Tema 4: El paradigma orientado a objetos.

Presentación.

- En este tema vamos a tratar de aclarar el significado del término *Orientado a objetos*.
- En realidad es un *paradigma* que busca dos beneficios muy concretos aplicados al *software* que creamos:
    - Reusabilidad
    - Extensibilidad
- Aunque nosotros nos centraremos en la parte de programación, está también muy ligado a las fases de *análisis* y *diseño*.
- Nos referiremos al término *Programación Orientada a Objetos* por su abreviatura en castellano: **POO** - **OOP** en inglés -.

Historia.

- Los distintos elementos individuales que posteriormente conformaron lo que se conoce como **POO** aparecen con el desarrollo del lenguaje Simula-67.
- Posteriormente se amplia con el desarrollo por parte de Xerox del lenguaje/entorno de computación SmallTalk junto con un hardware novedoso: mira este vídeo.
- Y estos otros donde se explica más detalladamente el funcionamiento de este *novedoso* interfaz de usuario.
- La idea clave detrás de la **POO** es la de poder simular en un computador de manera sencilla modelos de la realidad.
- Para ello usamos en nuestro programa los *mismos términos* que empleamos al describir la realidad.

- Herencia.
- Enlace dinámico, etc...

Elementos básicos de la POO.

Clases.

- Son la descripción de uno o más objetos en base a una serie de atributos y servicios.
- A estos atributos se les llama también 'variables de instancia' y 'variables de clase', mientras que a los servicios se les llama 'métodos'.
- Una clase es la 'esencia' del objeto. En ocasiones se describe como un "*conjunto de objetos que comparten una estructura y comportamiento comunes*".
- Una clase debe tener una 'parte' no visible desde el exterior y una parte visible que es la encargada de acceder a esta parte no visible. En la parte no visible se suelen situar las variables de instancia y/o clase, mientras que en la visible se colocan los métodos de instancia y/o clase.

Objetos.

- Los objetos en el mundo real son aquellas 'cosas'...
  - Que son tangibles y/o visibles.
  - Que pueden ser comprendidas mentalmente.
  - A las que va dirigido el pensamiento o la acción.
- Mientras que cuando los vemos desde el prisma del desarrollo de software, los objetos son:
  - Todo aquello que modela algo real y por tanto ocupa un espacio y un tiempo.
  - Aquellas entidades reales o abstractas con un papel bien definido en el dominio del problema.
  - Toda entidad que en nuestro diseño de una aplicación presenta un **estado**, un **comportamiento** y una **identidad** propios.

igualdad.

- El **estado** de un objeto representa todas las propiedades, normalmente estáticas, del objeto además de los valores actuales, normalmente dinámicos, de cada una de estas propiedades.
- El **comportamiento** de un objeto es el modo en que éste actúa y reacciona (ante *mensajes* recibidos), hablando en términos de cambios en su estado y envío de mensajes a otros objetos.
- Los **mensajes** son las acciones que un objeto realiza sobre otro. A la manera concreta que un objeto responde a un mensaje enviado por otro, se le llama **método**.

Tipos de operaciones sobre un objeto.

- Modificadoras (*setters*)
- Selectoras (*getters*)
- Iteradoras
- Constructoras
- Destructoras
- Estas operaciones pueden formar parte de una clase o estar fuera de cualquiera de ellas.
- Si una de ellas pertenece a una *clase* se le llama **método**, mientras que si no lo hace se le llama simplemente *Función* o también **subprograma o función libre**.
- Al conjunto de métodos y 'subprogramas libres' asociados con un objeto se le llama protocolo.

¿Qué es un Lenguaje Orientado a Objetos?

Aquel que dispone de las siguientes características:

- Encapsulación

Herencia.

