Tema 7: Relaciiones entre objetos. Herencia. Polimorfismo. Enlace dinámico.

Herencia

- Las relaciones de herencia suelen representar situaciones donde las clases base son o representan conceptos más generales mientras que las derivadas son más específicas.
- Ya sabemos que modelan una relación de *parecido* entre elementos del mundo real: **es un**.
- Aunque, si el lenguaje lo permite, podemos modelar otro tipo de relaciones con ellas.
- Cuando una clase base es abstracta y además no tiene *datos* se dice que es un interfaz.

C++ y las relaciones de herencia.

- **C++** permite *etiquetar* la relación de herencia entre dos clases.
- Una clase puede heredar de otra de forma **publica**, **privada** y/o **protegida**.

```
class Derived : public Base {};
class Derived : private Base {};
class Derived : protected Base {};
```

- IMPORTANTE:
  - **Herencia pública ::** Lo público en la clase base es público y lo protegido en la clase base es protegido en la clase derivada. Lo privado de una clase siempre es privado de esa clase. Representa relaciones **es un**.
  - **Herencia privada ::** Lo público y protegido en la clase base es privado en la clase derivada. Representa relaciones de **composición** o **agregación**.

Herencia múltiple de clases.

- Sirve para representar que una nueva clase tiene cierto tipo de relación con otras clases de forma simultánea.
- En **C++** basta con separar por comas las clases base de las que heredamos:

```
class Derived : public Base1, protected Base2, private Base3 {};
```

- Ya sabemos que la herencia múltiple de clases puede plantearnos el problema de la herencia repetida.
- Para resolverlo podemos (en ocasiones) transformar ciertas clases en *interfaces*, o emplear la *herencia virtual*:

```
class Storable {...};
class Transmitter: public virtual Storable {...};
class Receiver: public virtual Storable {...};
class Radio: public Transmitter, public Receiver {...};
```

Reutilización de métodos en clases derivadas.

- Un método heredado en una clase derivada se puede:
  - quedar como se hereda:

```
class B : {
public:
   virtual void m () {...}
};
class D : public B {...};
D d;
d.m();                          // B::m
```

- reescribir completamente:

```
class B : {
```

```
public:
  void m () { ... };
  // C++11 admite override, p.e.
  void m () { ... } override;
...
};
D d;
d.m();                          // D::m
```

- Para más información sobre override consulta su documentación.
- Un método heredado en una clase derivada se puede:
  - *refinar*:

```
class B : {
public:
   virtual void m () {...}
};
class D : public B {
public:
   void m () {
      ...;
      B::m ();                  // Heredado
      ...;
   };
...
};
D d;
d.m();                          // D::m
```

Constructores y destructor en clases derivadas.

- En cualquier constructor de una clase derivada podemos invocar el constructor que queramos de sus clases base.
- Si no lo hacemos, entonces se invoca automáticamente el constructor por defecto. Recuerda que si no lo creamos y nuestra clase no tiene declarado de forma explícita ningún

- La forma de invocar un constructor de una clase base desde uno de una clase derivada es:

```
class B : {
public:
  B (int b) {...}
};
class D : public B {
public:
   D (int n, char c) : B(n), _c(c) , ac(n) { ... }; // Fase de iniciacion.
private:
   char _c;
   AnotherClass ac;
};
```

- **Importante:** A los constructores y destructor les afecta la zona de visibilidad de la clase bajo la cual se declaran! Tenlo en cuenta.
- **Importante:** El destructor de una clase base debería tener enlace dinámico: `virtual ~T()` . Piensa por qué puede ser apropiado hacerlo.
- Los constructores de clases derivadas se invocan en el orden de la herencia, primero los de clases base luego los de objetos contenidos por valor y finalmente el de la clase derivada.
- El orden de invocación de destructores es justo a la inversa.
- Un destructor no se suele invocar directamente, pero si es necesario se puede hacer.

Composición y agregación.

- Aunque son dos relaciones similares, en el fondo son diferentes.

Composición.

- El objeto contenedor es *dueño* del ámbito de vida del objeto contenido:

```
class Robot {
```

```
};

Robot* r2d2 = new Robot;        // Creamos a r2d2
...
delete r2d2;                    // 'mySensor' desaparece con r2d2
```

Agregación.

- El objeto contenedor sólo tiene una referencia al objeto contenido, no controla su ámbito de vida:

```
class Robot {
public:
   Robot () { aSensor = nullptr; }
   void setSensor (Sensor* s) { aSensor = s; }
...
private:
   Sensor* aSensor;             // 'aSensor' es una referencia
};

Robot* c3po = new Robot;
Sensor lightSensor;

c3po.setSensor (&lightSensor);
...
delete c3po;
...
{
   Robot anotherRobot;
   anotherRobot.setSensor (&lightSensor);
   ...
}

if (lightSensor.isLightOn()) ...
```

Relaciones de uso.

