FACULTAD DE INFORMÁTICA

Métodos virtuales, polimorfismo y clases abstractas

SOLUCIONES DEL TALLER

Programación orientada a objetos — Unidad 9

Autor: Luis Hernández Yáñez

Relaciones a través de punteros

Más sobre la lista de **Publicacion**es (*resolución en grupo*)

En la clase **Volumen** está declarado un atributo **_autor** de clase **Persona**. Como podemos tener un mismo autor para distintos volúmenes (libros, enciclopedias o artículos), queremos que ese atributo **_autor** sea mejor una relación entre el volumen y la persona que es autor de ese volumen.

Se pide que se convierta ese atributo en una relación entre el **Volumen** y la **Persona** que es su autor.

Cuando se muestre la información del **Volumen** se mostrará también la información de los objetos relacionados.

Pensad bien si hay otras clases de la aplicación que necesiten algún cambio debido a la nueva implementación del atributo anterior.

La clase **Volumen** con atributo relación

```
// Clase Volumen - Archivo de cabecera "volumen.h"
#ifndef volumen_h // Evitar inclusiones múltiples
#define volumen_h

#include <iostream>
#include <string>
using namespace std;
#include "persona.h"
#include "publicacion.h"

class Volumen : public Publicacion {
public:
  // F.C.O.
  Volumen(string = "", string = "", string = "",
          Fecha = Fecha(), Persona* = NULL);
  // Datos: título, ciudad, país, fecha de edición,
  // y vínculo con la Persona que es el autor.
  // Por defecto no está establecida la relación con
  // ninguna Persona (NULL).
  Volumen(const Volumen&);
```

(*continúa*)

La clase **Volumen** con atributo relación

```
  Volumen& operator=(const Volumen&);
  ~Volumen();
  // Accedente
  Persona* autor() const;
  // Mutador
  void autor(Persona*);
  // Otros métodos
  void leer();
  void mostrar() const;
private:
  Persona* _autor;
};

#endif
```

La clase **Volumen** con atributo relación

```
// Clase Volumen - Implementación "volumen.cpp"
#include "volumen.h"

Volumen::Volumen(string tit, string c, string p, Fecha f,
   Persona* a) : Publicacion(tit, c, p, f), _autor(a) { }
  // Las relaciones no se crean

Volumen::Volumen(const Volumen& otro) : Publicacion (otro) {
  // Las relaciones no se crean
  _autor = otro._autor;
}

Volumen& Volumen::operator=(const Volumen& otro) {
  Publicacion::operator=(otro);
  _autor = otro._autor;
  // Relacionado con la misma Persona (misma dirección)
  return *this;
}

Volumen::~Volumen() { } // Las relaciones no se destruyen
```

(*continúa*)

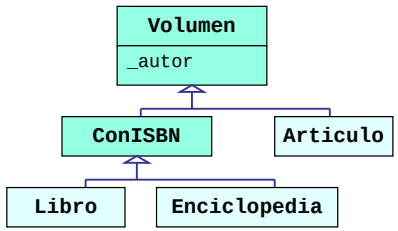
La clase **Volumen** con atributo relación

```
Persona* Volumen::autor() const { return _autor; }
// La dirección de la Persona con la que está relacionada

void Volumen::autor(Persona* p) { _autor = p; }
// Se relaciona con la Persona cuya dirección se proporciona

void Volumen::leer() {
  Publicacion::leer();
  // El objeto con el que está relacionado ya existe
  // y ya tiene sus datos leídos.
}

void Volumen::mostrar() const {
  Publicacion::mostrar();
  if(_autor == NULL) cout << "Sin datos. ";
  else cout << _autor->nombreCompleto() << ". ";
  // Se puede usar el objeto con el que se está relacionado
  // siempre y cuando esté establecida la relación
}
```


Relaciones a través de punteros

Al haber cambiado el constructor de la clase **Volumen** debemos cambiar también los constructores de todas sus subclases, ya que todas ellas hacen uso de ese constructor, quien requiere ahora un puntero para el atributo **_autor**, que ahora es una relación.

Las clases que tenemos que adaptar son: **Articulo**, **ConISBN**, **Libro** y **Enciclopedia**. Los cambios son los mismos en todos los casos.



