# Descripción detallada de una solución en C#

Macario Polo Usaola
Escuela Superior de Informática
Universidad de Castilla-La Mancha
macario.polo@uclm.es

## Índice

# 1 Descripción de la solución

En .NET, una solución es un conjunto formado por uno o más proyectos. La solución adjunta incluye tres proyectos:

- banco1, que simula un sistema de gestión bancario que permite:
  - Crear cuentas corrientes.
  - Crear y tarjetas de débito y crédito y asociarlas a las cuentas corrientes.
  - Realizar las operaciones de ingreso, retirada y transferencia con las cuentas corrientes.
  - Poner en marcha un Servidor, que permite que desde el otro proyecto (Cajero) se puedan realizar operaciones.
- Cajero, que simula un cajero automático. Realiza las operaciones contra el Servidor incluido en el proyecto banco1, con el que se comunica mediante sockets.
- ControlNumeroDeCuenta, en donde se implementa un control de usuario que sirve para manejar cómodamente los números de cuenta (compuestos de los campos Entidad, Sucursal, Dígito de Control y Código Cuenta Cliente).

## *1.1 Proyecto "banco1"*

La Figura 1 muestra los ficheros contenidos en el proyecto *banco1*. Los ficheros están distribuidos en varios subsistemas, que se corresponden con diferentes Namespaces de C#:

- En *presentacion* se encuentran todas las ventanas de la aplicación, que son las que utilizan los empleados del banco para realizar las operaciones:
  - FMenu es la ventana por la que arranca este proyecto. Incluye una barra de menús para acceder a todas las funcionalidades de la aplicación.
  - FCuenta permite crear, buscar, cancelar, bloquear y desbloquear cuentas, retirar, ingresar y realizar transferencias a otras cuentas del banco, mostrar la ventana que permite crear tarjetas para asociarlas a la cuenta que se está mostrando, así como mostrar la lista de tarjetas asociadas a esta cuenta.

- FLista es una ventana de la que heredan FListaDeCuentas y FListaDeTarjetas. Incluye un control ListView para mostrar listados.
- FTarjeta se utiliza para crear tarjetas de crédito y débito, que estarán asociadas a una cuenta.

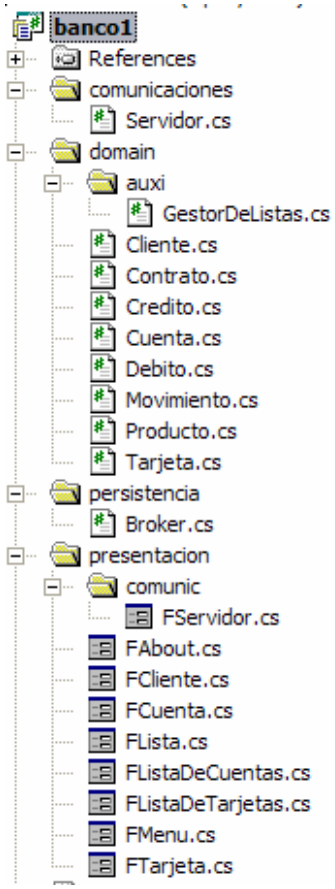



**Figura 1. Ficheros incluidos en el proyecto banco1**

- *presentacion/comunic* incluye FServidor, que es una ventana que permite lanzar el servidor, que escucha las peticiones recibidas desde el cajero automático, contenido en el proyecto Cajero.
- *domain* contiene las clases de "dominio": es decir, las clases que forman parte del enunciado del problema, aquellas que incluyen la lógica para resolverlo. Las ventanas (contenidas en *presentacion*) son meros transmisores de mensajes entre el usuario y la capa de dominio.

- *domain/auxi* contiene GestorDeListas, una clase que se utiliza para recuperar listados de registros de la base de datos. Esta clase es utilizada por las ventanas que heredan de FLista.
- *persistencia* contiene Broker, una clase que centraliza el acceso a la base de datos. Cuando una clase de dominio desea realizar una operación sobre la base de datos, la realiza a través del Broker. El Broker responde a lo que se llama un "agente de base de datos".
- *comunicaciones* contiene la clase Servidor, que representa el servidor que acepta conexiones desde los cajeros automáticos.

### 1.1.1 Correspondencia entre la base de datos y el modelo de clases

Como se ha mencionado, la aplicación utiliza una base de datos que se encuentra implementada en Microsoft Access. Su estructura es la que se muestra en la Figura 2.

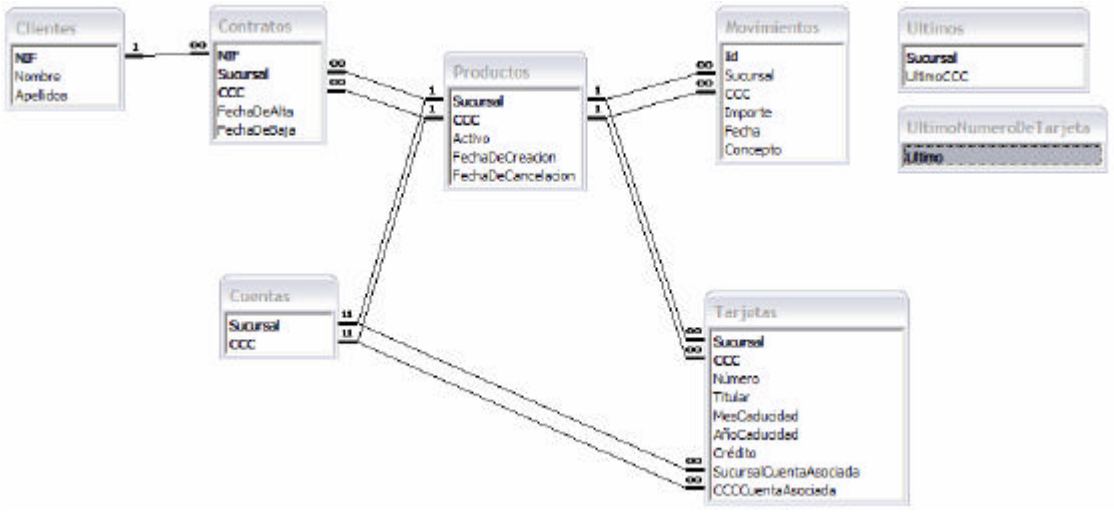



**Figura 2. Esquema de la base de datos utilizada**

Esta base de datos procede de la transformación del modelo de clases que se muestra en la Figura 3. Como se ve, la relación de muchos a muchos que aparece en la Figura 3 entre Cliente y Producto (y que representa el hecho de que un cliente contrata uno de los productos que oferta nuestro banco) se ha traducido, en el modelo relacional, a tres tablas: una para Cliente, otra para Producto y otra que representa la propia relación de muchos a muchos (Contrato). La relación existente en el modelo de clases se traduce a

relaciones de clave externa entre las tablas, estableciéndose tales relaciones usando la clave primaria de cada tabla (o claves alternativas).

Las dos relaciones de herencia se han traducido a dos relaciones de clave externa con multiplicidad uno en ambos lados[1]. Las relaciones de uno a muchos entre Cuenta y Tarjeta y entre Producto y Movimiento que aparecen en la Figura 3 se traducen a relaciones de clave externa entre las correspondientes tablas.

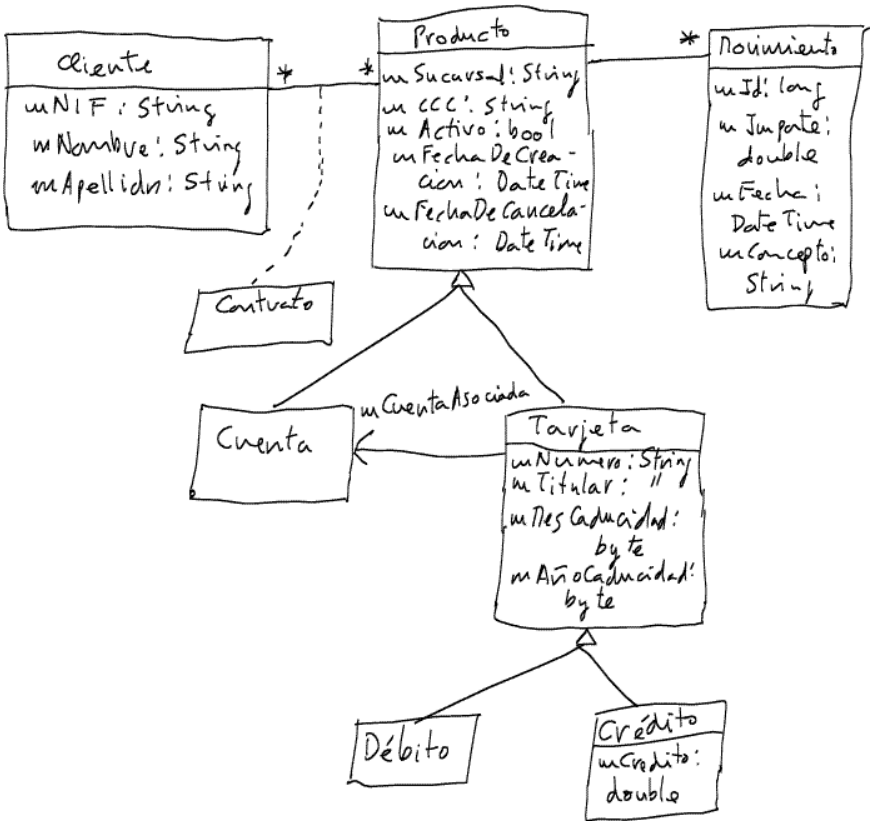



**Figura 3. Modelo de clases de la capa de dominio**

La relación de herencia entre Tarjeta y sus dos especializaciones (Débito y Crédito) se ha traducido a una sola tabla, ya que la diferencia de estructura entre ambos tipos de tarjeta es muy pequeña, difiriendo tan sólo en el campo "mCrédito". Así, cuando se vaya a introducir una tarjeta de Débito en la tabla Tarjetas, se asignará valor cero a la columna Crédito, y un valor positivo cuando se trate de una tarjeta de crédito.

[1] Aunque erróneamente aparece infinito en el extremo correspondiente a Tarjetas.

Existen dos tablas adicionales (Ultimos y UltimoNumeroDeTarjeta) que almacenan los últimos números de cuenta y de tarjeta concedidos por el banco. Se utilizan cuando se va a crear un nuevo producto y una nueva tarjeta. En Ultimo se almacenan dos columnas: la combinación de ambas representa el último número de cuenta (columna UltimoCCC) concedido en la sucursal indicada en la primera columna (Sucursal). UltimoNumeroDeTarjeta almacena el último número de tarjeta concedido por el banco.

### 1.1.2 Detalles en la construcción de la base de datos

En el diseño de la base de datos se debe tener cuidado a la hora de establecer las restricciones de los valores que pueden tomar las columnas. Además de indicar adecuadamente las columnas que componen la clave principal en cada tabla, es preciso establecer restricciones *check*, mensajes de error, claves alternativas, etc.

Todos los productos del banco están identificados por su número de Sucursal y su número CCC, de cuatro y diez dígitos, respectivamente. Pero ocurre, por ejemplo, que las tarjetas tienen, además, un número de dieciséis dígitos que identifica unívocamente a cada tarjeta, de modo que no puede haber dos tarjetas con el mismo número. A la hora de escribir la aplicación, podríamos escribir un método que comprobara que, al insertar una tarjeta, el número no exista. Sin embargo, es más sencillo indicar en la base de datos que la columna Número es una clave alternativa (índice único). En cada gestor de base de datos, esto se realiza de una manera diferente.

Igualmente, es conveniente establecer también las longitudes máximas y mínimas de los valores que puede almacenar cada columna: por ejemplo, decirle a la base de datos que la columna Número de Tarjetas tiene que tener dieciséis dígitos exactamente.

De forma general, resulta más sencillo y seguro indicar todas estas restricciones en la base de datos, en lugar de hacerlo en el programa: éste enviará mensajes a la base de datos, y capturará las excepciones que se vayan produciendo.

