I102 - Paradigmas de Programación

OOP - HERENCIA

6

Prof. Mariano Scaramal

Ingeniería en Inteligencia Artificial

Esta Presentación Cubre…

- Conceptos:
  - Tipos de relaciones entre objetos
  - Herencia:
    - Herencia Pública y Privada
    - Constructores y Destructores en Herencia
    - Herencia Múltiple
    - Ejemplo de Herencia

Tipos de Relación entre Objetos

Los tipos de relación son las formas en que las clases interactúan y se conectan entre sí para modelar relaciones. Hay 6 tipos:

1. Asociación
2. Agregación
3. Composición
4. Generalización
5. Realización
6. Dependencia

Estas relaciones representan conceptos léxicos como “is-a”, “has-a”, “uses-a” y “is-like-a”. Por ejemplo, “Un perro is-a mamífero”, “Un auto has-a motor”, etc. Estas relaciones no tienen que ser uno a uno.

Relaciones entre Objetos - Asociación

La asociación representa una relación entre dos o más objetos que interactúan y quedan relacionados sin una fuerte dependencia entre ellos. No existe un ownership, lo que significa que si un objeto es eliminado, el otro puede continuar existiendo de manera independiente. Esta relación a veces se la llama relación estructural por el vínculo lógico que genera entre los objetos.

Ejemplo: un libro y su autor en un sistema de biblioteca. Ni el autor ni el libro tienen ownership sobre el otro. Si se borra el objeto autor, el libro aún sigue estando en el sistema y viceversa. También se puede pensar que porque el libro se quema en un incendio, el autor deja de existir o modifica su vida debido a esto. Sus lifetimes son independientes.

Entre los objetos existe una relación pero esta es débil, además, sus ciclos de vida (lifetimes) son independientes.

Relaciones entre Objetos - Agregación

La agregación representa una relación del tipo has-a entre un objeto que actúa como un todo (whole) y sus partes (parts). En este caso, el todo mantiene una relación con las partes, pero sin una dependencia total: el ciclo de vida de un objeto no depende del otro. Es decir, aunque los objetos están más estrechamente relacionados que en una simple asociación, las partes pueden existir de manera independiente del todo.

Ejemplo: profesores (*parts*) y un departamento de una universidad (*whole*). Los profesores pueden ser contratados, administrados económicamente, evaluados, organizados en roles, etc. En cuanto a su lifetime, en el caso de que departamento se cierre, los profesores pueden cambiar de departamento sin dejar de ser profesores. O bien, si renuncian todos los profesores, el departamento deberá de contratar más profesores, pero no dejar de existir.

En este caso hay una relación más intima entre los objetos dado que existe una acción entre el departamento y los profesores.

Relaciones entre Objetos - Composición

Composición representa una relación del tipo “has-a” que es más fuerte que agregación. En este caso, un objeto tiene ownership sobre otros y controla su lifetime (ciclo de vida). Esto significa que, cuando el objeto contenedor es destruido, sus partes (objetos contenidos) también son eliminadas.

Ejemplo: una casa (*whole*) y sus habitaciones (*parts*). Los cuartos por si solos no tienen sentidos fuera de una casa. Al mismo tiempo, si una casa es destruida, las habitaciones también son destruidas porque son una parte integral de la misma.

Relaciones entre Objetos - Generalización

Generalización es una relación “is-a” que permite crear, a partir de una clase más abstracta (llamada superclase o clase base), una clase más específica (llamada subclase o clase derivada). La subclase hereda propiedades y métodos de la superclase, los cuales pueden ser modificados y o bien pueden agregar nuevos.

Ejemplo: tener una clase mamífero que tenga atributos como cantidad de patas, desplazarse, etc. Clases derivadas de estas pueden ser un gato (“is-a” mamífero) y una ballena (“is-a” mamifero). Un objeto gato tiene 4 patas y se desplaza corriendo y el objeto ballena no tiene patas y se desplaza nadando.

Herencia y potencialmente polimorfismo permiten la implementación de esta relación.

Relaciones entre Objetos - Realización

Realización se refiere a la implementación de una interfaz o una clase abstracta (implements-a). Es decir, consiste en proveer el comportamiento definido mediante una clase abstracta o una interfaz, y luego implementarlo en una clase concreta.