Cambios en la clase **Articulo**

En **articulo.h**:

```
...
class Articulo : public Volumen {
public:
  // F.C.O.
  Articulo(string = "", string = "", string = "",
           Fecha = Fecha(), Persona* = NULL, string = "",
           int = 1, int = 1);
  ...
```

En **articulo.cpp**:

```
...
#include "articulo.h"

Articulo::Articulo(string tit, string c, string p, Fecha f,
  Persona* a, string s, int ini, int fin)
  : Volumen(tit, c, p, f, a), _serie(s), _inicial(ini),
  _final(fin) { }
  ...
```

Cambios en la clase **ConISBN**

En **conisbn.h**:

```
...
class ConISBN : public Volumen {
public:
  // F.C.O.
  ConISBN(string = "", string = "", string = "",
          Fecha = Fecha(), Persona* = NULL, string = "",
          int = 1, string = "");
  ...
```

En **conisbn.cpp**:

```
...
#include "conisbn.h"

ConISBN::ConISBN(string tit, string c, string p, Fecha f,
  Persona* a, string isbn, int ed, string edit)
  : Volumen(tit, c, p, f, a), _isbn(isbn), _edicion(ed),
  _editorial(edit) { }
  ...
```


Cambios en la clase **Libro**

En **libro.h**:

```
...
class Libro : public ConISBN {
public:
  // F.C.O.
  Libro(string = "", string = "", string = "", Fecha = Fecha(),
        Persona* = NULL, string = "", int = 1, string = "",
        int = 0);
  ...
```

En **libro.cpp**:

```
...
#include "libro.h"

Libro::Libro(string tit, string c, string p, Fecha f,
  Persona* a, string isbn, int ed, string edit, int pag)
  : ConISBN(tit, c, p, f, a, isbn, ed, edit), _paginas(pag) { }
  ...
```


Cambios en la clase **Enciclopedia**

En **enciclopedia.h**:

```
...
class Enciclopedia : public ConISBN {
public:
  // F.C.O.
  Enciclopedia(string = "", string = "", string = "",
               Fecha = Fecha(), Persona* = NULL, string = "",
               int = 1, string = "", int = 1);
  ...
```

En **enciclopedia.cpp**:

```
...
#include "enciclopedia.h"

Enciclopedia::Enciclopedia(string tit, string c, string p,
  Fecha f, Persona* a, string isbn, int ed, string edit, int vol)
  : ConISBN(tit, c, p, f, a, isbn, ed, edit), _volumenes(vol) { }
  ...
```


Prueba de la clase **Volumen** con atributo relación

```
// Prueba de la clase Lista (de publicaciones)
#include "articulo.h"
#include "libro.h"
#include "revista.h"
#include "listabib.h"

int main() {
  Lista biblioteca;
  Persona* autor1 = new Persona("223344G", 31, "Juan",
                                "Roque Gil");
  Persona* autor2 = new Persona("123867X", 53, "Miguel",
                                "Vegas Sanz");
  Revista* rev = new Revista("Programadores de C++", "Madrid",
                             "España", Fecha(1,10,2002), 5, 12,
                             "Ed. SoftTop");
  biblioteca.insertar(rev);
  Articulo* art =
    new Articulo("Los punteros o cómo morir en el intento",
                 "Madrid", "España", Fecha(11,9,2002), autor1,
                 "Programadores de C++", 134, 161);
  biblioteca.insertar(art);
```

(*continúa*)

Prueba de la clase **Volumen** con atributo relación

```
    art =
      new Articulo("Los atributos dinámicos, armas de doble filo",
          "Barcelona", "España", Fecha(23,9,2002), autor2,
          "Programadores de C++", 17, 31);
    biblioteca.insertar(art);

    Libro* lib = new Libro("POO con C++", "Madrid", "España",
                    Fecha(3,6,2001), autor1, "8-05400-2143-0",
                    2, "Ed. SoftTop", 237);
    biblioteca.insertar(lib);

    biblioteca.mostrar();

    delete autor1;
    delete autor2;

    return 0;
}
```

>>> biblio.cpp

Métodos virtuales, clonación y clases abstractas

La biblioteca funcionando correctamente (*resolución en grupo*)

