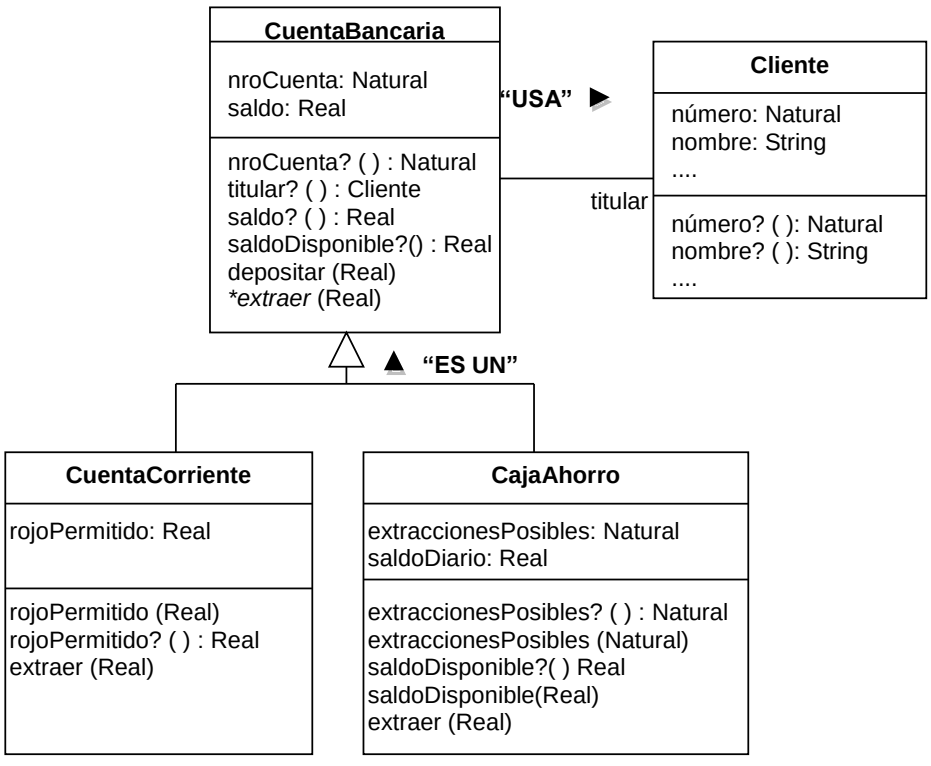
TDA CUENTAS BANCARIAS: UN EJEMPLO DE JERARQUIA DE CLASES

Se desea implementar un sistema bancario en el que existen dos tipos de cuentas bancarias, Cuenta Corriente y Caja de Ahorro con las siguientes carácterísticas:

Cuenta Corriente	**Cuenta Caja De Ahorro**
Tiene: un número de cuenta, un titular (suponemos 1 solo), un saldo y un rojo permitido(puede variar según el cliente)	Tiene: un número de cuenta, un titular (suponemos 1 solo), un saldo, una cantidad de extracciones posibles(mensuales) y un saldo diario disponible
Se desea: - conocer el número de cuenta - conocer al titular - conocer el saldo - conocer el rojo permitido - modificar el rojo Permitido - obtener saldo disponible - depositar un cierto monto - extraer un monto dado	Se desea: - conocer el número de cuenta - conocer al titular - conocer el saldo - conocer las extracciones posibles - modificar la cantidad de extracciones posibles - obtener saldo disponible - reiniciar saldo disponible - depositar un cierto monto - extraer un monto dado

Ambos TDA tienen similitudes en cuanto a su comportamiento (funcionalidad), y esto se debe a que tanto **CuentaCorriente como CajaAhorro son CUENTAS BANCARIAS.** Se crea un nuevo tipo de dato llamado **CuentaBancaria** que tiene todas las características comunes a todas las cuentas bancarias, *número de cuenta*, *titular*, *saldo*, *saldo disponible*, *depositar* y *extraer*. Luego se definen **CajaAhorro** y **CuentaCorriente** como subtipos del tipo **CuentaBancaria**, estas heredan todo ese comportamiento común y a su vez definen su comportamiento propio. Se obtiene de esta manera la siguiente jerarquía:



*Todas las cuentas bancarias, tienen la función extraer, pero no tienen forma general de definirla, ya que *extraer* de una cuenta corriente no tiene el mismo significado que extraer de una caja de ahorro. Una cuentra corriente puede tener un rojo permitido, mientras que de la caja de ahorro no puede extraerse mas dinero del que hay ni del monto diario permitido y además el número de extracciones mensuales tiene un límite. Cada una de las clases que hereden de cuenta Bancaria definirá extraer de manera apropiada.

A continuación se muestra la especificación algebraica en NEREUS y la implementación en C++.