Ejemplo: en un sistema se utilizan clases círculo y rectángulo que requieren la implementación de calcularArea() y calcularPerimetro(). Para asegurar su implementación se utiliza una interfaz o una clase abstracta. Como estas figuras tienen fórmulas distintas para estos métodos, ambas clases deberán proporcionar sus propias implementaciones de estos métodos.

Realización está fuertemente relacionada con polimorfismo. Este es el tema de la próxima presentación.

Relaciones entre Objetos - Dependencia

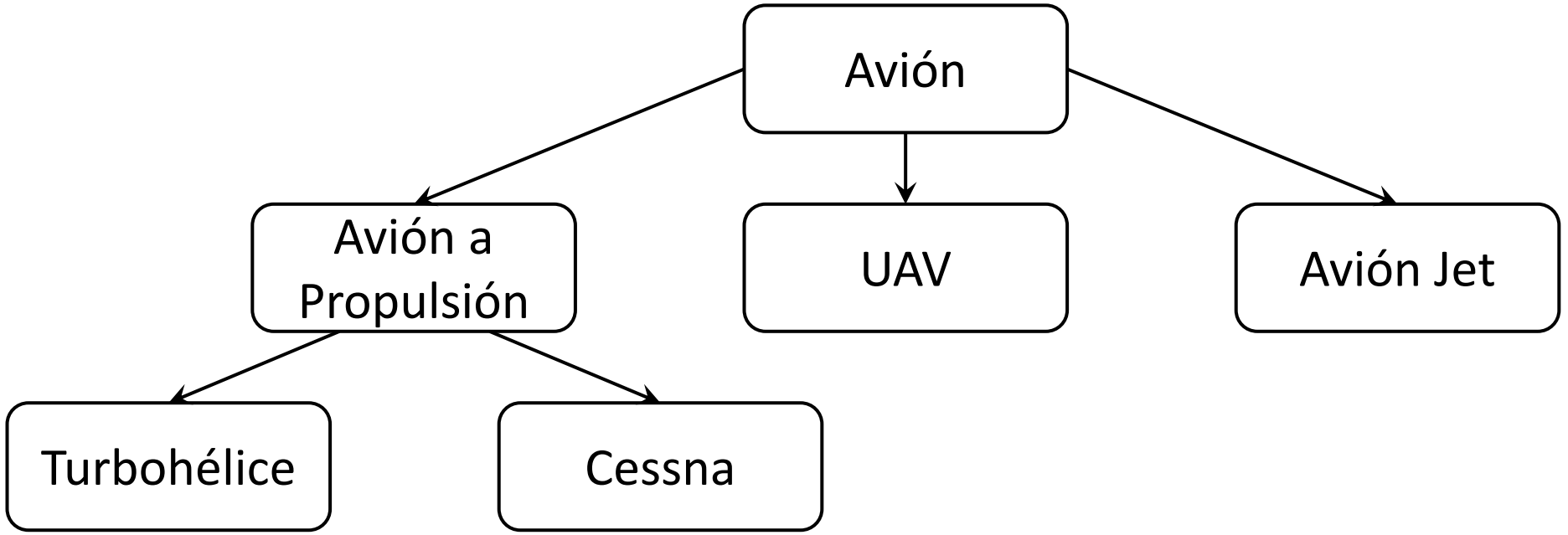
Dependencia es una relación del tipo “uses-a” dado que la relación implica que una clase (o método) utiliza otro objeto para cumplir con su función o proveer alguna capacidad. Sin embargo, a pesar de que use este objeto, no se comparte ownership.

Ejemplo: en un servicio de correo electrónico, una clase Usuario será dependiente de una clase ServicioEmail para poder enviar un mail. En este caso, al usuario no le interesa como funciona ServicioEmail, sólo lo utiliza para poder cumplir con la capacidad de compartir un documento vía email. Es importante notar que la clase ServicioEmail no pertenece a un determinado objeto Usuario sino que todos los objetos Usuario pueden utilizarla para enviar mails.

Herencia

Herencia implica la creación de una nueva clase a partir de una clase ya existente de la cual hereda sus atributos y métodos. Esto es, la nueva clase, la *clase derivada,* hereda sus datos y funciones miembro de la *clase base*.

Conceptualmente las clases derivadas son más específicas que las clase base.