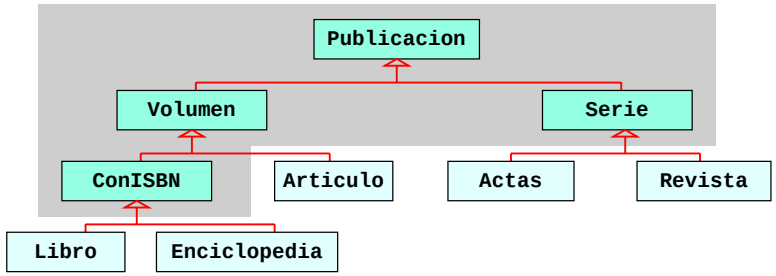
Para conseguir que la lista de publicaciones funcione bien del todo nos faltan unos pocos pasos:

1. Incorporar la vinculación dinámica en aquellos métodos afectados de problemas de vinculación (convirtiéndolos en virtuales).
2. Completar las clases de publicaciones con un método de clonación que permita la duplicación de la lista de publicaciones.
3. Incorporar en la lista de publicaciones el constructor de copia y el operador de asignación.

De paso, aprovecharemos para convertir las clases de publicaciones que no son terminales en la jerarquía en clases abstractas, para reflejar el hecho de que no se crean tales tipos de publicaciones. Son las clases que tienen definida alguna subclase.

Métodos virtuales

Las clases donde hay métodos que necesitan ser vinculados dinámicamente son las clases no terminales de la jerarquía:



Y los métodos que deben hacerse virtuales (además de los destructores) son **`leer()`** y **`mostrar()`**.

Clonación y clases abstractas

En TODAS las clases de publicaciones debe haber un método **`clon()`** que devuelva una copia dinámica del objeto receptor. En todas las clases debe estar definido igual:

```
Publicacion* clon() const;
```

En las clases no terminales debe ser además un método virtual.

Por otro lado, nunca se van a crear ejemplares de las clases no terminales de la jerarquía de publicaciones, por lo que deben convertirse en clases abstractas. Para hacerlo basta con que alguno de los métodos virtuales sea virtual puro.

Los destructores y los métodos **`leer()`** y **`mostrar()`** tienen implementación en todas las clases, por lo que no nos sirven.

Pero si no se van a crear ejemplares de esas clases, está claro que tampoco se van a necesitar clones, por lo que el método **`clon()`**, que es virtual es esas clases, no necesita estar implementado allí, pudiendo estar declarado como virtual puro. Además, basta con que esté declarado así en la clase **`Publicacion`** y no se redefina en **`Volumen`**, **`ConISBN`** o **`Serie`**.

La clase **Publicacion** final (abstracta)

```
// Clase Publicacion - Archivo de cabecera "publicacion.h"
#ifndef publicacion_h
#define publicacion_h

#include <iostream>
#include <string>
using namespace std;
#include "fecha.h"

class Publicacion {
public:
  Publicacion(string = "", string = "", string = "",
              Fecha = Fecha());
  // Datos: título, ciudad, país, fecha de edición.
  Publicacion(const Publicacion&);
  Publicacion& operator=(const Publicacion&);
  virtual ~Publicacion();

  string titulo() const;
  string ciudad() const;
```

(*continúa*)

La clase **Publicacion** final (abstracta)

```
  string pais() const;
  Fecha fecha() const;

  void titulo(string);
  void ciudad(string);
  void pais(string);
  void fecha(Fecha);

  virtual void leer();
  virtual void mostrar() const;
  virtual Publicacion* clon() const = 0;
private:
  string _titulo;
  string _ciudad, _pais;
  Fecha* _fecha;
};

#endif
```

La implementación no cambia

La clase **Volumen** final (abstracta)

```
...
class Volumen : public Publicacion {
public:
  Volumen(string = "", string = "", string = "",
          Fecha = Fecha(), Persona* = NULL);
  // Datos: título, ciudad, país, fecha de edición, puntero
  // al autor. Por defecto no está establecida la relación
  // con ninguna Persona (NULL).
  Volumen(const Volumen&);
  Volumen& operator=(const Volumen&);
  virtual ~Volumen();
  Persona* autor() const;
  void autor(Persona*);
  virtual void leer();
  virtual void mostrar() const;
private:
  Persona* _autor;
};

#endif
```

