cantidad, cuentaDest);
    transfer.crearMovimiento(especDest, codigoTransf, concepto, cantidad, cuentaOrigen);

    finalizarTransferencia();
}
}
```

Clase EspecCuenta

```
public class EspecCuenta
{
    private CuentaID codigoIBAN;
    private Dinero cargoOP;
    private Dinero saldo;
    private ArrayList movimientosList;
    private CuentaID cuentaID;

    public EspecCuenta
        (CuentaID codigoIBAN, Dinero cargoOP, Dinero saldo, ArrayList movimientosList, CuentaID cuentaID)
    {
        this.codigoIBAN = codigoIBAN;
        this.cargoOP = cargoOP;
        this.saldo = saldo;
        this.movimientosList = movimientosList;
        this.cuentaID = cuentaID;

    }

    public CuentaID getCodigoIBAN()
    {
        return codigoIBAN;
    }

    public Dinero getCargoOP()
    {
        return cargoOP;
    }

    public void addCargo(Dinero cantidad)
    {
        cargoOP = cargoOP.add(cantidad);
    }

    public Dinero getSaldo()
    {
        return saldo;
    }

    public void setSaldo(Dinero cantidad)
    {
        saldo = new Dinero(cantidad);
    }
```

```
    public ArrayList getMovimientos()
    {
        return movimientosList;
    }

    public void setMovimientos(ArrayList nuevaMovimientosList)
    {
        movimientosList = nuevaMovimientosList;
    }

    public CuentaID getCuentaID()
    {
        return cuentaID;
    }
}
```

Clase Transferencia

```
public class Transferencia
{
    private List movimientos = new ArrayList();
    private Date fecha = new Date();
    private boolean esCompleta() = false;

    public void seHaCompletado()
    {
        esCompleta = true;
    }

    public void esCompleta()
    {
        return esCompleta;
    }

    public void crearMovimiento(EspecCuenta cuentaObjeto, CodigoOP codigoTransf, String
    concepto, Dinero cantidad, EspecCuenta cuentaOrig)
    {
        private Dinero cargo = cuentaObjeto.getCargo();

        movimientos.add( new Movimiento(cuentaObjeto, codigoTransf, fecha,
                                        concepto, cantidad, cargo, cuentaOrig) );
    }

    public ArrayList presentaResumen()
    {
        private Movimiento movimientoObjeto = movimientos.get(0);

        return movimientoObjeto.getResumen();
    }
```

```
    public EspecCuenta actualizaOrigen()
    {
        private EspecCuenta cuentaObjeto;
        private Movimiento movimientoObjeto = movimientos.get(0);

        cuentaObjeto = movimientoObjeto.cargaEnCuenta();
        cuentaObjeto = movimientoObjeto.anotaMovimiento();
        return cuentaObjeto;
    }

    public EspecCuenta actualizaDest()
    {
        private EspecCuenta cuentaObjeto;
        private Movimiento movimientoObjeto = movimientos.get(1);

        cuentaObjeto = movimientoObjeto.agregaEnCuenta();
        cuentaObjeto = movimientoObjeto.anotaMovimiento();
        return cuentaObjeto;
    }
}
```

Clase Movimiento

```
public class Movimiento
{
    private EspecCuenta cuentaObjeto;
    private CodigoOP codigoOP;
    private Date fecha;
    private String concepto;
    private Dinero cantidad;
    private Dinero cargo;
    private EspecCuenta cuentaOrig;
    private List resumen = new ArrayList();

    public Movimiento (EspecCuenta cuentaObjeto, CodigoOP codigoOP, Date fecha; String concepto,
                                                   Dinero cantidad, Dinero cargo, EspecCuenta cuentaOrig)
    {
        this.cuentaObjeto = cuentaObjeto;
        this.codigoOP = codigoOP;
        this.fecha = fecha;
        this.concepto = concepto;
        this.cantidad = cantidad;
        this.cargo = cargo;
        this.cuentaOrig = cuentaOrig;
    }
```

```
public void resetResumen()
{
    private List newResumen = new ArrayList();

    // Construye el resumen
    newResumen.add( cuentaObjeto.getCodigoIBAN() )
    newResumen.add(codigoOP)
    newResumen.add(fecha)
    newResumen.add(concepto)
    newResumen.add(cantidad)
    newResumen.add(cargo)
    newResumen.add( cuentaOrig.getCodigoIBAN() )

    resumen = newResumen;
}

public ArrayList getResumen()
{
    resetResumen();
    return resumen;
}

public EspecCuenta cargaEnCuenta()
{
    private Dinero nuevoSaldo;

    nuevoSaldo = saldo.minus( cantidad.add( cargo ) );
    cuentaObjeto.setSaldo(nuevoSaldo);
    resetResumen;
    return cuentaObjeto;
}

public EspecCuenta agregaEnCuenta()
{
    private Dinero nuevoSaldo;

    nuevoSaldo = saldo.add( cantidad ) );
    cuentaObjeto.setSaldo(nuevoSaldo);
    resetResumen();
    return cuentaObjeto;
}
```