Herencia (cont'd)

Herencia pública (public inheritance):

- La clase derivada tiene acceso a datos y funciones miembro del tipo *public* y *protected.*
- Los objetos derivados tiene sus propios elementos miembro.
- La palabra reservada *protected* significa que solo se puede acceder por el mismo objeto o su derivado a través de herencia.

Herencia privada (private inheritance):

- Las clases derivadas verán los miembros de la clase base como miembros privados.
- Quiere decir que estos miembros no serán miembros públicos de la clase derivada. No se pueden usar fuera de la clase, sólo dentro de ellas.

Ejemplo – Herencia Pública y Privada

Suponiendo que se tengan las siguiente clases base:

```
// Clase Base Avion (se usará como herencia pública)
class Avion {
public:
   Avion(std::string modelo) : modelo(modelo) {}
   void volar() const {
      std::cout << modelo << " esta volando!" <<
      std::endl;
   }

protected:
   std::string modelo;
};
```

```
// Clase Base Motor (se usará como herencia privada)
class Motor {
public:
   Motor(float horsepower) : horsepower(horsepower) {}
   void encenderMotor() const {
      std::cout << "Encendiendo motor con " <<
      horsepower << " HP." << std::endl;
   }

protected:
   float horsepower;
};
```

Ejemplo – Herencia Pública y Privada (cont'd)

Universidad de SanAndrés

Al crearse la base *PropPlane*, esta hereda los miembros públicos y protegidos de la clase *Avion* y *Motor*. No obstante, los miembros de *Motor* no pueden ser accedidos desde el objeto *cessna* en *main.*

```
// Clase deriveda PropPlane (herencia publica de Avion y
// privada de Motor)
class PropPlane : public Avion, private Motor {
public:
   PropPlane(std::string modelo, float horsepower)
      : Avion(modelo), Motor(horsepower) {}

   void endenderPropPlane() {
   // Esta permitido llamar a encenderMotor() desde
   // PropPlane porque  Motor se hereda en forma privada
      Motor::encenderMotor();
      std::cout << modelo << " esta listo para despegar!" <<
      std::endl;
   }
protected:
  int nroPasajeros = 1;
};
```

Aquí podría usarse Avion::modelo, pero no hay ambigüedad.

```
int main() {
   PropPlane cessna("Cessna 172", 160);
   // Se llama un metodo de PropPlane
   cessna.encenderPropPlane();
   // Se llama este metodo heredado de Avion
   cessna.volar();
   // cessna.encenderMotor();  // Error: encenderMotor() es
   //privada en PropPlane porque se heredo de Motor en forma
   //privada (ver class PropPlane)
   // std::cout << cessna.nroPasajeros;  // Error: nroPasajeros no
   // se puede acceder!   Por qué?

   return 0;
}
```

Constructores y Destructores en Herencia

Una clase derivada siempre llamará el constructor de su clase base primero.

Los destructores son llamados en orden inverso, la clase derivada primero.

Si no existe un constructor default en la clase base, de ser necesario el compilador creará uno vacío para poder instanciar la clase base.

```
Creando objeto de clase Derivada
Constructor de clase base
Constructor de clase intermedia
Constructor de clase derivada
Fin del main.
Destructor de clase derivada
Destructor de clase intermedia
Destructor de clase base
```

```
#include <iostream>

class Base {
public:
   Base() { std::cout << "Constructor de clase base" << std::endl; }
   ~Base() { std::cout << "Destructor de clase base" << std::endl; }
};

class Intermedia : public Base {
public:
   Intermedia() { std::cout << "Constructor de clase intermedia" << std::endl; }
   ~Intermedia() { std::cout << "Destructor de clase intermedia" << std::endl; }
};

class Derivada : public Intermedia {
public:
   Derivada() { std::cout << "Constructor de clase derivada" << std::endl; }
   ~Derivada() { std::cout << "Destructor de clase derivada" << std::endl; }
};

int main() {
   std::cout << "Creando objeto de clase Derivada\n";
   Derivada obj;
   std::cout << "Fin del main.\n";
   return 0;
}
```

Constructores Explícitos e Implícitos en Herencia

- El constructor de la clase base puede ser llamado desde la clase derivada para evitar repetir código.
- Estos llamados pueden ser explícitos o implícitos (ver ejemplos en la diapositiva siguiente)
- En la forma explicita, el llamado se realiza al llamar el constructor de la clase base desde el constructor de la clase derivada, por lo tanto, el mismo debe escribirse.
- En la forma implícita, no es necesario escribir el constructor que llama al constructor de la clase base.
- Se pueden heredar múltiples constructores de la clase base, pero estos también pueden ser sobrescritos.