### 1.1.3 Descripción de las clases incluidas en *domain*

#### 1.1.3.1 Producto

La clase Producto representa uno cualquiera de los productos que oferta el banco: en principio sólo cuentas y tarjetas, pero también podrían ser préstamos, fondos de

inversión, etc. No existen productos como tales, sino sus especializaciones, por lo que se establece Producto como una clase abstracta. A la hora de traducir a código la clase Producto (nos fijamos ahora en el modelo de clases mostrado en la Figura 3), los campos "simples" se traducen a su correspondiente tipo de datos en C#; las asociaciones se traducen, por lo general, a campos que se corresponden con algún tipo de colección (ArrayList, por ejemplo). Con estas consideraciones, los campos de la clase Producto son los siguientes:

```
protected static String mEntidad="1000";
protected String mSucursal, mCCC;
protected ArrayList mMovimientos, mTitulares;
protected bool mActivo;
protected DateTime mFechaDeCreacion, mFechaDeCancelacion;
```

**Fragmento de código 1. Campos de la clase Producto**

Como se observa, hemos añadido el campo "mEntidad", que representa el código del banco. Puesto que todos los productos de nuestro banco tienen el mismo código, declaramos este campo como "static", de modo que todas las instancias de banco compartan este valor. Los productos bancarios se identifican también por un "dígito de control", formado realmente por dos dígitos. Éste es un valor que se calcula en función del código de entidad (mEntidad), de la sucursal (mSucursal) y del "código cuenta cliente" (campo mCCC), por lo que, en lugar de crear un campo "mDC", creamos un método o una propiedad que devuelva este valor. En este caso hemos optado por crear una propiedad (Fragmento de código 2) que llame a un método[2] (Fragmento de código 3).

```
public String digitoDeControl
{
      get
      {
            return getDigitoDeControl(mSucursal, mCCC);
      }
}
```

**Fragmento de código 2. Propiedad *digitoDeControl* de *Producto***

```
public static String getDigitoDeControl(String sucursal,
                                         String ccc)
{
      if (sucursal.Equals("")) sucursal="0";
      if (ccc.Equals("")) ccc="0";
      int e=int.Parse(mEntidad);
```

[2] El algoritmo que se muestra para calcular el dígito de control no es el auténtico.

```
            int s=int.Parse(sucursal);
            int iccc=int.Parse(ccc);
            int dc=(e+s+iccc) % 25;
            String result=""+dc;
            if (result.Length==1)
                  result="0" + result;
            return result;
      }
```

**Fragmento de código 3.**

Producto incluye también tres constructores:

- Producto(), que no asigna valor a ningún campo.
- Producto(String sucursal), que crea un nuevo producto en la sucursal pasada como parámetro , asignándole un valor nuevo al campo mCCC. Para ello, busca en la tabla Últimos el valor de la columna UltimoCCC correspondiente a la sucursal, le suma uno y lo asigna a mCCC. A continuación, inserta el producto en la tabla Productos y actualiza la tabla Últimos. O sea, que este constructor se utiliza cuando creamos una cuenta corriente nueva.
- Producto(String sucursal, String ccc) se usa para "materializar" un producto: es decir, para crear una instancia de clase Producto a partir de la información que haya en la base de datos. Para ello, busca en la tabla de Productos un registro cuyas columnas Sucursal y CCC tengan los valores pasados como parámetros. Si lo encuentra, asigna los valores del registro a los diferentes campos de la instancia. O sea, que este constructor se utiliza cuando, por ejemplo, llega un cliente y nos dice que quiere realizar alguna operación con su cuenta, que ya existía.

```
public Producto()
{
}

public Producto(String sucursal)
{
   this.mSucursal=sucursal;
   this.mCCC=getCCCSiguiente(sucursal);
   this.mFechaDeCreacion=DateTime.Now;
   insert();
   actualizaCCCSiguiente(sucursal,
                         mCCC);
   mTitulares=new ArrayList();
}
```

```
public Producto(String sucursal,
                String ccc)
{
   String SQL="Select Activo, " +
   "FechaDeCreacion, " +
        "FechaDeCancelacion " +
        "from Productos " +
        "where Sucursal=? and CCC=?";
   OleDbCommand cmd = new
        OleDbCommand(SQL);
   cmd.Parameters.Add("",
   OleDbType.VarChar).Value=sucursal;
   cmd.Parameters.Add("",
        OleDbType.VarChar).Value=ccc;
   cmd.Connection = Broker.getBD();
   OleDbDataReader r=cmd.ExecuteReader();
   if (r.Read())
   {
        mSucursal=sucursal;
        mCCC=ccc;
```

```
        mActivo=r.GetBoolean(0);
        mFechaDeCreacion=r.GetDateTime(1);
        if (!r.IsDBNull(2))
         mFechaDeCancelacion=r.GetDateTime(2);
        mMovimientos=new ArrayList();
        mTitulares=new ArrayList();
        cargaTitulares();
    }
    cmd.Connection.Close();
}
```

**Fragmento de código 4. Constructores de *Producto***

Como se ve en el Fragmento de código 4, el código de Producto(String sucursal) asigna a mSucursal el valor pasado como parámetro. A continuación, asigna a mCCC el resultado de ejecutar la operación getCCCSiguiente(String sucursal) (Fragmento de código 5), que recupera de la tabla Últimos el último valor de CCC utilizado y lo devuelve incrementado en uno. Luego, asigna a la fecha de creación del producto la fecha del sistema e inserta la instancia en la base de datos usando el método insert() (Fragmento de código 6). Por último, actualiza la tabla Últimos con el mCCC de este producto e inicializa el campo mTitulares.

```
protected String getCCCSiguiente(String sucursal)
{
        String SQL="Select UltimoCCC from Ultimos where Sucursal=?";
        OleDbCommand cmd=Broker.getCommand(SQL);
        cmd.Parameters.Add("[sucursal]", OleDbType.VarChar).Value=sucursal;
        OleDbDataReader r=cmd.ExecuteReader();
        String ultimoCCC="";
        if (r.Read())
        {
                ultimoCCC=r.GetString(0);
                double dUltimo=double.Parse(ultimoCCC)+1;
                ultimoCCC=""+dUltimo;
                int longitud=ultimoCCC.Length;
                for (int i=0; i<10-longitud; i++)
                {
                        ultimoCCC="0" + ultimoCCC;
                }
                cmd.Connection.Close();
                return ultimoCCC;
        } else throw new Exception("Error en "
                 +Producto::getCCCSiguiente(String)");
}
```

**Fragmento de código 5.**

El Fragmento de código 5 recupera el valor de la columna UltimoCCC de la tabla Últimos correspondiente a la sucursal pasada como parámetro. Para ello, construye una sentencia adecuada en lenguaje SQL en la que deja como parámetro el valor de la sucursal. A continuación, el Broker (clase que centraliza el acceso de la capa de dominio hacia la base de datos, descrito en la página 5) entrega un objeto de clase

OleDbCommand[3] (al que llamamos *cmd*), que cargamos con la instrucción SQL. Puesto que hemos dejado la instrucción pendiente de la asignación de un parámetro (representado con el carácter "?"), le damos el valor al parámetro en la línea `cmd.Parameters.Add("[sucursal]", OleDbType.VarChar).Value=sucursal` y ejecutamos la sentencia (usando el método ExecuteReader sobre el objeto *cmd*). Puesto que esta SQL devuelve un conjunto de datos (ya que se trata de un *Select*), debemos recuperar el resultado en un tipo de objeto que nos permita su manipulación. Por ello, asignamos el resultado de ExecuteReader al objeto *r*, de clase OleDbDataReader.

A continuación, preguntamos si *r* tiene datos (con *r.Read()*[4], que devuelve un bool) y, en caso afirmativo, procesamos la información leída. En caso contrario, devolvemos una excepción.

El Fragmento de código 6 se encarga de insertar la instancia en la base de datos: construye la instrucción SQL (usando en esta ocasión cuatro parámetros), obtiene un OleDbCommand a partir del Broker, lo carga con la instrucción, asigna valores a los cuatro parámetros y la ejecuta. Puesto que esta instrucción no devuelve ningún conjunto de registros, usamos en esta ocasión el método ExecuteNonQuery() del OleDbCommand. Por último, cerramos la conexión con la base de datos.

```
public virtual void insert()
{
        String SQL="Insert into Productos " +
          "(Sucursal, CCC, Activo, FechaDeCreacion) values (?, ?, ?, ?)";
        OleDbCommand cmd = Broker.getCommand(SQL);
        cmd.Parameters.Add("", OleDbType.VarChar).Value=this.mSucursal;
        cmd.Parameters.Add("", OleDbType.VarChar).Value=this.mCCC;
        cmd.Parameters.Add("", OleDbType.Boolean).Value=this.mActivo;
        cmd.Parameters.Add("", OleDbType.Date).Value=this.mFechaDeCreacion;
        cmd.ExecuteNonQuery();
        cmd.Connection.Close();
}
```

**Fragmento de código 6.**

Junto a las citadas, la clase Producto tiene las operaciones (métodos y propiedades) que se muestran en la Figura 4. Las operaciones que modifican el estado de la instancia

[3] Los objetos OleDbCommand representan tanto sentencias que se ejecutan contra bases de datos como procedimientos almacenados en la base de datos.

[4] En este caso, y puesto que la SQL devolverá a lo sumo un registro (cosa que sabemos porque la columna Sucursal forma la clave principal en la tabla Productos), preguntamos con un *if*. Si la SQL devolviera más de un registro, podríamos recorrerla con un *while*, un *for* y otro bucle.

(como los métodos *bloquear*, *desbloquear*, *cancelar* o *add(Cliente)*) implican la ejecución de alguna operación sobre la base de datos, de manera que el estado de la instancia se encuentre sincronizado con el correspondiente registro en la base de datos. El Fragmento de código 7 muestra cómo, tras cancelar un producto, se ejecuta el método *update* para reflejar el nuevo estado de la instancia en la base de datos.

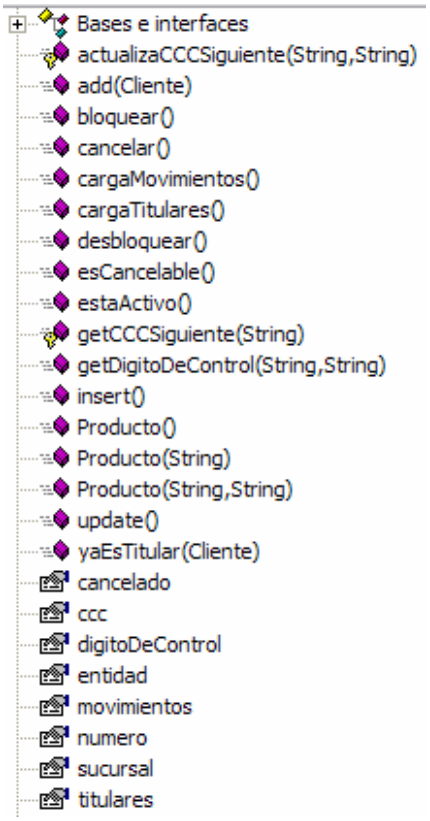



**Figura 4. Métodos y propiedades de Producto**

```
public void cancelar()
{
   this.mFechaDeCancelacion=
                  System.DateTime.Now;
   mActivo=false;
   update();
}
```

**Fragmento de código 7.**

Producto incluye también la operación abstracta *esCancelable():bool*, cuya implementación depende del Producto que se desee cancelar (Fragmento de código 8): una tarjeta puede ser cancelada en cualquier momento, pero una cuenta sólo puede ser cancelada si tiene saldo positivo.

```
public abstract bool
        esCancelable();
```

```
public override bool
          esCancelable()
{
   if (saldo<0 || !mActivo)
             return false;
   return true;
}
```

```
public override bool
          esCancelable()
{
    return true;
}
```

**Fragmento de código 8. La operación *esCancelable* en Producto (izquierda), Cuenta (centro) y Tarjeta (derecha).**

#### 1.1.3.2 Cuenta

La clase Cuenta es una subclase concreta de Producto que representa una cuenta corriente que no aporta campos nuevos a los que ya hereda, pero que posee las operaciones que se muestran en la Figura 5.