```
    public EspecCuenta anotaMovimiento()
    {
        private List movimientosList = new ArrayList();

        movimientosList = cuentaObjeto.getMovimientos();
        resetResumen();
        movimientosList.add( resumen );
        cuentaObjeto.setMovimientos(movimientosList);
        return cuentaObjeto;
    }
}
```

Controlador de operación en la sesión:

Clase SelectorOperacion

```
public class SelectorOperacion
{
    private CommAdaptBank conexionBanco;
    private ClientID cliente;
    private CuentaID cuentaOrigen;
    private List servicesList = new ArrayList();
    private List logEvents = new ArrayList();
    private CodigoOP codigoOP;
    private RegistroOP registroOP;

    public SelectorOperacion (CommAdaptBank conexionBanco, ClientID cliente, CuentaID
    cuentaOrigen, ArrayList servicesList, ArrayList logEvents)
    {
        this.conexionBanco = conexionBanco;
        this.cliente = cliente;
        this.cuentaOrigen = cuentaOrigen;
        this.servicesList = servicesList;
        this.logEvents = logEvents;
    }

    public void crearNuevaOperacion(CodigoOP codigoOP)
    {
        private Map event = new HashMap();

        event.put (cliente, codigoOP)
        logEvents.add(event);
        // Genérico:
        //  registroOP = new RegistroOP(conexionBanco, cliente, cuentaOrigen, logEvents, codigoOP);
        if ( codigoOP.getTextCode == "TRF" )
        {
            registroOP = new RegistroTransfer(conexionBanco, cuentaOrigen, logEvents);
        }
    }
```

```
    public RegistroOP getRegistroOP()
    {
        return registroOP;
    }
}
```

Clase CommAdaptBank

```
public class CommAdaptBank
{
    private Bank banco;
    private AuthCode codigoAuth;

    public CommAdaptBank (Bank entidad)
    {
        this.banco = entidad;
    }

    public boolean getAuth(ClientID cliente)
    {
        codigoAuth = banco.authorize(cliente)
        return parse(codigoAuth);
    }

    public HashMap obtenerServicios(ClientID cliente, TarjetaID tarjeta)
    {
        return banco.findServices(codigoAuth, tarjeta);
    }

    public EspecCuenta getEspecClnt(CuentaID cuentaOrigen, CodigoOP codigoOP)
    {
        return banco.getVistaCuenta(cuentaOrigen, codigoOP);
    }

    public EspecCuenta comprobarDatosCExt
                    (CuentaID cuentaOrigen, Dinero cantidad, cuentaDestinoIBAN, CodigoOP codigoOP)
    {
        return banco.getVistaCuentaExt(cuentaOrigen, cantidad, codigoOP);
    }

    public boolean comprobarFirma(firma)
    {
        …
    }

    public void updateCuentas(EspecCuenta especOrigen, EspecCuenta especDest)
    {
        …
    }
```

```
}
```

Junto con **CommAdaptBank**, el '*envoltorio*' del caso de uso:

Controlador/Registro de sesión del cajero:

Clase ControlAcceso

```
public class ControlAcceso
{
    private CommAdaptBank conexionBanco;
    private List logSession = new ArrayList();
    private Acceso acceso;
    private ClientID clienteAuth;
    private TarjetaID tarjetaAcceso;
    private CuentaID cuentaOrigen;
    private SelectorOperacion selectorOP;
    private RegistroOperacion registroOP;

    public ControlAcceso (CommAdaptBank conexion)
    {
        this.conexionBanco = conexion;
    }

    public void finalizarAcceso ()
    {
        acceso.seHaCompletado();
    }

    public RegistroOP getRegistroAcc()
    {
        if ( acceso.esCompleto() )
        {
          registroOP = registroOP.getRegistroOP();
          return registroOP;
        }
        else { return null;}
    }
```