Llamadas Explícitas e Implícitas a Constructores

```
#include <iostream>

class Base {
public:
   Base(int x, int y) {  // Constructor de la clase Base
      std::cout << "El constructor base es llamado con x=" <<
      x << ", y=" << y << std::endl;
   }
};

class Derived : public Base {
public: // Se llama el constructor de Base explicitamente
   Derived(int xx, int yy) : Base(xx, yy) {
      std::cout << “Se llama al constructor derivado" << std::endl;
   }
};

int main() { // Se llama Base(10,20) con Derived(10,20)
   Derived obj(10, 20);
   return 0;
}
```

Llamada explícita al constructor base

```
#include <iostream>

class Base {
public:
   Base(int x, int y) {  // Constructor de la clase Base
      std::cout << "El constructor base es llamado con x=" <<
      x << ", y=" << y << std::endl;
   }
};

class Derivada : public Base {
public: // Se heredan el constructor de Base
   using Base::Base;
};

int main() {
   // No se define el constructor de Derivada
   Derivada obj(10, 20);
   return 0;
}
```

Llamada implícita al constructor base

Llamadas Implícitas Múltiples a Constructores

```
#include <iostream>

class Base {
public:
   Base() {  // Constructor default de Base
      std::cout << "Constructor Default de Base\n";
   }

   Base(int x) {  // Constructor de Base con 1 param
      std::cout << "Constructor de Base con x=" << x
      << std::endl;
   }

   Base(int x, int y) {// Constructor de Base con 2 params
      std::cout << "Constructor de Base con x=" << x
      << " e y = " << y << std::endl;
   }
};
```

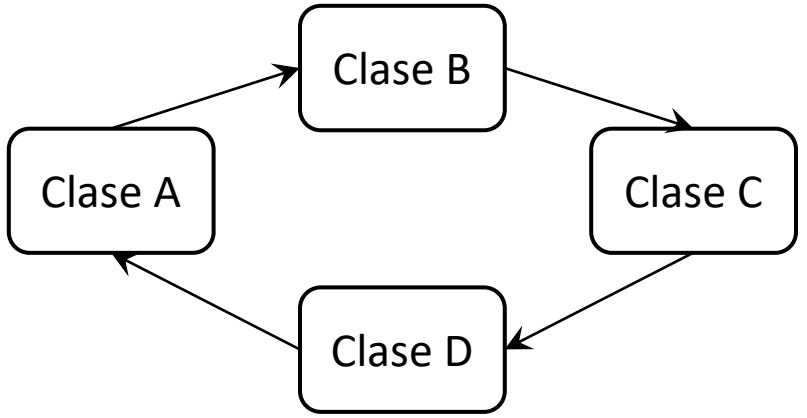
```
class Derivada : public Base {
public: // Se heredan todos los constructores de Base
   using Base::Base;

   Derivada(int x, int y) {
      std::cout << "Constructor de Derivada con x="
       << x << " e y=" << y << std::endl;
   }// Se sobreescribe el const de Base de 2 params
};

int main() {
   // Constructor default de Base
   Derivada obj1;
   // Constructor de 1 param de Base
   Derivada obj2(10);
   // Constructor con 2 params de Derivada
   Derivada obj3(10, 20);
   return 0;
}
```

Herencia Múltiple

- Como se vio anteriormente, una clase puede tener varias clases base.
- De esta manera, una clase puede heredar múltiples comportamientos y datos.
- Si dos métodos de las clases bases poseen el mismo nombre, puede producir ambigüedad. En esto casos, siempre conviene usar las referencias a las clases: Base1::comer() o Base2::comer().
- Herencia múltiple puede conducir a ciclos cerrados de herencia que son ilegales en C++ (se producirá un error).