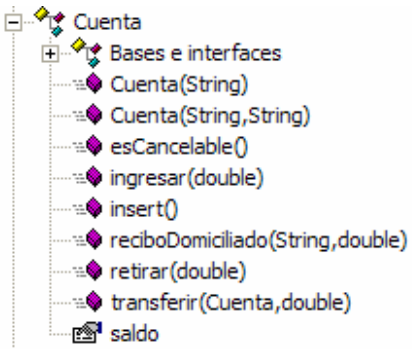



**Figura 5. Operaciones de Cuenta**

El primer constructor (Fragmento de código 9) pasa la llamada al constructor de Producto que toma el mismo parámetro (sucursal, de tipo String), y no realiza ninguna operación adicional. El segundo (Fragmento de código 10) llama al constructor de Producto que toma los dos mismos parámetros (dos Strings), pero luego realiza la operación adicional *cargaMovimientos.*

```
public Cuenta(String sucursal) :
                        base(sucursal)
{
}
```

**Fragmento de código 9**

```
public Cuenta(String s, String ccc) : base(s, ccc)
{
    this.cargaMovimientos();
}
```

**Fragmento de código 10**

Igual que Producto, posee un método *insert* que se utiliza para insertar cuentas en la base de datos: para insertar una cuenta, es preciso insertar en la tabla Cuentas; pero, como hay una relación de integridad referencial desde Cuenta hacia Producto (véase Figura 2), hay que insertar previamente en la tabla Producto. El Fragmento de código 11 muestra este hecho: lo primero que se hace es llamar al método *insert* de la clase base (es decir, de Producto) y, a continuación, se inserta en la tabla Cuentas.

```
public override void insert()
{
    base.insert();
    String SQL="Insert into Cuentas " +
        "(Sucursal, CCC) values (?, ?)";
    OleDbCommand cmd = Broker.getCommand(SQL);
    cmd.Parameters.Add("", OleDbType.VarChar).Value=this.mSucursal;
    cmd.Parameters.Add("", OleDbType.VarChar).Value=this.mCCC;
    cmd.ExecuteNonQuery();
    cmd.Connection.Close();
}
```

**Fragmento de código 11. El método insert en Cuenta**

Las operaciones *ingresar*, *retirar*, *transferir* y *reciboDomiciliado* suponen la realización de movimientos sobre la cuenta. Al ingresar una cantidad en una cuenta, se crea una instancia de Movimiento con un importe positivo y cuyo concepto puede ser "ingreso de efectivo"; al retirar, un Movimiento con un importe negativo cuyo concepto puede ser "retirada de efectivo"; al transferir, se crea un movimiento con importe positivo en la cuenta de destino y uno con importe negativo en la de origen. El Fragmento de código 12 muestra el código de la operación *transferir*, que toma como parámetros la cuenta de destino y el importe que se desea transferir (la cuenta origen es aquella sobre la que se ejecuta el método): inicialmente se realizan algunas comprobaciones (que el importe sea positivo, que haya dinero suficiente en la cuenta, que la cuenta no esté cancelada…) y, a continuación, se lleva a cabo la operación, creándose dos instancias de la clase Movimiento (una para cada cuenta), añadiéndolas a la lista de movimientos de las cuentas correspondientes e insertándolos en la base de datos.

```
public void transferir(Cuenta destino, double importe)
{
    if (importe<=0)
      throw new Exception("Debe transferir un importe positivo");
    if (importe>saldo)
      throw new Exception("No dispone de suficiente saldo para realizar " +
             "la operación");
    if (this.cancelado || !this.estaActivo())
      throw new Exception("La cuenta ordenante está bloqueada o cancelada");
    if (destino.cancelado || !destino.estaActivo())
      throw new Exception("La cuenta destino está bloqueada o cancelada");

    Movimiento m=new Movimiento("Orden de transferencia",
                          System.DateTime.Now, -importe, this);
    this.mMovimientos.Add(m);
    m.insert();
    Movimiento m2=new Movimiento("Transferencia a su favor",
                                 DateTime.Now, importe, destino);
    destino.mMovimientos.Add(m2);
    m2.insert();
}
```

**Fragmento de código 12**

Cuenta incluye la propiedad *saldo*, que devuelve el saldo de la cuenta. Se calcula sumando los importes de todos los movimientos de la cuenta (Fragmento de código 13).

```
public double saldo
{
     get {
        double result=0;
        for (int i=0; i<this.mMovimientos.Count; i++)
        {
             Movimiento m=(Movimiento) this.mMovimientos[i];
             result+=m.importe;
        }
        return result;
     }
}
```

**Fragmento de código 13. La propiedad *saldo* de la clase Cuenta, que es de sólo lectura (no incluye la parte *set*)**

#### 1.1.3.3 Tarjeta, Débito y Crédito

Tarjeta es una clase abstracta que es a su vez subclase de Producto. Tarjeta sirve como base para crear los dos tipos de tarjetas que maneja el banco: Débito y Crédito. De sus operaciones (Figura 6), resultan interesantes *sacarDinero* y *pagarEnComercio*. Ambas son abstractas, ya que el comportamiento de estas operaciones es distinto, según el tipo de tarjeta con que se saque dinero o se pague en un comercio.

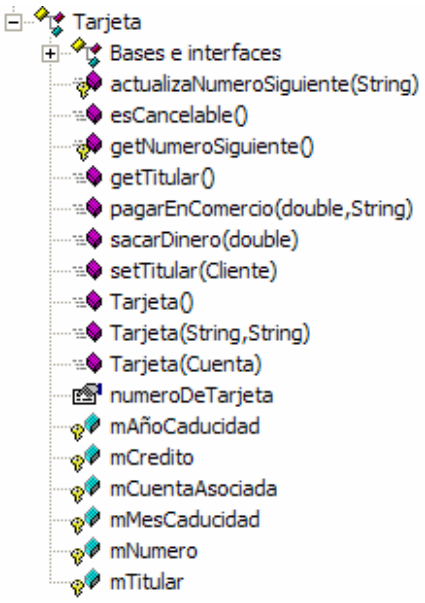



**Figura 6. Operaciones y campos de Tarjeta**

En las tarjetas de débito, la operación se transmite directamente a la cuenta asociada a la tarjeta, de modo que el cargo se realiza de manera inmediata (obsérvese, en el

Fragmento de código 14, que lo que se hace es redirigir un mensaje a la cuenta asociada a la tarjeta, representado por el campo mCuentaAsociada, de tipo Cuenta). La Cuenta se encarga de realizar las comprobaciones correspondientes de existencia de saldo, importe positivo, etc.

```
public override void sacarDinero(double importe)
{
      this.mCuentaAsociada.retirar(importe);
}

public override void pagarEnComercio(double importe, String comercio)
{
      this.mCuentaAsociada.reciboDomiciliado(comercio, importe);
}
```

**Fragmento de código 14. Las operaciones de Tarjeta redefinidas en Débito**

En las de tarjetas de crédito, sin embargo, se añade el movimiento a la lista de movimientos de la tarjeta para su posterior procesamiento. En este caso, la propia tarjeta debe comprobar que se disponga de crédito suficiente. Esto se ilustra en el Fragmento de código 15, en donde se utiliza la operación auxiliar *creditoDisponible():double*.

```
public override void sacarDinero(double importe)
{
   if (creditoDisponible()<importe*1.03)
      throw new Exception("No dispone de suficiente crédito " +
            "para retirar " + importe);
   Movimiento m=new Movimiento("Retirada de efectivo",
      DateTime.Now, importe, this);
   this.movimientos.Add(m);
   m.insert();
   Movimiento m2=new Movimiento("Comisión por retirada de efectivo",
      DateTime.Now, importe*0.03, this);
   this.movimientos.Add(m2);
   m2.insert();
}

public override void pagarEnComercio(double importe, String comercio)
{
   if (creditoDisponible()<importe)
      throw new Exception("No dispone de suficiente crédito para " +
         "retirar " + importe);
   Movimiento m=new Movimiento("Retirada de efectivo",
      DateTime.Now, importe, this);
   this.movimientos.Add(m);
   m.insert();
}

public double creditoDisponible()
{
   String SQL="Select sum(importe) from Movimientos where " +
      "Sucursal=? and CCC=? and Month(Fecha)=?";
   OleDbCommand cmd = Broker.getCommand(SQL);
   cmd.Parameters.Add("", OleDbType.VarChar).Value=this.mSucursal;
   cmd.Parameters.Add("", OleDbType.VarChar).Value=this.mCCC;
   cmd.Parameters.Add("", OleDbType.Integer).Value=DateTime.Now.Month;
```

```
        OleDbDataReader r=cmd.ExecuteReader();
        double result=0;
        if (r.Read())
        {
            result=r.GetDouble(0);
            cmd.Connection.Close();
        } else throw new Exception("Error en Credito::creditoDisponible");
        return result;
    }
```

**Fragmento de código 15. Las operaciones de Tarjeta redefinidas en Crédito, más una operación adicional**

#### 1.1.3.4 Movimiento

La clase Movimiento (Figura 7) representa un movimiento que se realiza sobre algún producto. Cada Movimiento tiene un importe, una fecha, un concepto y un identificador único (campo mId), así como una referencia a su producto correspondiente (campo mProducto:Producto, que en la tabla Productos se corresponde con las columnas Sucursal y CCC, que apuntan a la clave principal de Productos, como se mostraba en la Figura 2, página 5).

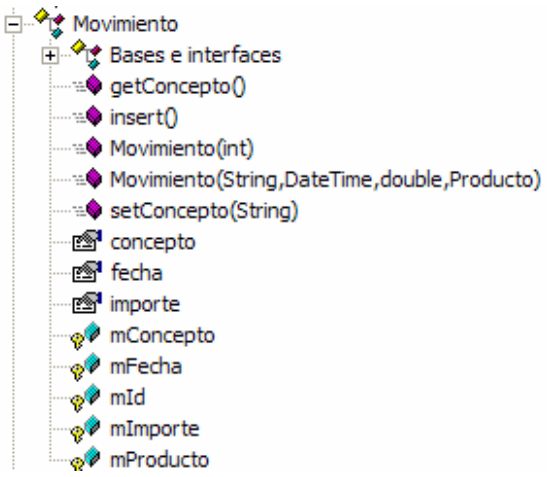



**Figura 7. Operaciones y campos de Movimiento**

El constructor *Movimiento(String concepto, DateTime fecha, double importe, Producto p)* crea una nueva instancia de Movimiento y asigna a los campos los parámetros correspondientes. El constructor *Movimiento(int)* se utiliza para materializar un movimiento (es decir, para construir una instancia de Movimiento a partir de un registro contenido en la tabla Movimientos).

Cuando se inserta un movimiento en la tabla hay que dar valor a todas las columnas. Los valores correspondientes a las columnas Sucursal y CCC se recuperan del campo mProducto, como se muestra en el Fragmento de código 16.

```
public void insert()
{
    String SQL="Insert into Movimientos " +
       "(Sucursal, CCC, Importe, Fecha, Concepto) values (?, ?, ?, ?, ?)";
    OleDbCommand cmd = Broker.getCommand(SQL);
    cmd.Parameters.Add("", OleDbType.VarChar).Value=mProducto.sucursal;
    cmd.Parameters.Add("", OleDbType.VarChar).Value=mProducto.ccc;
    cmd.Parameters.Add("", OleDbType.Double).Value=this.mImporte;
    cmd.Parameters.Add("", OleDbType.Date).Value=mFecha;
    cmd.Parameters.Add("", OleDbType.VarChar).Value=mConcepto;
    cmd.ExecuteNonQuery();
    cmd.Connection.Close();
    }
```

**Fragmento de código 16. El método insert de Movimiento**

#### 1.1.3.5 Contrato

La clase Contrato (Figura 8) representa el hecho de que un Cliente es titular de un Producto, razón por la que posee dos campos de estos tipos (mCliente y mProducto, respectivamente). Esta clase representa la asociación de muchos a muchos entre Cliente y Producto que mostrábamos en la Figura 3 (página 6).

Su único constructor asigna a sus dos únicos campos los valores de los parámetros que recibe.