También es abstracta porque no implementa **clon()**

La implementación no cambia

La clase **Serie** final (abstracta)

```
...
class Serie : public Publicacion {
public:
  Serie(string = "", string = "", string = "",
        Fecha = Fecha(), int = 1, int = 1);
  // Datos: título, ciudad, país, fecha de edición, volumen
  // y número.
  Serie(const Serie&);
  Serie& operator=(const Serie&);
  virtual ~Serie();
  int volumen() const;
  int numero() const;
  void volumen(int);
  void numero(int);
  void insertar(Articulo*);
  virtual void leer();
  virtual void mostrar() const;
private:
  ...
```

También es abstracta porque no implementa **clon()**

La implementación no cambia

La clase **ConISBN** final (abstracta)

```
...
class ConISBN : public Volumen {
public:
  ConISBN(string = "", string = "", string = "",
          Fecha = Fecha(), Persona* = NULL, string = "",
          int = 1, string = "");
  // Datos: título, ciudad, país, fecha de edición, autor, ISBN,
  // edición, editorial.
  ConISBN(const ConISBN&);
  ConISBN& operator=(const ConISBN&);
  virtual ~ConISBN();
  string isbn() const;
  int edicion() const;
  string editorial() const;
  void isbn(string);
  void edicion(int);
  void editorial(string);
  virtual void leer();
  virtual void mostrar() const;
private:
  ...
```

También es abstracta porque no implementa **clon()**

La implementación no cambia

>>> conisbn.h

La clase **Articulo** final (no abstracta)

Tan sólo hay que implementar el método **clon()**.

En **articulo.h**

```
  ...
  void final(int);
  void leer();
  void mostrar() const;
  Publicacion* clon() const;
private:
  ...
```

En **articulo.cpp**

```
...
Publicacion* Articulo::clon() const {
  Articulo* p = new Articulo(*this);
  return p;
}
```


La clase **Libro** final (no abstracta)

Tan sólo hay que implementar el método **clon()**.

En **libro.h**

```
  ...
  void paginas(int);
  void leer();
  void mostrar() const;
  Publicacion* clon() const;
private:
  ...
```

En **libro.cpp**

```
...
Publicacion* Libro::clon() const {
  Libro* p = new Libro(*this);
  return p;
}
```


La clase **Enciclopedia** final (no abstracta)

Tan sólo hay que implementar el método **clon()**.

En **enciclopedia.h**

```
  ...
  void volumenes(int);
  void leer();
  void mostrar() const;
  Publicacion* clon() const;
private:
  ...
```

En **enciclopedia.cpp**

```
...
Publicacion* Enciclopedia::clon() const {
  Enciclopedia * p = new Enciclopedia(*this);
  return p;
}
```

La clase **Actas** final (no abstracta)

Tan sólo hay que implementar el método **clon()**.

En **actas.h**

```
  ...
  void congreso(string);
  void leer();
  void mostrar() const;
  Publicacion* clon() const;
private:
  ...
```

En **actas.cpp**

```
...
Publicacion* Actas::clon() const {
  Actas* p = new Actas(*this);
  return p;
}
```


La clase **Revista** final (no abstracta)

Tan sólo hay que implementar el método **clon()**.

En **revista.h**

```
  ...
  void editorial(string);
  void leer();
  void mostrar() const;
  Publicacion* clon() const;
private:
  ...
```

En **revista.cpp**

```
...
Publicacion* Revista::clon() const {
  Revista* p = new Revista(*this);
  return p;
}
```