- Estos otros objetos pueden estar accesibles como:
  - variables globales:

```
TempSensor aTempSensor;
...
class Robot {
   void doSomeThing () {
      float temp = aTempSensor.readTemp();
   }
}
```

- Estos otros objetos pueden estar accesibles como:
  - variables locales:

```
class Robot {
   void doSomeThing () {
      TempSensor aTempSensor;
      float temp = aTempSensor.readTemp();
   }
}
```

  - parámetros de un método:

```
class Robot {
   void doSomeThing (TempSensor aTempSensor) {
      float temp = aTempSensor.readTemp();
   }
}
```

Polimorfismo.

- Es un término *nuevo* aplicado a la tarea de programar que aparece con la relación de herencia que modela la relación **es un**.

```
class Shape {                          |   class Circle : public Shape {
public:                                |   public:
   virtual float area () = 0;          |      virtual float area () {...}
   virtual void draw () = 0;           |      virtual void draw () {}
   ...                                 |      ...
}                                      |   }

class Rectangle : public Shape {       |   class Triangle : public Shape {
public:                                |   public:
   virtual float area () {...}         |      virtual float area () {...}
   virtual void draw () {}             |      virtual void draw () {}
   ...                                 |      ...
}                                      |   }

...
Shape* aos[MAXSHAPES];
for (int i = 0; i < MAXSHAPES; i++) {
  int st = rand(1..4);          // numero aleatorio: 1=circulo,
                                // 2=rectangulo, 3=triangulo
  if (st == 1) aos[i] = new Circle;
  if (st == 2) aos[i] = new Rectangle;
  if (st == 3) aos[i] = new Triangle;
}
...
for (int i = 0; i < MAXSHAPES; i++) aos[i]->draw(); // No switch was harmed here!
```

- En este vídeo puedes ver el ejemplo de las figuras geométricas de forma práctica.

Principio de sustitución de Liskov.

- Principio definido por Bárbara Liskov en 1994.

> Dados dos tipos **S** y **T**, si **S** es un subtipo de **T** (p.e. por *herencia*) entonces en un programa que haga uso de ambos, cualquier *objeto* de tipo **T** puede ser sustituído por

la sintáctica.

Enlace dinámico.

- Algunos autores también lo llaman **enlace tardío** (*late binding*), y en algunas traducciones podeis encontrar referencias a él como *despacho dinámico*.
- De temas anteriores ya sabemos lo que esto implica: La elección del código a ejecutar (*método*) en respuesta a un mensaje enviado a un objeto se hace en tiempo de ejecución y no de compilación.
- Solemos encontrarnos con esta situación cuando tenemos clases relacionadas mediante herencia (*jerarquía de clases*) las cuales proporcionan implementaciones diferentes para el mismo mensaje.
- En **C++** los métodos con enlace dinámico son aquellos que van anotados con la palabra reservada `virtual`. De lo contrario el enlace por defecto de métodos en **C++** es estático (*tiempo de compilación*), esto es así por razones de eficiencia del código en tiempo de ejecución.

Enlace dinámico de métodos en C++.

Para que en **C++** un método tenga enlace dinámico tenemos que tener en cuenta lo siguiente:

- El método en la clase base debe ser declarado `virtual`, y además...
- Debe tener *exactamente la misma signatura* en la clase derivada, con una excepción, se permiten *tipos covariantes* en el tipo de resultado del método en clases derivadas.
- Dados dos tipos **B** y **D**, se consideran tipos covariantes si **D** deriva directa o indirectamente de **B**, por tanto los métodos virtuales definidos en clases derivadas pueden tener un tipo de resultado distinto del método en la clase base si este tipo es derivado del tipo de dato que devuelve el método en la clase base.

Un ejemplo de enlace dinámico y covarianza de tipos en C++.

```
  virtual Shape* clone () = 0; // Fíjate en la clase derivada...
};

class Line : public Shape {
public:
  Line () { std::cout << "Line const.\n";}
  ~Line () { std::cout << "Line Dest.\n";}

  Line* clone () { return new Line; } // Line deriva de Shape!
};
```

¿Es necesario el modificador virtual en los métodos de clases derivadas?

- No, no lo es. Lo podemos ver en el ejemplo anterior.

  El método `Line::clone()` no lo tiene. El compilador ya lo sabe.