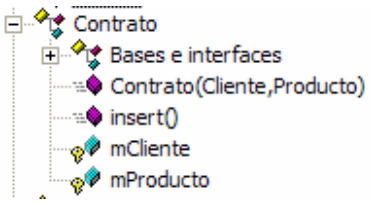



**Figura 8. La clase Contrato**

Las instancias de Contrato se insertan en la base de datos cuando se añade un Cliente a la lista de titulares de un Producto. Previamente, debe comprobarse que el Cliente no sea ya titular de dicho Producto. Todo esto se realizaba en el método *add(Cliente)* de la clase Producto (Fragmento de código 17, que muestra tanto la operación *add* como la operación auxiliar *yaEsTitular*, utilizada por *add*).

```
public void add(Cliente c)
{
    if (yaEsTitular(c))
        throw new Exception("El cliente con NIF " + c.nif +
                            " ya es titular de este producto");
    else
    {
        Contrato con=new Contrato(c, this);
        con.insert();
    }
}
public bool yaEsTitular(Cliente c)
{
  String SQL="Select count(*) from Contratos where NIF=? and " +
        "Sucursal=? and CCC=?";
  OleDbCommand cmd=Broker.getCommand(SQL);
  cmd.Parameters.Add("", OleDbType.VarChar).Value=c.nif;
  cmd.Parameters.Add("", OleDbType.VarChar).Value=this.mSucursal;
  cmd.Parameters.Add("", OleDbType.VarChar).Value=this.mCCC;
  OleDbDataReader r=cmd.ExecuteReader();
  int result=0;
  if (r.Read())
  {
        result=r.GetInt32(0);
  }
  cmd.Connection.Close();
  return result==1;
}
```

**Fragmento de código 17. Las operaciones *add(Cliente)* y *yaEsTitular(Cliente)*, definidas en la clase Producto.**

#### 1.1.3.6 GestorDeListas

GestorDeListas (Figura 9) es una clase auxiliar, no de dominio, que se utiliza para recuperar conjuntos de registros de la base de datos.


**Figura 9. La clase GestorDeListas**

Su método *getVista* (Fragmento de código 18) ejecuta la instrucción SQL pasada como parámetro sobre la base de datos, recupera los registros y los devuelve, pero transformados en un ArrayList. Esta clase es utilizada únicamente por las ventanas que se encargan de mostrar listados. Cada elemento del ArrayList devuelto es un array de objetos tipo *object* (es decir, de cualquier cosa). El primer elemento del ArrayList está

formado por los nombres de las columnas leídas; cada uno de los restantes elementos del ArrayList contiene los datos correspondientes a un registro.

```
public static ArrayList getVista(String SQL) {
      OleDbCommand cmd=Broker.getCommand(SQL);
      OleDbDataReader r=cmd.ExecuteReader();
      int columnas=r.FieldCount;
      ArrayList result=new ArrayList();
      object[] titulos=new object[columnas];
      for (int i=0; i<columnas; i++)
      {
            titulos[i]=r.GetName(i);
      }
      result.Add(titulos);
      while (r.Read())
      {
            object[] fila=new object[columnas];
            for (int i=0; i<columnas; i++)
                  fila[i]=r.GetValue(i);
            result.Add(fila);
      }
      cmd.Connection.Close();
      return result;
}
```

**Fragmento de código 18. El método *getVista* de GestorDeListas**

Así, por ejemplo, suponiendo que los datos recuperados por la instrucción SQL son los que se muestran en la Figura 10, el ArrayList devuelto en el objeto *result* tendría estos datos:

Elemento 0: {“Sucursal”, “CCC”, “Activo”, “FechaDeCreacion”, “FechaDeCancelacion”}

Elemento 1: {“1000”, “000 000000012”, 1, #02/03/2005 2:15:0#, #0:00:00#}

…

| Sucursal | CCC | Activo | FechaDeCreacion | FechaDeCancelacion |
|---|---|---|---|---|
| 1000 | 0000000012 | 1 | 02/03/2005 2:15:0 | 0:00:00 |
| 1000 | 0000000013 | 1 | 02/03/2005 23:54: | 0:00:00 |
| 1000 | 0000000014 | 1 | 02/03/2005 23:56: | 0:00:00 |
| 1000 | 0000000017 | 1 | 03/03/2005 16:40: | 0:00:00 |
| 1000 | 0000000019 | 1 | 03/03/2005 17:13: | <NULL> |
| 1000 | 0000000020 | 1 | 03/03/2005 17:30: | <NULL> |

**Figura 10**

El número de columnas recuperadas se conoce gracias al método FieldCount que se ejecuta sobre el objeto *r*, de clase OleDbDataReader.

Por último, *getTabla* toma como parámetro el nombre de una tabla o vista, construye una instrucción SQL y llama a *getVista:*

```
public static ArrayList getTabla(String tabla)
{
      String SQL="Select * from [" + tabla + "]";
      return getVista(SQL);
}
```

**Fragmento de código 19**

### 1.1.4 Descripción de las clases incluidas en *presentación*

La Figura 11 muestra el conjunto de clases incluida en la capa de presentación.

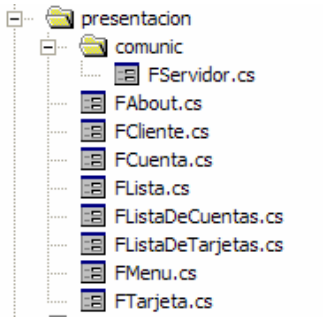



**Figura 11. Clases incluidas en *presentación***

#### 1.1.4.1 FMenu y FLista

La aplicación arranca por FMenu, una ventana que muestra el menú principal. FMenu es una ventana "contenedora", lo que significa que, en su interior, puede albergar otras ventanas, como se muestra en la Figura 12.

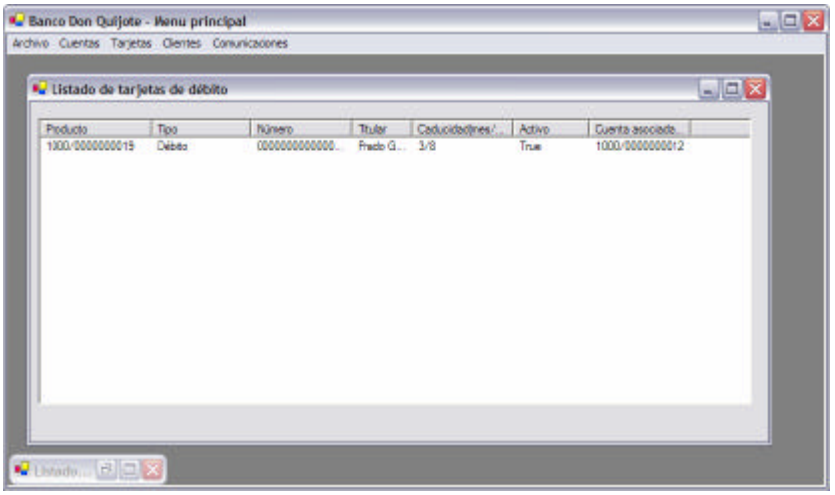



**Figura 12. Ventana FMenu**

La ventana etiquetada “Listado de tarjetas de débito” que aparece en la Figura 12 es una instancia de la clase FListaDeTarjetas, que es a su vez una subclase de FLista. La Figura 13 muestra el aspecto de FLista en tiempo de diseño. El gran control de color blanco que ocupa casi todo el formulario es un ListView, que permite mostrar listados, iconos, etc. Queremos que FLista sirva como plantilla para crear ventanas que muestren listados de tarjetas, cuentas, clientes, etc., pero no existirán instancias de FLista propiamente dichas, sino algunas de sus especializaciones.

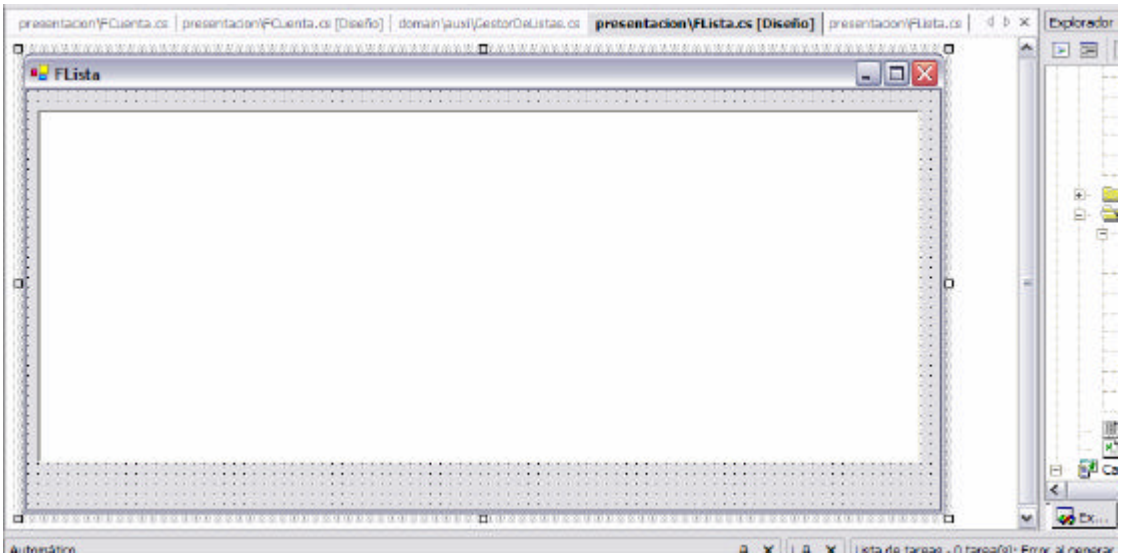



**Figura 13. La ventana FLista, en tiempo de diseño**

Para ello es preciso construir formularios que hereden de FLista. Los formularios heredados pueden crearse usando el asistente del entorno .NET (Figura 14).

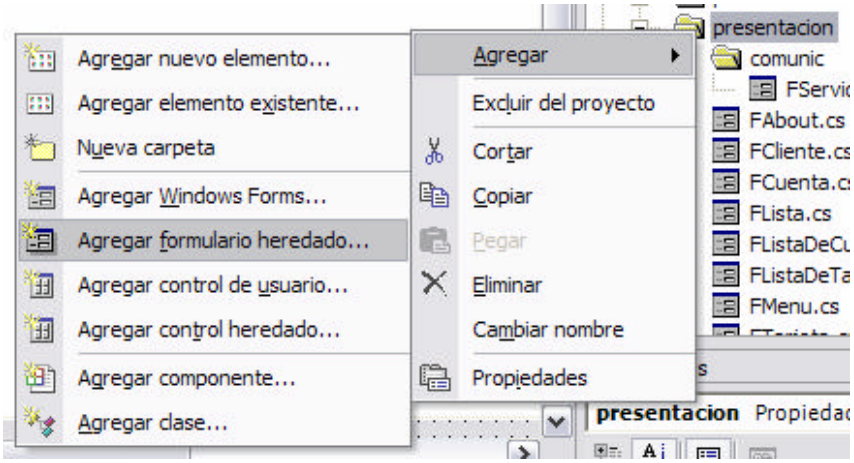



**Figura 14**

En nuestra aplicación, FLista se ha declarado como abstracta. Su constructor (Fragmento de código 20) recibe como parámetro una cadena (que puede corresponderse con el texto que aparecerá en la barra de título de la ventana) y un ArrayList con los datos que se desean listar. Este ArrayList se habrá construido utilizando la clase GestorDeListas que mencionábamos en la sección 1.1.3.6. El constructor establece el aspecto de la ventana (método *InitializeComponent*, generado automáticamente por el entorno de desarrollo), pone los títulos en el ListView que mostrará los datos (objeto *listView1*) y llama al método *cargaDatos()*. Ésta es una operación abstracta, que se redefinirá en las especializaciones de esta clase (el Fragmento de código 21, por ejemplo, muestra la implementación de *cargaDatos* que se hace en FListaDeCuentas).

```
public FLista(String tabla, ArrayList datos)
{
    //
    // Necesario para admitir el Diseñador de Windows Forms
    //
    InitializeComponent();

    this.listView1.Columns.Add("", 0, HorizontalAlignment.Left);
    object[] titulos=(object[]) datos[0];
    for (int i=0; i<titulos.Length; i++)
        this.listView1.Columns.Add(titulos[i].ToString(),
                listView1.Width/titulos.Length, HorizontalAlignment.Left);
    listView1.View = View.Details;
    mTabla=tabla;
    mDatos=datos;
    cargaDatos();
}

protected abstract void cargaDatos();
```

**Fragmento de código 20. Constructor y operación abstracta en FLista**

```
protected override void cargaDatos()
{
    this.listView1.Columns[1].Width=100;
    this.listView1.Columns[2].Width=100;
    for (int i=1; i<this.mDatos.Count; i++)
    {
        object[] filaDeDatos=(object[]) mDatos[i];
        ListViewItem fila = new ListViewItem();
        for (int j=0; j<filaDeDatos.Length; j++)
               fila.SubItems.Add(filaDeDatos[j].ToString());
        listView1.Items.Add(fila);
    }
    }
```

**Fragmento de código 21. El método *cargaDatos* en FListaDeCuentas**

#### 1.1.4.2 FCuenta

FCuenta (Figura 15) permite a los usuarios gestionar cuentas. Esta ventana es un mero transmisor de mensajes del usuario hacia un objeto mCuenta de clase Cuenta, que representa la cuenta sobre la que se está trabajando. mCuenta es, por tanto, un campo de la clase FCuenta.