La clase **Lista** (de publicaciones) final

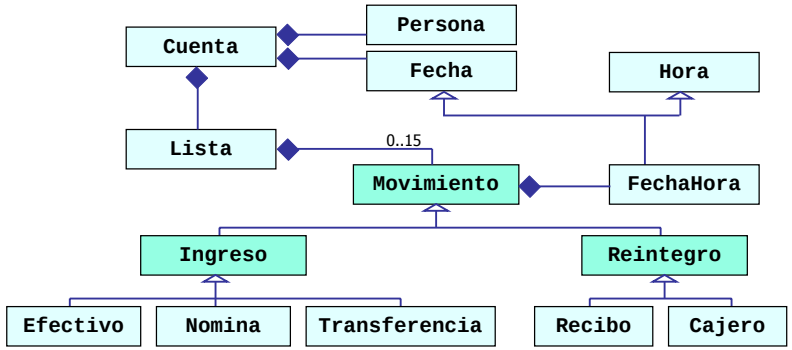
```
// Clase Lista (de publicaciones) - Cabecera "listabib.h"

#ifndef listabib_h
#define listabib_h

#include "publicacion.h"

class Lista {
public:
  Lista();
  Lista(const Lista&);
  Lista& operator=(const Lista&);
  ~Lista();
  bool llena() const;
  ...
```


La clase **Lista** (de publicaciones) final

```
...
Lista::Lista() : _cont(0) {}

Lista::Lista(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]->clon();
}

Lista& Lista::operator=(const Lista& otra) {
  // Primero eliminamos los elementos de la lista
  for(int i = 0; i < _cont; i++) delete _array[i];
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]->clon();
  return *this;
}
...
```

La lista polimórfica de movimientos

Para completar las clases de la aplicación de cuentas bancarias debemos aplicar todo lo anterior: métodos virtuales, clonación, clases abstractas y duplicación de la lista polimórfica.



La clase **Movimiento** final (abstracta)

```
...
class Movimiento { // Clase abstracta
public:
  Movimiento(string = "", double = 0, FechaHora = FechaHora());
  // Datos: Concepto, cantidad y momento
  Movimiento(const Movimiento&);
  Movimiento& operator=(const Movimiento&);
  virtual ~Movimiento();
  string concepto() const;
  double cantidad() const;
  FechaHora momento() const;
  void concepto(string);
  void cantidad(double);
  void momento(FechaHora);
  void leer();
  void mostrar() const;
  virtual double apunte() const = 0; // Método virtual puro
  virtual Movimiento* clon() const = 0; // Método virtual puro
private:
  ...
```

La implementación no cambia

>>> movimiento.h

Las clases **Ingreso** y **Reintegro** finales (abstractas)

```
...
class Ingreso : public Movimiento { // Clase abstracta
public:
  Ingreso(string = "", double = 0, FechaHora = FechaHora());
  // Datos: Concepto, cantidad y momento
  Ingreso(const Ingreso&);
  Ingreso& operator=(const Ingreso&);
  virtual ~Ingreso();
  double apunte() const;
};
...
```

>>> ingreso.h

```
...
class Reintegro : public Movimiento { // Clase abstracta
public:
  Reintegro(string = "", double = 0, FechaHora = FechaHora());
  // Datos: Concepto, cantidad y momento
  Reintegro(const Reintegro&);
  Reintegro& operator=(const Reintegro&);
  virtual ~Reintegro();
  double apunte() const;
};
...
```

>>> reintegro.h

Las implementaciones no cambian

La clase **Efectivo** final

```
...
class Efectivo : public Ingreso {
public:
  Efectivo(double = 0, FechaHora = FechaHora());
  // Datos: Cantidad y momento
  Efectivo(const Efectivo&);
  Efectivo& operator=(const Efectivo&);
  ~Efectivo();
  Movimiento* clon() const;
};
...
```

>>> efectivo.h

```
...
Movimiento* Efectivo::clon() const {
  Efectivo* m = new Efectivo();
  *m = *this;
  return m;
}
```

>>> efectivo.cpp

La clase **Nomina** final

```
...
class Nomina : public Ingreso {
public:
  Nomina(double = 0, FechaHora = FechaHora());
  // Datos: Cantidad y momento
  Nomina(const Nomina&);
  Nomina& operator=(const Nomina&);
  ~Nomina();
  Movimiento* clon() const;
};
...
```

>>> nomina.h

```
...
Movimiento* Nomina::clon() const {
  Nomina* m = new Nomina();
  *m = *this;
  return m;
}
```