TDA CUENTAS BANCARIAS: UN EJEMPLO DE JERARQUIA DE CLASES

Especificación algebraica en NEREUS (Incompleta)

CLASS CuentaBancaria
IMPORTS Real, Natural, Cliente
BASIC CONSTRUCTORS crearCB
EFFECTIVE
TYPES CuentaBancaria
OPERATIONS
crearCB: Natural*Cliente*Real-> CuentaBancaria;

nroCuenta?: CuentaBancaria -> Natural;
titular?: CuentaBancaria -> Cliente;
saldo?: CuentaBancaria -> Real;
saldoDisponible?: CuentaBancaria -> Real;
depositar:CuentaBancaria * Real->CuentaBancaria;
DEFERRED [1]
extraer: cuentaBancaria * Real-> CuentaBancaria;

AXIOMS
n: Natural; c: Cliente; s,m: Real;

nroCuenta? (crearCB(n, c, s)) = n;

titular? (crearCB(n, c, s)) = c;

saldo? (crearCB(n, c, s)) = s;

saldoDisponible? (crearCB(n, c, s)) = s;

depositar (crearCB(n, c, s) , m) =
crearCB (n ,c, s + m);

END-CLASS

//archivo: cuentaBancaria.h
//Declaración de la clase cuentaBancaria en C++

```
#include "cliente.h"
class CuentaBancaria
{
public:

CuentaBancaria(unsigned int numero, const
        Cliente & cliente, double saldoInicial);
   unsigned int obtenerNroCuenta() const;
   const Cliente & obtenerTitular() const;
   double obtenerSaldo() const;
   virtual double obtenerSaldoDisponible() const; [2]
   void depositar(double saldo);

   virtual void extraer(double monto) = 0; [3]
   virtual ~CuentaBancaria(); [4]

protected: [5]
   double saldo;
   unsigned int numero;
   Cliente titular;
};
```

//archivo: cuentaBancaria.cpp:
//implementación de la clase cuentaBancaria.

```
#include "cuentaBancaria.h"
CuentaBancaria::CuentaBancaria (unsigned int numero,
const Cliente & cliente, double saldoInicial) :
titular(cliente) [6]
{   this->numero = numero;
    saldo = saldoInicial;
}
unsigned int CuentaBancaria::obtenerNroCuenta() const
{   return numero;
}
const Cliente & CuentaBancaria::obtenerTitular() const
{   return titular;
}
double CuentaBancaria::obtenerSaldo() const
{   return saldo;
}
double CuentaBancaria::obtenerSaldoDisponible() const{
   return saldo;
}
void CuentaBancaria::depositar(double saldo)
{   this->saldo += saldo;}
CuentaBancaria::~CuentaBancaria(){
 }
```

[1] Cláusula **DEFERRED**: agrega tipos, funciones o axiomas incompletos, debido a que no hay por ejemplo suficientes axiomas, para definir el comportamiento de una función o no hay suficientes funciones para generar todos los valores del tipo dado (especificación incompleta).

[2] Incorpora una implementación por defecto pero permite su redefinición.

[3] Un método virtual puro (=0) no tiene implementación, obliga a las subclases de las cuales se crearán instancias a implementarlo. Una clase que tiene al menos un método virtual puro es una clase abstracta: no pueden generarse instancias de ella. Generalmente son abstracciones conceptuales, no representan objetos del mundo real.

[4] Si hay métodos virtuales puros el destructor debe ser virtual.

[5] Para poder acceder a las variables internas desde las subclases se deben definir protected.

[6] La clase Cliente no tiene constructor vacío, por lo tanto la variable titular debe inicializarse antes de comenzar el ámbito del constructor. Se especifica el constructor deseado después de los dos puntos (:).

TDA CUENTAS BANCARIAS: UN EJEMPLO DE JERARQUIA DE CLASES

Especificación algebraica en NEREUS

```
CLASS CuentaCorriente
INHERIT CuentaBancaria
BASIC CONSTRUCTORS crear
EFFECTIVE
TYPES CuentaCorriente
OPERATIONS
    crear: Natural * Cliente *  Real * Real->
                        CuentaCorriente;
    rojoPermitido?: CuentaCorriente -> Real;
    rojoPermitido: CuentaCorriente * Real ->
                        CuentaCorriente;
    extraer: CuentaCorriente (cc) * Real (m)->
                        CuentaCorriente
       PRE:saldo?(cc)–m >= rojoPermitido?(cc);
```

```
AXIOMS
    n: Natural; c: Cliente; s,rp,nrp,m: Real ;

    crear (n,c,s,rp) = crearCB (n,c,s);

    rojoPermitido? ( crear(n, c, s, rp) ) = rp;

    rojoPermitido ( crear(n, c, s, rp) ,nrp) =
                          crear(n, c, s, nrp);

    extraer ( crear(n, c, s, rp) , m) =
                        crear (n ,c, s – m,rp);
```

```
END-CLASS
```

// archivo: cuentaCorriente.h
//Declaración de la clase CuentaCorriente en C++

```
#include "cuentaBancaria.h"

class CuentaCorriente : public[7] CuentaBancaria
{
public:
    CuentaCorriente (unsigned int numero, const Cliente &
          cliente, double saldoInicial, double rojoPermitido);
    double obtenerRojoPermitido () const;
    void cargarRojoPermitido (double nuevoMonto);

    void extraer (double monto);
    ~CuentaCorriente();

private:
   double rojoPermitido;
};
```