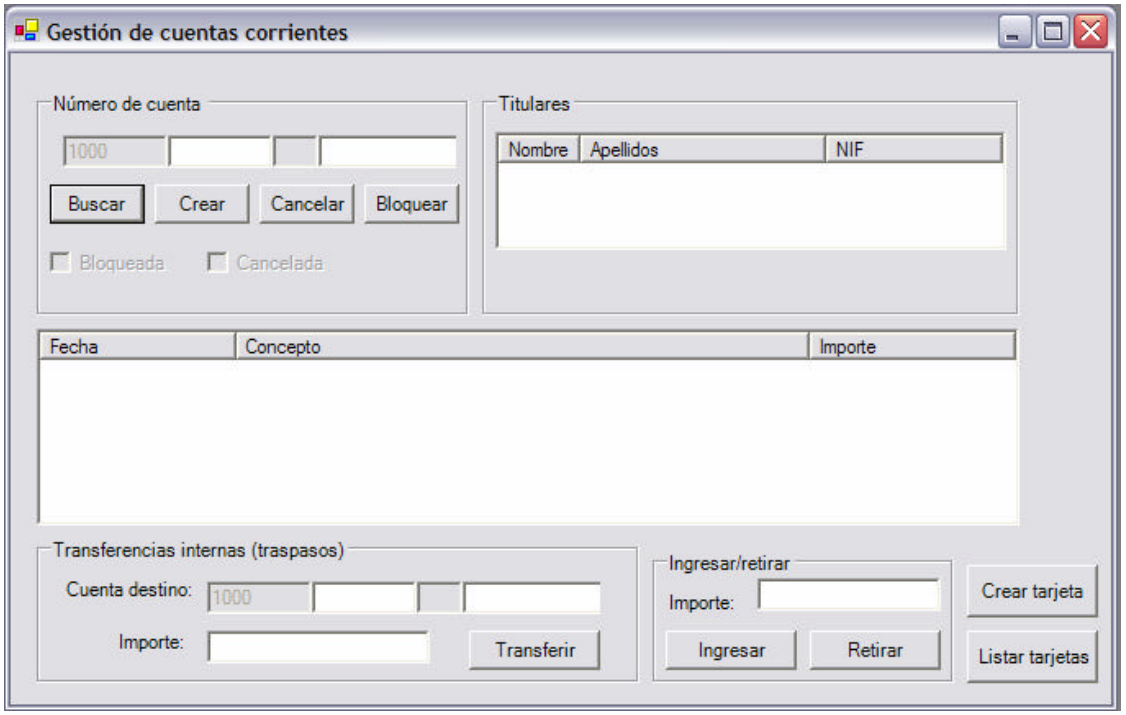



**Figura 15. Ventana FCuenta**

La siguiente tabla muestra las principales operaciones que se ejecutan dependiendo de cómo interactúe el usuario con la ventana.

| Cuando el usuario… | …se ejecuta |
|---|---|
| pulsa el botón Buscar | `mCuenta=new Cuenta(this.cuentaOrigen.sucursal, this.cuentaOrigen.ccc);` (véase Fragmento de código 9, página 13) |
| pulsa el botón Crear | `mCuenta=new Cuenta(this.cuentaOrigen.sucursal);` (véase Fragmento de código 10, página 13) |
| pulsa el botón Ingresar | `this.mCuenta.ingresar(double.Parse(this.tbImporteIR.Text));` |
| pulsa el botón Retirar | `this.mCuenta.retirar(double.Parse(this.tbImporteIR.Text));` |
| pulsa el botón Transferir | `Cuenta destino=new Cuenta(this.cuentaDestino.sucursal, this.cuentaDestino.ccc);` `...` `mCuenta.transferir(destino, double.Parse(this.tbImporte.Text));` (véase Fragmento de código 12, página 14) |
| pulsa el botón Crear tarjeta | `FTarjeta f=new FTarjeta(mCuenta);` `f.MdiParent=this.MdiParent;` `f.Show();` (se crea una ventana de gestión de tarjetas a la que pasamos la cuenta con la estamos trabajando; decimo s a la ventana que estará contenida en la misma ventana en la que FCuenta está contenida –es decir, en FMenu-; mostramos la ventana) |
| pulsa el botón Listar tarjetas | `String SQL="Select * from TarjetasDeTitulares where " + "[Cuenta asociada (sucursal/CCC)]='" + mCuenta.sucursal + "/" + mCuenta.ccc + "'";` `ArrayList datos=GestorDeListas.getVista(SQL);` `FListaDeTarjetas f=new FListaDeTarjetas("Tarjetas asociadas a la cuenta " + mCuenta.numero, datos);` `f.MdiParent=this.MdiParent;` `f.Show();` (se recupera un conjunto de registros de una vista –ojo: no de una tabla- usando la clase GestorDeListas, descrita en la Sección 1.1.3.6, página 19, y se muestra una ventana de tipo FListaDeTarjetas) |

**Tabla 1. Algunas acciones producidas por la ventana**

#### 1.1.4.3 FTarjeta

FTarjeta se utiliza para crear tarjetas, ya sean de débito o de crédito. FTarjeta conoce a un objeto de clase Tarjeta denominada mTarjeta. Al pulsar el botón “Crear tarjeta”, mTarjeta se instancia bien a una tarjeta de crédito o a una de débito, dependiendo de lo que se haya seleccionado en la ventana.

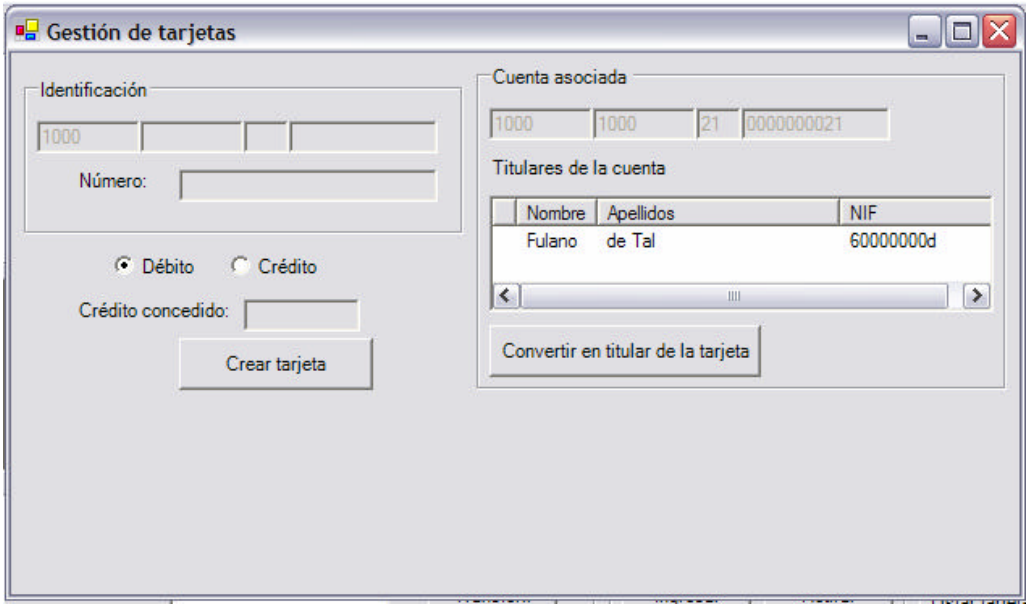



**Figura 16. Ventana FTarjeta**

#### 1.1.4.4 Descripción de *persistencia*

persistencia contiene únicamente la clase Broker, que representa el punto de acceso común a la base de datos. El

```
public class Broker {
   protected OleDbConnection mBD=null;

   public Broker() {
      String cad="Provider=Microsoft.Jet.OLEDB.4.0; Data Source=" +
         "C:\\banco.mdb";
      mBD=new OleDbConnection(cad);
      mBD.Open();
   }

   public static OleDbConnection getBD() {
      Broker broker=new Broker();
      return broker.mBD;
   }

   public static OleDbCommand getCommand(String SQL) {
      OleDbCommand cmd=new OleDbCommand(SQL);
      cmd.Connection=getBD();
      return cmd;
   }

   public void close() {
      mBD.Close();
   }
}
```

**Fragmento de código 22. Código de la clase Broker**

#### 1.1.4.5 Descripción de *comunicaciones*

Aquí se encuentra la clase Servidor, que contiene un socket que recibe peticiones de los cajeros automáticos. Las órdenes son recibidas carácter a carácter en el método OnDataReceived. Cuando la orden ha llegado completamente (el final de la orden está delimitado por una almohadilla), la orden se procesa (Fragmento de código 23).

```
private  void OnDataReceived(IAsyncResult asyn)
{
    try
    {
        CSocketPacket theSockId = (CSocketPacket)asyn.AsyncState ;
        //end receive...
        int iRx  = 0 ;
        iRx = theSockId.thisSocket.EndReceive (asyn);
        char[] chars = new char[iRx +  1];
        System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
        int charLen = d.GetChars(theSockId.dataBuffer, 0, iRx, chars, 0);
        System.String orden = new System.String(chars);
        if (orden[0]=='#')
        {
               this.mFinDeOrden=true;
               procesaOrden();
        }
        else
               mOrden+=orden[0];
        if (mClienteConectado)
               WaitForData(mCliente);
    }
    catch(Exception se)
    {
        mVentana.logError(se.Message);
    }
}

protected void procesaOrden()
{
    if (mOrden.StartsWith("SACAR"))
    {
        String numeroTarjeta=mOrden.Substring(5, 16);
        String sImporte=mOrden.Substring(21);
        double importe=double.Parse(sImporte);
        try
        {
               Tarjeta t=Debito.getTarjeta(numeroTarjeta);
               t.sacarDinero(importe);
               this.Enviar("ACEPTADA#");
        }
        catch (Exception ex)
        {
               this.Enviar("ERROR" + ex.Message);
        }
    }
}
```

**Fragmento de código 23. Recepción de datos y procesamiento de órdenes en el servidor**

El Servidor se pone en marcha pulsando el botón “On” que hay en presentacion/comunic/FServidor.

### *1.2 Proyecto “Cajero”*

El proyecto Cajero consta de tres clases y una interfaz. Aunque este proyecto no está estructurado en capas, dos de las clases se encontrarían en la capa de presentación y una en la de comunicaciones.

En presentación nos encontramos con las clases FCajero y Teclado. Teclado (Figura 17) es realmente un control de usuario que muestra un teclado de cajero automático y un display. Cuando el usuario pulsa alguna tecla, ésta se envía al elemento sobre el que está situado el control, que puede ser “casi cualquier cosa”. Es decir, que el Teclado es un mero intermediario entre el usuario y el cajero automático propiamente dicho.