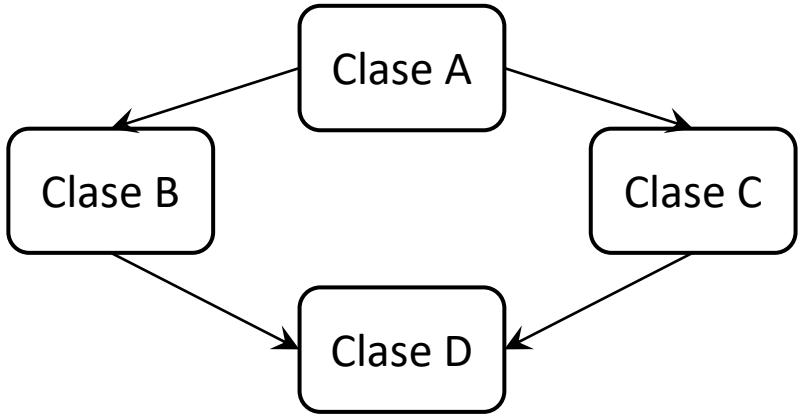
Herencia Múltiple – Problema del Diamante

Otro problema que se puede generar con herencia múltiple es *el problema del diamante*, cuyo nombre proviene de las relaciones de herencia entre las clases.

Ocurre cuando hay varias clases base (clases B y C) que tienen una clase derivada (clase D) y una misma clase ancestro (clase A).

El problema ocurre cuando se desea hacer uso de un método de la clase ancestro, para la cual existen dos posibles caminos (a través de la clase B y la C), generándose una ambigüedad.

Notar que el sentido de las flechas es distinto al caso anterior. La clase A es la superclase de B y C, y estas son las clases base de D.



Problema del Diamante (en código)

```
#include <iostream>
using namespace std;

class A {
public:
   A() { cout << "Constructor de A" << endl; }
   void show() { cout << "Showing..." << endl; }
};

class B : public A {
public:
   B() { cout << "Constructor de B" << endl; }
};

class C : public A {
public:
   C() { cout << "Constructor de C" << endl; }
};

class D : public B, public C {
public:
   D() { cout << "Constructor de D" << endl; }
};

int main() {
   D d;
   // Hay dos caminos para llegar a show()
   d.show(); //Error: "D::show" is ambiguous

   return 0;
}
```

Como Evitar el Problema del Diamante

```
#include <iostream>
using namespace std;

class A {
public:
   A() { cout << "Constructor de A" << endl; }
   void show() { cout << "Showing..." << endl; }
};

class B : virtual public A {
public:
   B() { cout << "Constructor de B" << endl; }
};

class C : virtual public A {
public:
   C() { cout << "Constructor de C" << endl; }
};
```

```
class D : public B, public C {
public:
   D() { cout << "Constructor de D" << endl; }
};

int main() {
   D d;
   // Hay dos caminos para llegar a show()
   d.show(); // No hay error

   return 0;
}
```

Imprime:

```
Constructor de A
Constructor de B
Constructor de C
Constructor de D
Showing...
```

Ejemplo de Herencia

```
#include <iostream>

class Point { // Se crea la clase Punto
public:
   Point (float a, float b): x(a), y(b){}
   void setPoint(float a, float b){x = a; y = b;}
   float getX(){return x;}
   float getY(){return y;}
private:
   float x, y;
};

class Circle : public Point { // Se crea la clase Circulo
public:
   const float PI =  3.14159;
   Circle (float r, float a, float b): rCirc(r), Point(a, b){}
   void setRadius(double r){rCirc = (r>=0 ? r : 0);}
   float getRadius() const {return rCirc;}
   double area() const { return PI*rCirc*rCirc;}
protected:
   float rCirc;
};
```

```
class Cylinder : public Circle{  // Se crea la clase Cilindro
public:
   Cylinder(float h, float r, float a, float b) : hCyl(h), Circle(r,a,b){}
   void setHight(double h){hCyl = (h>=0 ? h : 0);}
   float getHeight() const {return hCyl;}
   float area(){ return 2*(Circle::area() + PI*hCyl*rCirc); }
   float volume(){ return hCyl*Circle::area()}
protected:
   float hCyl;
};

int main(){
   Cylinder cy(3,1,5,4);
   std::cout << "El cilindro se encuentra en: (" <<
   cy.getX() << ", " << cy.getY() << ")" << std::endl;
   std::cout << "Tiene un area de: " <<
   cy.area() << std::endl;
   std::cout << "Tiene un volumen de: " <<
   cy.volume() << std::endl;
   return 0;
}
```

Universidad de SanAndrés

Preguntas?