>>> nomina.cpp

La clase **Transferencia** final

```
...
class Transferencia : public Ingreso {
public:
  Transferencia(double = 0, FechaHora = FechaHora());
  // Datos: Cantidad y momento
  Transferencia(const Transferencia&);
  Transferencia& operator=(const Transferencia&);
  ~Transferencia();
  Movimiento* clon() const;
};
...
```

>>> transferencia.h

```
...
Movimiento* Transferencia::clon() const {
  Transferencia* m = new Transferencia();
  *m = *this;
  return m;
}
```

>>> transferencia.cpp

La clase **Recibo** final

```
...
class Recibo : public Reintegro {
public:
  Recibo(double = 0, FechaHora = FechaHora());
  // Datos: Cantidad y momento
  Recibo(const Recibo&);
  Recibo& operator=(const Recibo&);
  ~Recibo();
  Movimiento* clon() const;
};
...
```

>>> recibo.h

```
...
Movimiento* Recibo::clon() const {
  Recibo* m = new Recibo();
  *m = *this;
  return m;
}
```

>>> recibo.cpp

La clase **Cajero** final

```
...
class Cajero : public Reintegro {
public:
  Cajero(double = 0, FechaHora = FechaHora());
  // Datos: Cantidad y momento
  Cajero(const Cajero&);
  Cajero& operator=(const Cajero&);
  ~Cajero();
  Movimiento* clon() const;
};
...
```

>>> cajero.h

```
...
Movimiento* Cajero::clon() const {
  Cajero* m = new Cajero();
  *m = *this;
  return m;
}
```

>>> cajero.cpp

La clase **Lista** final

En **listamov.h**

```
...
Lista();
Lista(const Lista&);
Lista& operator=(const Lista&);
...
```

En **listamov.cpp**

```
...
Lista::Lista(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]->clon();
}

Lista& Lista::operator=(const Lista& otra) {
  for(int i = 0; i < _cont; i++) delete _array[i];
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]->clon();
  return *this;
}
```


La clase **Cuenta** final

```
...
Cuenta::Cuenta(Persona cliente, Fecha fecha, double saldo)
  : _saldo(saldo) {
  _cliente = new Persona(cliente);
  _fecha = new Fecha(fecha);
  _movs = new Lista(); }

Cuenta::Cuenta(const Cuenta& otra) {
  _cliente = new Persona(*otra._cliente);
  _fecha = new Fecha(*otra._fecha);
  _saldo = otra._saldo;
  _movs = new Lista(*otra._movs); }

Cuenta& Cuenta::operator=(const Cuenta& otra) {
  *_cliente = *otra._cliente;
  *_fecha = *otra._fecha;
  _saldo = otra._saldo;
  *_movs = *otra._movs;
  return *this; }
...
```

La cabecera (**.h**) no cambia

Un programa principal de prueba

```
#include <iostream>
#include <string>
using namespace std;
#include "fechahora.h"
#include "efectivo.h"
#include "transferencia.h"
#include "nomina.h"
#include "recibo.h"
#include "cajero.h"
#include "listamov.h"
#include "cuenta.h"

int main() {
  Cuenta* cc = new Cuenta(Persona("223344G", 25, "Javier",
               "Vegas Vegas"), Fecha(23, 11, 2002), 100);
  cc->extracto();
  FechaHora fh(Fecha(29,11,2002), Hora(10,36,59));
  cc->nuevo(new Efectivo(125, fh));
  fh = fh + 90012; // Un tiempo después
  cc->nuevo(new Transferencia(325.60, fh));
```

(*continúa*)

Un programa principal de prueba

```
  cc->nuevo(new Transferencia(325.60, fh));
  fh = fh + 70235; // Un tiempo después
  cc->nuevo(new Cajero(50, fh));
  fh = fh + 215014; // Un tiempo después
  cc->nuevo(new Nomina(1025.73, fh));
  fh = fh + 100001; // Un tiempo después
  cc->nuevo(new Recibo(120, fh));
  cc->extracto();
  Cuenta* cc2 = new Cuenta(*cc);
  delete cc;
  cc2->extracto();
  Cuenta* cc3 = new Cuenta();
  *cc3 = *cc2;
  delete cc2;
  cc3->extracto();
  delete cc3;

  return 0;
}
```