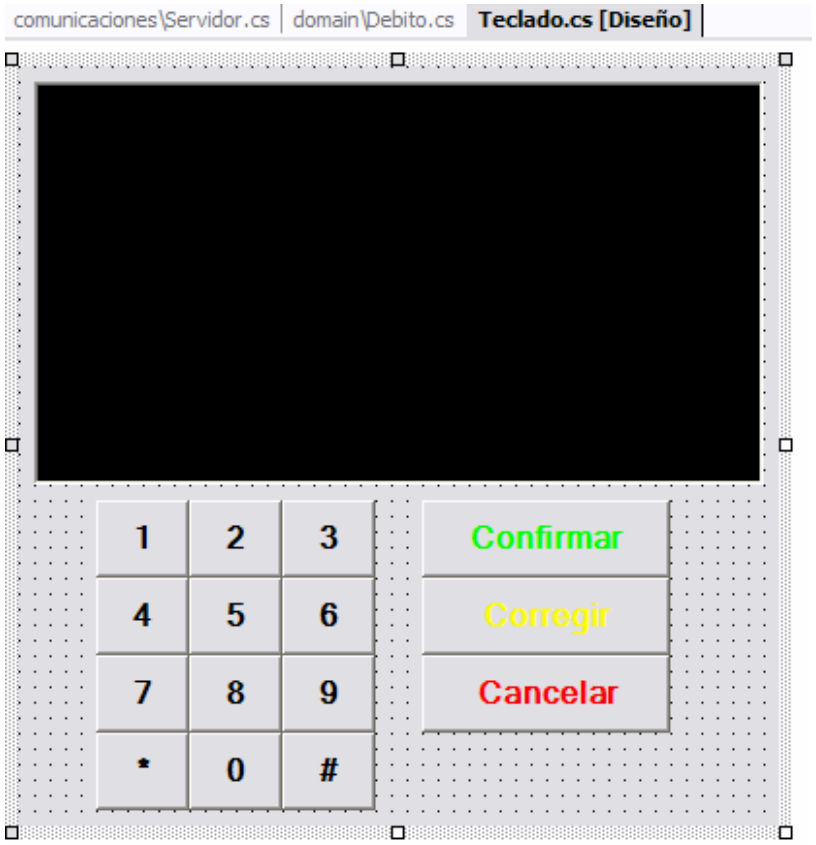



**Figura 17. El control de usuario Teclado**

Como se ha dicho, las pulsaciones se envían al objeto sobre el que está situado el Teclado. Cuando se pulsa una tecla, se ejecuta sobre dicho objeto el método

*recibirPulsacion(String texto)*. El objeto que recibe este mensaje puede ser cualquiera, pero debe implementar *recibirPulsacion*, que es una operación descrita en la interfaz ISoporteDeTeclado. La estructura general del cajero es la siguiente:

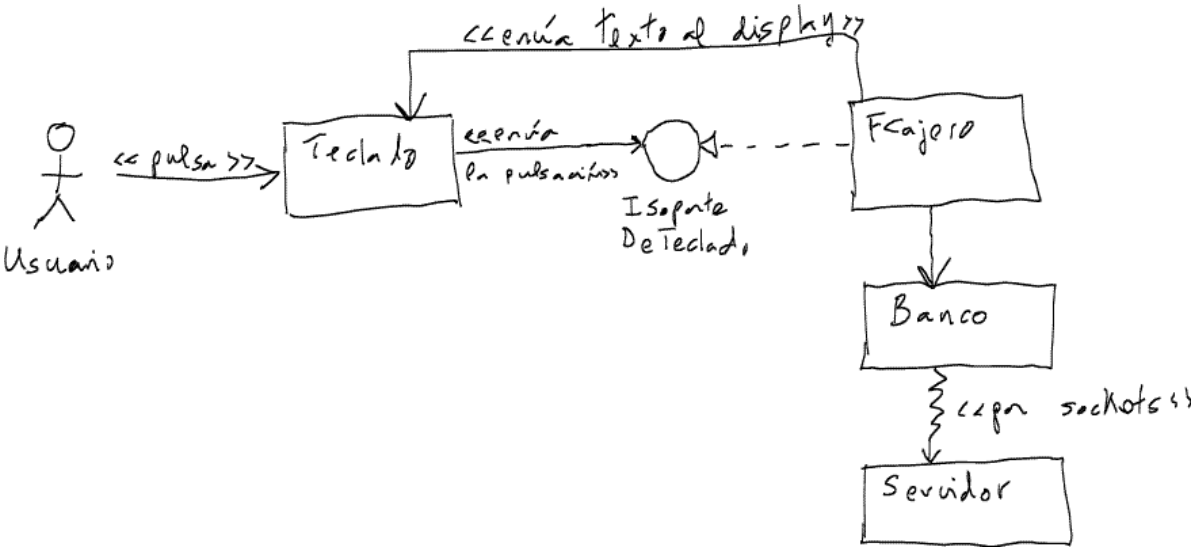



**Figura 18. Estructura general del proyecto Cajero**

Cuando el usuario pulsa una tecla, la instancia de Teclado la envía al objeto sobre el que está situado, que puede ser de cualquier clase que implemente la interfaz ISoporteDeTeclado. En este caso, las pulsaciones las recibe una instancia de FCajero, que procesa la pulsación y, dependiendo del estado en que se encuentre, realiza una u otra operación. En ocasiones, la operación consiste en enviar una orden al Servidor del banco (véase sección 1.1.4.5, página 27), lo cual se realiza mediante la clase Banco usando sockets.

## *1.3 Proyecto ControlNumeroDeCuenta*

Este proyecto comprende únicamente el control de usuario NumeroDeCuenta, que contiene cuatro cajas de texto para manejar de forma cómoda los números de cuenta.

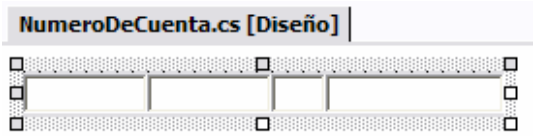



**Figura 19. El control NumeroDeCuenta, en tiempo de diseño**

La primera y la tercera cajas de texto están deshabilitadas (propiedad *enabled* a false) ya que se corresponden con el código de entidad (que en nuestro banco es siempre fijo y vale 1000) y el dígito de control (que se calcula en función de los otros tres campos).

Cuando cambia el texto en las cajas segunda o cuarta, se recalcula el dígito de control, actualizándose el valor mostrado en la tercera caja:

```
private void tbSucursal_TextChanged(object sender, System.EventArgs e)
{
     this.tbDC.Text=getDigitoDeControl(this.tbSucursal.Text,
                                                    this.tbCCC.Text);
}

private void tbCCC_TextChanged(object sender, System.EventArgs e)
{
     this.tbDC.Text=getDigitoDeControl(this.tbSucursal.Text,
                                                    this.tbCCC.Text);
}
```

**Fragmento de código 24. Código que se ejecuta al cambiar las cajas de texto correspondientes a la Sucursal y al CCC**

Se decidió por crear este control ya que se utiliza con frecuencia en el proyecto banco1.

# 2 Funcionalidades adicionales

.NET incluye un conjunto importante de facilidades para manipular datos, mantener comunicaciones con otros sistemas o dispositivos, etc.

## *2.1 Importación y exportación de datos*

Puede ocurrir que el banco necesite enviar los datos de sus clientes a, por ejemplo, el Ministerio de Hacienda. .NET incluye la posibilidad de usar la clase DataSet (y otras clases asociadas) para manipular con comodidad conjuntos de datos.

El siguiente fragmento de código guarda en un fichero denominado Clientes.xml el contenido de la tabla Clientes:

```
public void exportarClientes()
{
   String cad="Provider=Microsoft.Jet.OLEDB.4.0; Data Source=" +
      "C:\\ banco.mdb";

   OleDbConnection bd=new OleDbConnection(cad);

   OleDbCommand cmd=new OleDbCommand("SELECT * from Clientes", bd);
   OleDbDataAdapter adaptador = new OleDbDataAdapter();
```

```
    adaptador.SelectCommand=cmd;
    DataSet listadoClientes= new DataSet();
    adaptador.Fill(listadoClientes, "Clientes");
    listadoClientes.WriteXml("c:\\clientes.xml");
}
```

**Fragmento de código 25. Método para exportar un conjunto de clientes en formato XML**

El aspecto del citado fichero sería el siguiente:

```
<?xml version="1.0" standalone="yes" ?>
- <NewDataSet>
  - <Cliente>
      <NIF>5660000</NIF>
      <Nombre>Lola</Nombre>
      <Apellidos>González Pérez</Apellidos>
    </Cliente>
  - <Cliente>
      <NIF>8899887</NIF>
      <Nombre>Lola</Nombre>
      <Apellidos>González Pérez</Apellidos>
    </Cliente>
  - <Cliente>
      <NIF>6667778</NIF>
      <Nombre>Manuel</Nombre>
      <Apellidos>Rosetti Hidalgo</Apellidos>
    </Cliente>
  - <Cliente>
      <NIF>7438778</NIF>
      <Nombre>Josefa</Nombre>
      <Apellidos>Pi Sánchez</Apellidos>
    </Cliente>
  - <Cliente>
      <NIF>1234567</NIF>
```

**Figura 20. Fichero obtenido al exportar los datos de la tabla Clientes**

Para importar los datos, el Ministerio de Hacienda usaría una operación de este estilo:

```
public void importarClientes()
{
      DataSet listadoClientes=new DataSet();
      listadoClientes.ReadXml("c:\\clientes.xml");
      Console.WriteLine(listadoClientes.Tables.Count);
      Console.WriteLine(listadoClientes.Tables[0].TableName);
      foreach (DataRow r in listadoClientes.Tables[0].Rows)
            Console.WriteLine("NIF: " + r[0]);
      }

}
```

**Fragmento de código 26. Código necesario para importar el fichero con los clientes**

## *2.2 Invocación de servicios web*

Los servicios web permiten la ejecución de operaciones remotas mediante la invocación de mensajes SOAP. Supongamos que, en el sistema de gestión bancario, antes de sacar dinero o de pagar en un comercio con la tarjeta de crédito, es preciso validar la operación preguntando a un ordenador de Visa. Puede que Visa ofrezca esta operación mediante un servicio web. Para utilizarla, bastará con que agreguemos una referencia al servicio web y hagamos uso de ella en nuestro proyecto.

Pulsando el botón secundario del ratón en el árbol del proyecto aparece el menú mostrado en la Figura 21. Elegimos Agregar referencia web y aparece la ventana mostrada en la Figura 22.

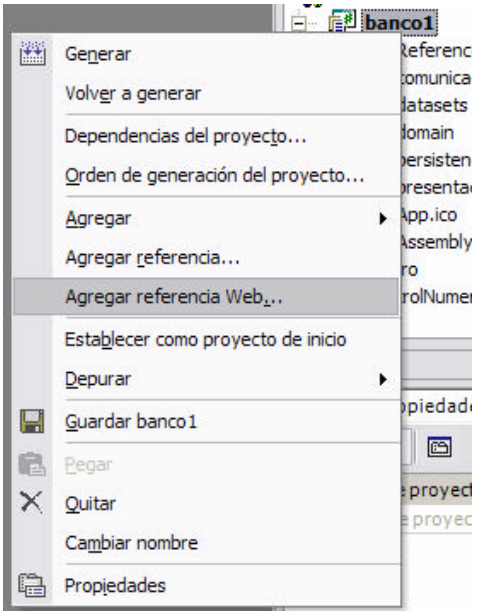



**Figura 21. Adición de una referencia a servicio web**

En la Figura 22 escribimos la URL en la que se encuentra el documento WSDL que describe al servicio web. Pulsamos Ir y, cuando la ventana se carga, se muestran las operaciones ofrecidas por el servicio web. Podemos cambiarle el nombre (.NET ofrece por defecto WebReference) y darle uno más significativo, como Visa. Entonces, pulsamos Agregar referencia.