```
    public void crearNuevoAcceso(ClientID cliente)
    {
        private Map event = new HashMap();
        boolean auth = conexionBanco.getAuth(cliente);
        acceso = new Acceso();
        if ( auth )
        {
          clienteAuth = cliente;
          acceso.seHaAutorizado();
        }
        else
        {
          finalizarAcceso();
        }
        event.put(cliente, acceso);
        logSession.add(event);
    }

    public void crearNuevaTarjeta(TarjetaID tarjeta)
    {
        private Map event = new HashMap();
        private Map cuenta_servicios = new HashMap();
        private List servicesList = new ArrayList();

        cuenta_servicios = conexionBanco.obtenerServicios(clienteAuth, tarjeta);
        cuentaOrigen = (CuentaID)cuenta_servicios.keySet();
        servicesList = cuenta_servicios.get(cuentaOrigen);

        event.put(tarjeta, cuentaOrigen);
        logSession.add(event);

        selectorOP = new SelectorOperacion(conexionBanco, logSession, servicesList);
        logSession.addAll(selectorOP.getRegistroEv);
        registroOP = selectorOP.getRegistro();
    }
}
```

Clase Acceso

```
public class Acceso
{
    private Date fecha = new Date();
    private boolean esCompleto() = false;
    private boolean autorizado() = false;

    public void seHaCompletado()
    {
        esCompleto = true;
    }
```

```
    public void esCompleto()
    {
        return esCompleto;
    }

    public void seHaAutorizado()
    {
        autorizado = true;
    }

    public void autorizado()
    {
        return autorizado;
    }
}
```

Lo inicia la interfaz del servidor del banco:

Clase Cajero

```
public class Cajero
{
    // Interfaz de la parte del banco
    private Bank banco;
    // Crea la interfaz de la parte del cajero
    private CommAdaptBank conexionBanco = new CommAdaptBank(banco);
    // Crea el Registro del cajero (ControlAcceso) y le pasa la interfaz
    private ControlAcceso registroCajero = new ControlAcceso(conexionBanco);

    public Cajero (Bank entidad)
    {
        this.banco = entidad;
    }

    public ControlAcceso getRegistroCajero()
    {
        registroCajero = registroCajero.getRegistroAcc();
        return registroCajero;
    }
}
```

A continuación se incluye un posible desarrollo para la clase *Dinero*, con abstracción para la gestión de la divisa.

Clase Dinero

```
import java.util.*;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.math.MathContext;

/**
 * Class Dinero - clase la divisa de los pagos.
 * Escrito en BlueJ para el examen de febrero 2013.
 * @author  José Félix Estívariz
 * @version 1.0, marzo 2013
 */
public class Dinero
{
    // inicializa variables de la instancia
    private double cantidad;
    private Locale pais;            //init-> = new Locale("es");
    private Currency divisa;        // init-> = Currency.getInstance(pais);

    /**
     * Constructor para objetos de la clase Dinero
     */
    public Dinero ( double nuevaCantidad )
    {
        // inicializa variables de la instancia
        this.cantidad = nuevaCantidad;
        this.divisa = divisa;
    }

    /**
     * Métodos.
     */
    /**
     * Cambia la cantidad de dinero.
     */
    public void setCantidad ( double nuevaCantidad )
    {
        cantidad = nuevaCantidad;
    }
```

```
/**
 * Añade una cantidad de dinero.
 */
public Dinero add(Dinero nuevaCantidad)
{
    //cantidad += nuevaCantidad;
    //        |de double a BigDecimal|     |de double a BigDecimal| |de Dinero a double| |de BigDecimal a double|
    cantidad = (new BigDecimal(cantidad)).add(new BigDecimal(nuevaCantidad.getCantidad())).doubleValue();
    return new Dinero(cantidad);
}

/**
 * Resta una cantidad de dinero.
 */
public Dinero minus(Dinero nuevaCantidad)
{
    //cantidad -= nuevaCantidad;
    //            |de double a BigDecimal|     |de double a BigDecimal| |de Dinero a double| |de BigDecimal a double|
    cantidad = (new BigDecimal(cantidad)).subtract(new BigDecimal(nuevaCantidad.getCantidad())).doubleValue();
    return new Dinero(cantidad);
}

/**
 * Multiplica la cantidad de dinero.
 */
public Dinero times(int veces)
{
    // BigDecimal veces2 = new BigDecimal(veces, new MathContext(2));
    // BigDecimal cantidad2 = new BigDecimal(cantidad);
    // cantidad2 = cantidad2.multiply(veces2);
    // cantidad = cantidad2.doubleValue();
    cantidad = (new BigDecimal(cantidad)).multiply(new BigDecimal(veces, new MathContext(2))).doubleValue();
    return new Dinero(cantidad);
}

/**
 * Devuelve la cantidad de dinero.
 */
public double getCantidad()
{
    return cantidad;
}

/**
 * Cambia la divisa.
 */
public void setDivisa(String nuevoPais)
{
    pais = new Locale(nuevoPais);
    divisa = Currency.getInstance(pais);
}
```

```
    /**
     * Devuelve la divisa de Dinero.
     */
    public String getDivisa()
    {
        return divisa.getCurrencyCode();
    }
}
```