- Además **C++11** introduce el modificador `override` (es opcional) y nos permite detectar en tiempo de compilación errores relacionados con el *enlace dinámico*, fíjate:

  ```
  class A {
  public:
      virtual void foo();         // Enlace dinámico
      void bar();
  };

  class B : public A {
  public:
      void foo() const override; // Error: B::foo does not override A::foo
                                 // (signature mismatch)
      void foo() override;       // OK: B::foo overrides A::foo
      void bar() override;       // Error: A::bar is not virtual
  };
  ```

- La mayoría de compiladores lo hacen de manera similar...
- Mediante una **tabla de saltos**, normalmente llamada *vtable*, *virtual method table (VMT)*, *virtual function table (vftable)*, *virtual call table* o también *dispatch table*.
- Algunos compiladores colocan esta tabla al principio de la memoria del objeto, otros al final...el caso es que esto nos puede dar problemas si intentamos enlazar código objeto generado con un compilador con el generado por otro.

C++, enlace dinámico y destructores.

- El destructor de una clase es un método más...
- Que puede estar *especializado* en clases derivadas.
- Es por eso que en la *clase base* debería estar declarado como `virtual` ...
- De lo contrario si destruimos un objeto de una clase derivada mediante un puntero a la clase base, sólo se invocará el destructor de la clase base y no el de la derivada.

Enlace dinámico de métodos y otros LOO.

- El caso de **C++** es uno de los pocos LOO que han optado por el *enlace estático de métodos por defecto*.
- En general, los LOO eligen *por defecto el enlace dinámico* de métodos (*Java*, *C#*, *D*, *SmallTalk*, etc...).
- En estos otros LOO tenemos que indicar explícitamente que queremos que un método tenga enlace estático para no incurrir en la penalización que ya conocemos respecto a la doble indirección.

  Casi todos ellos suelen emplear el modificador **final**:

  ```
  // Java
  ```

- Esto, además, implica que `Shape.store()` no puede ser redefinido en clases derivadas.

¡Un momento! C++11 ... ¡tiene modificador *final* para métodos y clases!

- Si, lo tiene.
- Símplemente dice que una función virtual no puede ser redefinida en una clase derivada o que no podemos derivar de una clase, mira:

```
struct A {
    virtual void foo() final;   // A::foo is final
    void bar() final;           // Error: non-virtual
                                // function cannot be final
};

struct B final : A {            // struct B is final
    void foo();                 // Error: foo cannot be overridden
                                // as it's final in A
};

struct C : B {                  // Error: B is final
};
```

UML.

- En el paradigma de *POO* existen formas de representar la estructura de clases y relaciones entre objetos de un diseño de manera independiente de la sintáxis concreta de un lenguaje de programación.
- Se conocen como **lenguajes de modelado**. El más empleado hoy en día es **UML**: **U** nified **M** odeling **L** anguage.
- Estos lenguajes nos permiten representar nuestro diseño orientado a objetos de manera gráfica, fácil de entender y que permite cierto tipo de automatizaciones.

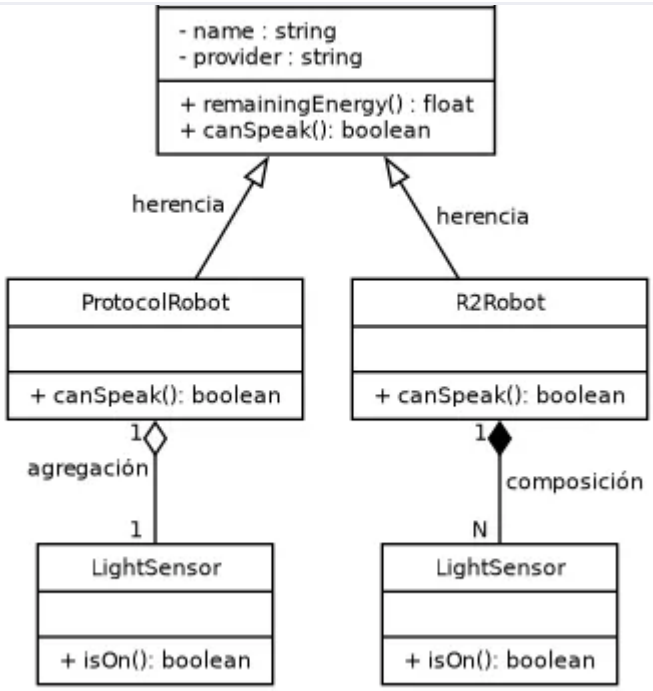

Figura 1: Herencia, composición y agregación.

Aclaraciones.

- **Este contenido no es la bibliografía completa de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.