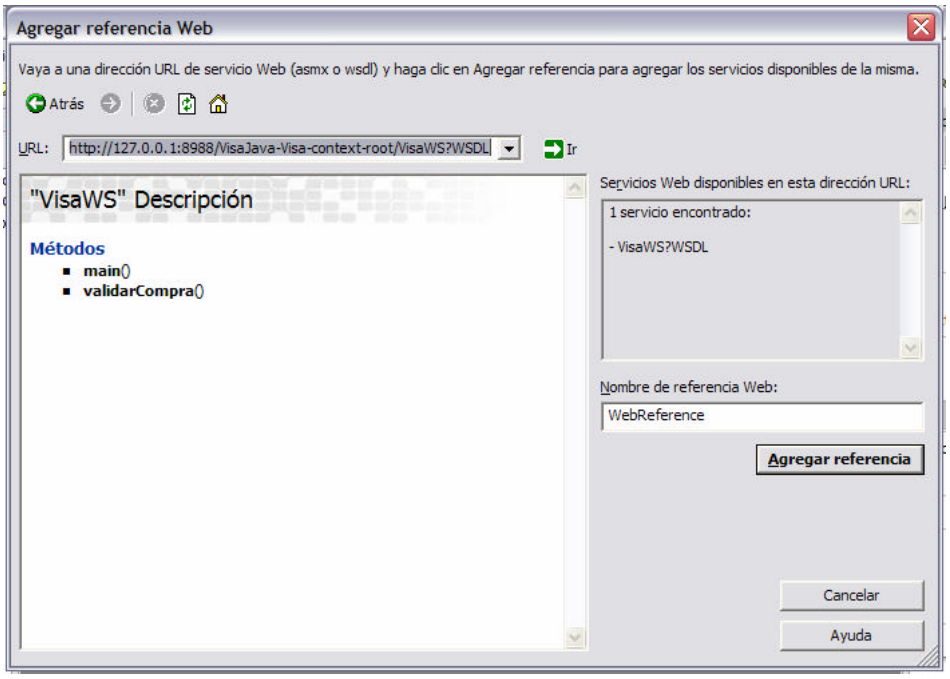



**Figura 22. Ventana para indicar la URL del servicio web**

Acto seguido, el entorno de desarrollo agrega a nuestro proyecto una referencia al servicio web, que podemos utilizar como si se tratase de una clase normal.

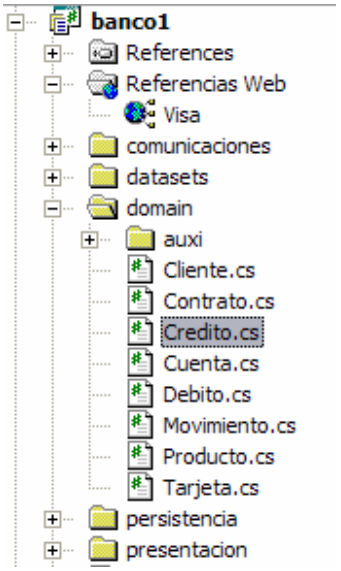



**Figura 23. El entorno ha añadido la referencia al servicio web Visa**

Esto se ilustra en el siguiente fragmento de código, que muestra el método *pagarEnComercio* de la clase Crédito. Obsérvense, en cursiva, las dos líneas que hemos añadido para utilizar la operación *validar* ofrecida por el servicio web.

```
public override void pagarEnComercio(double importe, String comercio)
{
   banco1.Visa.VisaWS visa=new banco1.Visa.VisaWS();

   if (!visa.validarCompra(this.mNumero, importe))
      throw new Exception("Operación no validada por Visa");

   if (creditoDisponible()<importe)
      throw new Exception("No dispone de suficiente crédito " +
                                        "para retirar " + importe);
   Movimiento m=new Movimiento("Compra en " + comercio,
      DateTime.Now, importe, this);
   this.movimientos.Add(m);
   m.insert();
}
```

**Fragmento de código 27. Operación *pagarEnComercio* de Crédito, que usa la operación *validar(String, double):bool* ofrecida por el servicio web Visa**

# 3 Instalación de los proyectos

Los proyectos banco1 y Cajero son aplicaciones que pueden ejecutarse independientemente, por lo que pueden entregarse como ficheros .exe. El proyecto ControlNumeroDeCuenta, sin embargo, no es una aplicación ejecutable, sino un control de usuario que, idealmente, puede reutilizarse en otros proyectos.

Por tanto, generaremos ControlNumeroDeCuenta en forma de biblioteca DLL. Para ello, en las propiedades del proyecto indicamos que el Tipo de resultado es una Biblioteca de clases (Figura 24).

Para los otros dos proyectos se elegirá Aplicación para Windows.

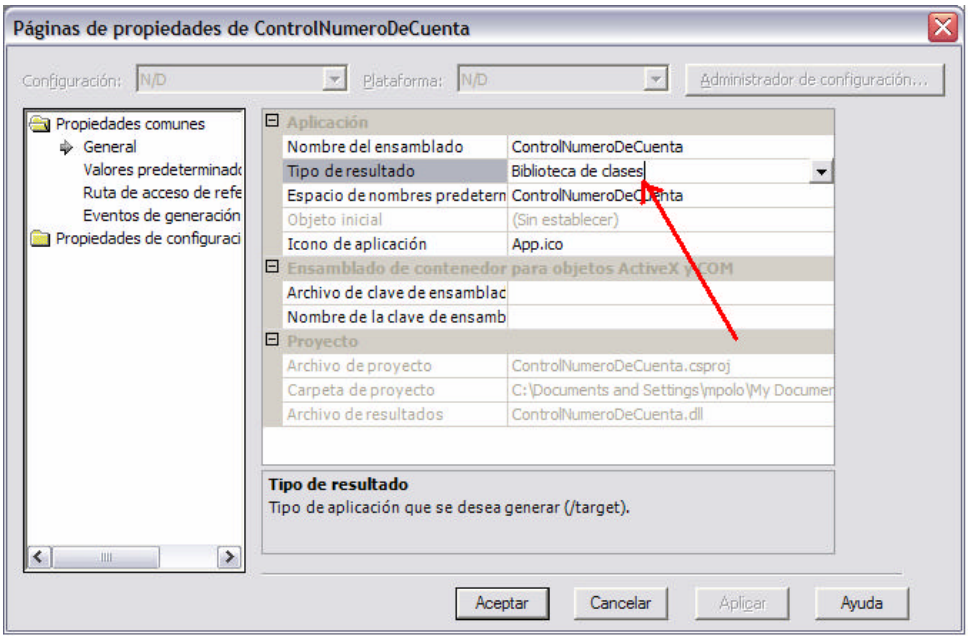



**Figura 24. Ventana de propiedades del proyecto**

Si pulsamos el botón “Administrador de configuración…” que aparece en la figura anterior aparece el siguiente cuadro de diálogo, que sirve para elegir el tipo de configuración de salida (Debug si es depuración, y Release si es una versión del proyecto que puede entregarse al cliente).

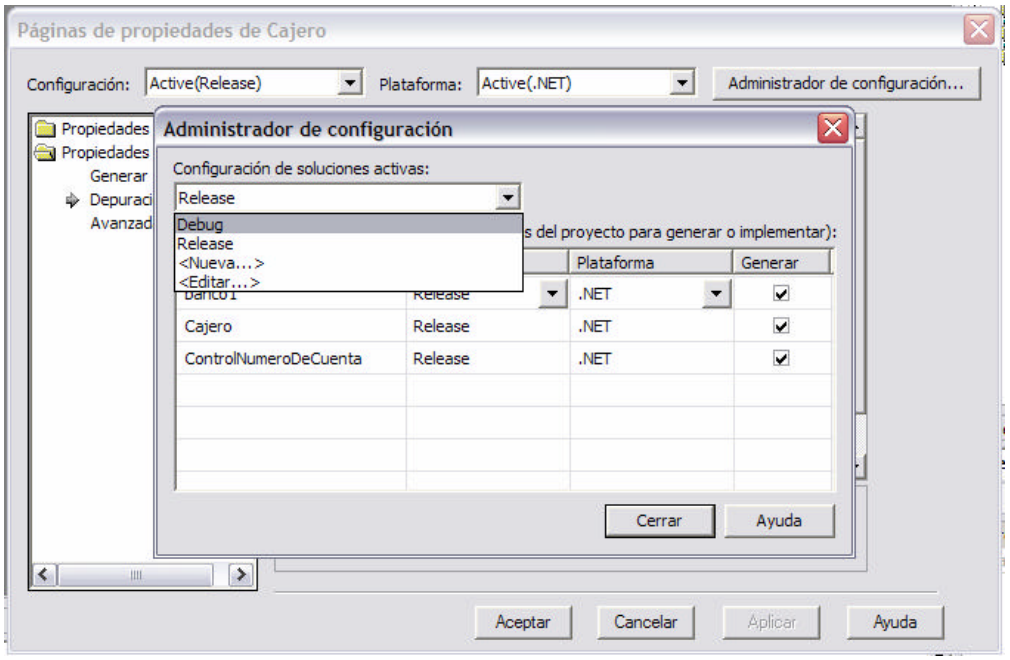



**Figura 25. Elección de la configuración de salida**

# 4 Aplicaciones para Pocket PC

.NET permite la creación de aplicaciones para otros dispositivos, como Pocket PCs. Para ello, debemos elegir “Aplicación para Smart Device” en la ventana del nuevo proyecto (Figura 26). La programación para estos dispositivos es prácticamente igual que la de aplicaciones “clásicas”.

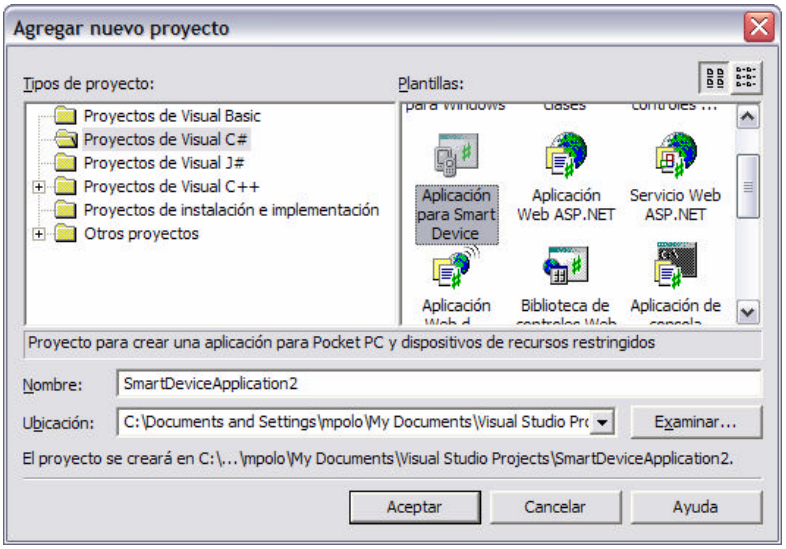



**Figura 26.**

La Figura 27 muestra la estructura de un proyecto para PDA en el que se hace uso del servicio web Visa explicado en la sección 2.2.

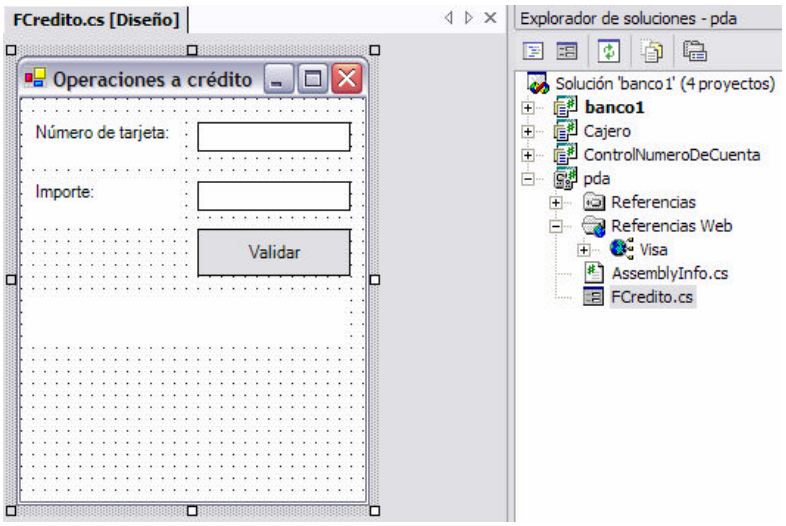



**Figura 27. Un proyecto para PDA**

# Transparencias de C#

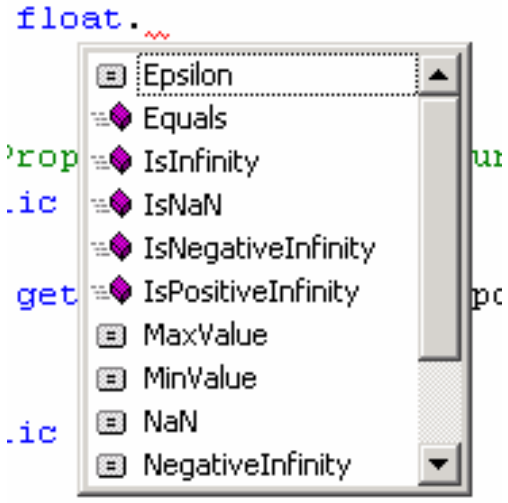
Macario Polo Usaola
Ignacio García Rodríguez de Guzmán