// archivo: cuentaCorriente.cpp:
// implementación de la clase cuentaCorriente.

```
#include "CuentaCorriente.h"
#include <assert.h>

CuentaCorriente::CuentaCorriente(unsigned int numero,
          const Cliente & cliente, double saldoInicial,
          double rojoPermitido) : [8]
             CuentaBancaria(numero, cliente, saldoInicial)
{   this->rojoPermitido = rojoPermitido;
}
double CuentaCorriente::obtenerRojoPermitido() const
{   return rojoPermitido;
}
void CuentaCorriente::cargarRojoPermitido(
                                        double nuevoMonto)
{   rojoPermitido = nuevoMonto;
}
void CuentaCorriente::extraer(double monto)
{  assert ( saldo - monto >= - rojoPermitido);
   saldo -= monto;
}
CuentaCorriente::~CuentaCorriente()
{
}
```

[7] El tipo de herencia (público o privado) influye sobre los privilegios de acceso a los elementos de la superclase. public: herencia pública, todo lo que es declarado privado en la superclase, permanece privado en la subclase, y todo lo que es declarado público en la superclase, permanece público en la subclase.

[8] Se especifica el constructor deseado después de los dos puntos (:).

TDA CUENTAS BANCARIAS: UN EJEMPLO DE JERARQUIA DE CLASES

Especificación algebraica en NEREUS

```
CLASS CA
INHERIT CuentaBancaria
BASIC CONSTRUCTORS crear
EFFECTIVE
TYPES CA
OPERATIONS
    crear:Natural * Cliente * Real * Natural
          * Real -> CA;

    extraccionesPosibles?: CA -> Natural;
    extraccionesPosibles: CA x Natural -> CA;
    extraer: CA (ca) * Real (m)-> CA
     PRE:saldo?(ca)>=m and m<=saldoDisponible?(ca)
           and extraccionesPosibles(ca) >0 ;
    saldoDisponible?: CA -> Real;

    saldoDisponible: CA * Real -> CA;

AXIOMS
    n,ep,nep: Natural;
    c: Cliente;
    s,sd,nsd,m,: Real;

crear (n,c,s,ep,sd) = crearCB (n,c,s);

extraccionesPosibles? ( crear(n, c, s, ep, sd) ) = ep;

extraccionesPosibles(crear(n, c, s, ep, sd), nep)=
                        crear(n, c, s, nep, sd);

extraer ( crear(n, c, s, ep, sd) , monto) =
          crear (n, c, s - monto, ep-1, sd - monto);

saldoDisponible?( crear(n, c, s, ep, sd)) = sd;

saldoDisponible( crear(n, c, s, ep, sd) , nsd) =
                    crear(n, c, s, ep, nsd);

END-CLASS
```

// archivo: cuentaCajaAhorro.h
//Declaración de la clase cajaAhorro en C++

```
#include "cuentaBancaria.h"

class CuentaCajaAhorro: public CuentaBancaria
{
public:
   CuentaCajaAhorro(unsigned int numero, const Cliente &
         cliente, double saldoInicial, unsigned int cantidad
           double saldoDiario);
   ~CuentaCajaAhorro();

     unsigned int obtenerExtraccionesPosibles() const;
     void cargarExtraccionesPosibles(unsigned int cantidad);
     void extraer(double monto);

     double obtenerSaldoDisponible() const; //Se redefine la
                                  //implementación por defecto.
     void reiniciarSaldoDisponible(unsigned int);
private:
   unsigned int extraccionesPosibles;
   double saldoDiarioDisponible;
};
```

// archivo: cuentaCajaAhorro.cpp:
//implementación de la clase cuentaCajaAhorro.

```
#include "cuentaCajaAhorro.h"
#include <assert.h>

CuentaCajaAhorro::CuentaCajaAhorro(unsigned int numero,
   const Cliente & cliente, double saldoInicial, unsigned int
   cantidad, double saldoDiario):
         CuentaBancaria(numero, cliente, saldoInicial)
{   extraccionesPosibles = cantidad;
    saldoDiarioDisponible= saldoDiario;
}
unsigned int CuentaCajaAhorro::
       obtenerExtraccionesPosibles() const
{   return extraccionesPosibles;
}
void CuentaCajaAhorro::cargarExtraccionesPosibles(
    unsigned int cantidad)
{   extraccionesPosibles = cantidad;
}
void CuentaCajaAhorro::extraer(double monto)
{   assert ((saldo >= monto) &&
     (extraccionesPosibles > 0) &&
     (monto <= saldoDiarioDisponible));
  saldo-= monto;
  saldoDiarioDisponible-=monto;
  extraccionesPosibles-=1;
}
double CuentaCajaAhorro::obtenerSaldoDisponible() const
{   return saldoDiarioDisponible;
}
void CuentaCajaAhorro::reiniciarSaldoDisponible
                  (unsigned int  saldoDiario)
{   saldoDiarioDisponible = saldoDiario;
}
CuentaCajaAhorro::~CuentaCajaAhorro() {
}
```