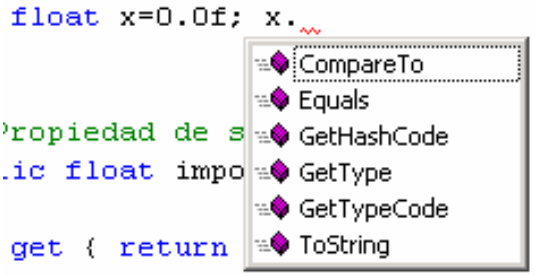
## Características generales

**Todos los tipos son clases**
**Propiedades**
**Nuevos modificadores de visibilidad**
**Diferente tratamiento de la herencia**
**Estructuras**
**Parámetros por valor o por referencia**

**Clases indexadas**
**Aserciones**
**Atributos**
**Reflexión**
**Destructores**
**Enumeraciones**
**Sobrecarga de operadores**
**Delegados**

## Todos los tipos son clases



## Propiedades

```
class Cliente {
        protected String mNIF, mNombre;
            ...
        public String getNIF()
        {
            return this.mNIF;
        }
            ...
        public String Nombre
        {
            get { return this.mNombre; }
            set { mNombre=value;  // "value" es palabra reservada
            }
        }
```

## Nuevos modificadores de visibilidad

***public***
***protected***
***private***
***(en blanco)***

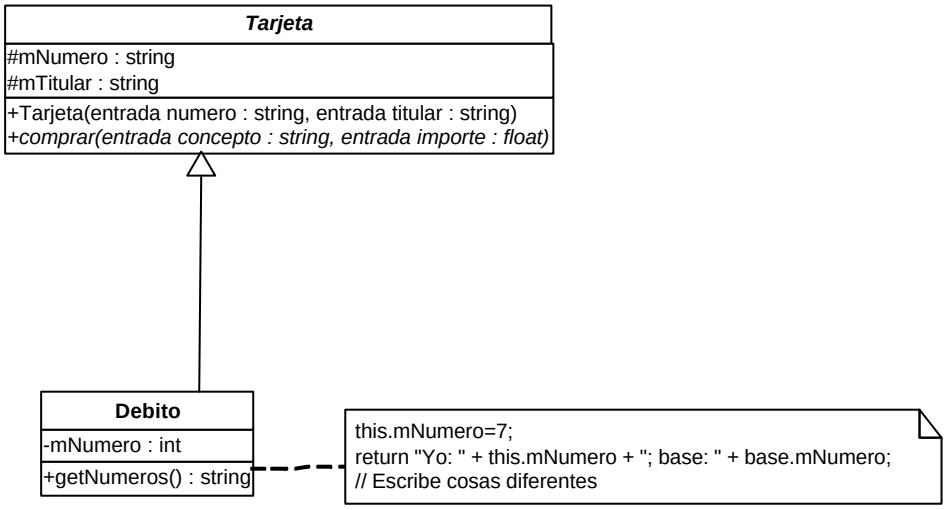
***internal: acceso limitado al proyecto actual***
***protected internal: íd., o a especializaciones***

## Diferente tratamiento de la herencia

**new***: "oculta" un miembro heredado*
**override***: redefine un método o propiedad heredado. Si es un método, debe ser virtual*

**virtual***: denota un miembro que puede ser reemplazado en una especialización*
**abstract***: método abstracto*

## Diferente tratamiento de la herencia



## Estructuras

***Similares a las clases, pero ni heredan de nadie ni se puede heredar de ellas***
***Pasan por valor***
***No pueden tener miembros* protected**
***Sus constructores deben tener parámetros***

## Estructuras

```
public struct NumeroDeCuenta
{
       private const String mE = "1234";
       private String mS, mDC, mCCC;

       public NumeroDeCuenta(String sucursal, String dc, String ccc)
       {
              mS=sucursal;
              mDC=dc;
              mCCC=ccc;
       }

       public String numero
       {
          get { return mE+mS+mDC+mCCC; }
          set {
              if (value.Length!=16)
                 throw new NumeroDeCuentaErroneo(value +
                       " no es un número de cuenta correcto");
              mS=value.Substring(0, 4);
              mDC=value.Substring(4, 2);
              mCCC=value.Substring(6, 10);
          }
       }
}
```

## Parámetros por valor o por referencia

***Pasan por valor las estructuras y los tipos básicos***

***Puede indicarse que uno de estos elementos pase por referencia usando el modificador "ref" en la signatura del método y en la llamada***

## Parámetros por valor o por referencia

```
private void sumar(int x, int y, int r)
{
      r=x+y;
}

private void sumar(int x, int y, ref int r)
{
      r=x+y;
}
Uso: int a=3, b=4, c;  sumar(a, b, ref c);
```

## Parámetros por valor o por referencia

***Pasan por referencia las instancias de clases***

```
private void sumar(Entero x, Entero y, Entero r)
{
      r.setValor(x.getValor()+y.getValor());
}
```

***...siendo* Entero *una clase***

## Parámetros por valor o por referencia

***Pasan por valor las instancias de estructuras***

```
private void sumar(Entero x, Entero y, ref Entero r)
{
    r.setValor(x.getValor()+y.getValor());
}
```

***...siendo* Entero *una estructura. Se usa de este modo: sumar(x, y, ref result)***

## Parámetros por valor o por referencia

***Ojo con los métodos estáticos, en los que todo pasa por valor y es preciso explicitar el paso por referencia***

```
private static void sumar(Entero x, Entero y, Entero r)
{
    r.setValor(x.getValor()+y.getValor());
    // r no cambia al salir del método
}
```

## Clases indexadas

***Permiten referirse a los elementos de una clase como si de un array se tratara***
***La clase tendrá un atributo de tipo colección (array, ArrayList, Hashtable...) que almacenará objetos de cualquier clase***
***Se dota a la clase de una o más propiedades* this**

## Clases indexadas

```
public class ListaDeCuentas
{
	protected Cuenta[] mCuentas;

	public ListaDeCuentas(int size)
	{
		mCuentas=new Cuenta[size];
	}

	public Cuenta this[int pos]
	{
		get { return mCuentas[pos]; }
		set { mCuentas[pos]=value; }
	}
```

## Clases indexadas

***La misma clase puede tener más de un indexador***

```
public Cuenta this[String numero]
{
      get {
            foreach (Cuenta c in this.mCuentas)
                  if (c.numeroDeCuenta==numero)
                        return c;
            return null;
      }
}
```

## Clases indexadas

***Forma de uso:***

```
...
ListaDeCuentas cuentas=null;
for (int i=0; i<10; i++) {
      cuentas[i]=new Cuenta(entidad, sucursal, digito, ccc + i);
}
...
Console.WriteLine("\n\nSaldo de la cuenta " + 1234567890 + ": " +
      cuentas["12341234121234567890"].saldo);
```

## Aserciones

***Comprueban condiciones durante la ejecución, lanzando un mensaje de error si la condición es falsa***
***La condición debe carecer de efectos colaterales***
***Requiere o #define TRACE o #define DEBUG***
***Requiere using System.Diagnostics;***

## Aserciones

```
public void comprar(String concepto, float importe)   {

      Trace.Assert(importe<=this.saldo,

            "No se debe realizar la compra, ya que " +

            "la cuenta sólo dispone de " + saldo + " €");

      this.mMovimientos.Add(new Movimiento(concepto, -importe));

}
```

***Hay tres formatos: Assert(condición), Assert(condición, String) y Assert(condición, String, String)***

***Mucho ojo: la ejecución no se interrumpe***

## Aserciones

***El mensaje de error se envía a Listeners***
***Por defecto, hay un solo listener en Listeners, pero esta colección puede modificarse:***

```
TextWriterTraceListener myWriter = new TextWriterTraceListener();

myWriter.Writer = System.Console.Out;

Trace.Listeners.Add(myWriter);

Trace.Listeners.RemoveAt(0);
```

## Atributos

***Permiten anotar una clase con metadatos (datos acerca de la propia clase)***
***Tales metadatos pueden recuperarse mediante introspección (reflexión)***
***Hay atributos predefinidos, aunque pueden crearse atributos específicos***

## Atributos

```
using System;


namespace banco2.atributos

{

	[AttributeUsage(AttributeTargets.Method)]

	public class Autor : System.Attribute

	{

		protected String mNombre;

		public String mFecha;


		public Autor(String nombre)

		{

			this.mNombre=nombre;

		}

	}

}
```

## Atributos

***Uso del atributo anterior:***

```
		[Autor("Macario Polo Usaola",mFecha="7/11/2004")]
		public void ingresar(float importe)
		{
			...
		}
```

## Atributos predefinidos

### *Conditional*

```
[Conditional("DEBUG")]
public static void Msg(string msg) { Console.WriteLine(msg);
}
```

### *Obsolete*

```
[Obsolete(“Método viejo", true)]
static void antiguo( ) {
     Console.WriteLine(“tal cosa");
}
```

## Atributos predefinidos

### *DllImport*

***[DllImport("User32.Dll", EntryPoint="MessageBox")]***

***static extern int ventanaError(int hWnd, string msg, string caption, int msgType);***

***...podría usarse de esta forma:***

```
public void ingresar(float importe)
{
      if (importe<0)
            ventanaError(0, "No puede ingresar un importe negativo",
                  "Error", 0);
      this.mMovimientos.Add(new Movimiento("Ingreso en efectivo",
            importe));
}
```

## Recuperación de atributos

***Los atributos de un elemento se recuperan así:***

```
Attribute[] atrs= Attribute.GetCustomAttributes(typeof(Cuenta));
```

***GetCustomAttributes toma como parámetro un System.Reflection.Assembly***

## Reflexión

***Existen clases como Type, ConstructorInfo o MethodInfo***

```
Cuenta c=new Cuenta(...);

Type tipoFormaA=c.GetType();

Type tipoFormaB=typeof(Cuenta);
```

***Un* Type *puede recorrerse para ser inspeccionado, ejecutado (si lo admite), etc.***

## Destructores

***El recolector de basura los ejecuta de manera automática***
***Se utilizan para realizar alguna operación al destruir instancias de clases***
***No pueden heredarse, sobrecargarse ni tomar parámetros***
***No tienen modificadores de visibilidad***
***No aplicables a las estructuras***

## Destructores

```
~Cuenta() {

      Console.WriteLine("Destruyendo la cuenta nº " +

            this.numeroDeCuenta);

}
```

## Enumeraciones

***Son colecciones de constantes con nombre***

```
public enum Dias {

      Lunes=1, Martes, Miércoles, Jueves, Viernes, Sábado,  Domingo

}
```

## Enumeraciones

***Se les puede dar un valor inicial (en el ejemplo, se asignaba 1 al Lunes); si no, se asume cero***

***Por defecto, son constantes* int*, pero pueden ser de cualquier tipo integral (salvo* char*)***

```
      public enum Dias : long {

            Lunes=1, Martes, Miércoles, Jueves, Viernes, Sábado,
      Domingo

      }
```

## Sobrecarga de operadores

***Binarios:*** **+, -, *!*, ~, ++, --, *true, false***

***Unarios:*** **+, -, *, /, %, &,** |, ^, <<, >>

***De comparación:*** **==, *!*=,** <, >, <=, >=

## Sobrecarga de operadores

***El siguiente código sobrecarga el + para añadir titulares a una cuenta*:**

```
public static Cuenta operator + (Cuenta c, Cliente titular)
{
      c.mTitulares.Add(titular);
      return c;
}
```

## Sobrecarga de operadores

```
public static Cuenta operator + (Cuenta c, Cliente titular)
{
      c.mTitulares.Add(titular);
      return c;
}

Cliente c=new Cliente(this.tbNIF.Text, this.tbNombre.Text);
mCuenta=mCuenta+c;
```

## Sobrecarga de operadores

```
            public static bool operator true(Cuenta c) {
                  return c.saldo > 0;
            }

            public static bool operator false(Cuenta c) {
                  return c.saldo <= 0;
            }
```

## Sobrecarga de operadores

```
      public static bool operator true(Cuenta c) {
            return c.saldo > 0;
      }
```

```
      ....    (false)


public static Cuenta operator+(Cuenta c, Cliente titular)
{
      if (c)
            c.mTitulares.Add(titular);
      return c;
}
```