- Mecanismo que permite expresar la similitud entre clases.
- ¿Cómo?: Podemos crear nuevas clases a partir de otra u otras ya existentes.
- A la clase nueva creada se le llama **clase derivada** y a la clase de la cual *heredamos* se le llama **clase base**. También se les suele llamar *subclase* y *superclase* respectivamente.
- De este modo incorporamos la estructura y el comportamiento de la clase *pre-existente* a la nueva que estamos creando.
- Dicho de otro modo: la clase derivada 'comparte' las variables de clase y de instancia, así como los métodos de clase y de instancia de su(s) superclase(s).
- La herencia reduce el número de cosas que hemos de 'decir' sobre una nueva clase que creamos si tenemos la precaución de hacer que herede de una clase parecida a ella.
- Las variables declaradas en una clase que se *duplican* o son propias de cada objeto creado de esa clase se llaman **variables de instancia**.
- Las variables declaradas en una clase que se *comparten* por todos los objetos de una misma clase se llaman **variables de clase**.
- Los métodos declarados en una clase que pueden acceder a las *variables de instancia* de un objeto se llaman **métodos de instancia**.
- Los métodos declarados en una clase que pueden acceder a las *variables de clase* de una clase se llaman **métodos de clase**.
- Tendremos **herencia simple** si heredamos sólo de una clase y **herencia múltiple** si heredamos de más de una.
- La *herencia múltiple* puede plantear problemas como el de la *herencia repetida*. **C++** es uno de los pocos LOO que soporta herencia múltiple de clases.

métodos son *abstractos*.

- La relación de herencia se debe utilizar (junto con otras) para reflejar relaciones entre objetos del *mundo real* de la manera más fiel posible.
- Los tipos de relaciones más habituales que podemos encontrar entre objetos en el *mundo real* son:
- **Es un** (*IsA*) : *Un robot **es un** autómata*.
- **Tiene un** (*HasA*) : *Un robot **tiene un** sensor*.
- **Usa un** (*Uses*) : *Un robot **usa un** cargador de baterías*.
- La relación **Es un** introduce el principio de sustitución.
- La relación **Tiene un** introduce el concepto de composición entre objetos.
- Los métodos heredados se pueden usar directamente o también se pueden reescribir en la clase derivada.
- En este caso podemos hacerlo de dos modos:
    - **Reemplazándolos** completamente.
    - **Refinándolos**, para ello en algún punto del método reescrito en la clase derivada invocamos al método heredado de la clase base.
- Es posible que a un método de una clase no tenga sentido proporcionarle una implementación. En este caso se puede dejar sin ella, pero *convenientemente anotado*, y se le llama **método abstracto**.
- Cuando una *clase* tiene al menos un **método abstracto** automáticamente pasa a ser una **clase abstracta**.
- Una **clase abstracta** no puede tener instancias.

Enlace Dinámico.

> Es el instante de tiempo en el que se determina o identifica el trozo de código, p.e. una función que ha de ser llamada tras el envío de un mensaje, o el significado de una 'construcción' especial en memoria, p.e. el tipo exacto de una variable o un dato.

- En POO existen al menos dos atributos que se ven directamente afectados por el *tiempo de enlace*:
  - El **tipo** del dato al que se refiere un identificador.
  - El **método** con el que se responde a un mensaje.
- Para hablar sobre el tipo de un identificador debemos distinguir entre:
  - **Identificador:** Es sólo un nombre.
  - **Valor:** El contenido de la memoria asociada a un identificador.
  - **Tipo:** Depende del lenguaje con el que trabajemos. En unos casos irá asociado a una variable y significará una cosa, mientras que en otros irá asociado a un valor y significará otra.
- Los lenguajes en los que el tipo de toda expresión se conoce en tiempo de compilación se llaman **fuertemente tipados** (*LFT*).
- En los *LFT*, p.e. el concepto de **tipo** va ligado al de variable, es decir, los tipos se asocian con un identificador mediante sentencias explícitas: `int n;`
- En los *LFT* el *tipo* sirve para dar idea de los posibles valores que puede tomar una variable.
- **C++-11** permite declarar variables de tipo `auto`, en las cuales el compilador *infiere* el tipo a partir de la expresión de inicialización de la variable: `auto n = 3;`
- Por completitud, los lenguajes donde el tipo no se asocia a un identificador sino a un valor, se llaman *débilmente tipados* (*LDT*).
- Ejemplos de lenguajes
  - **débilmente tipados**: `python`, `smalltalk`, `perl`, etc...
  - **fuertemente tipados**: `C`, `C++`, `Java`, `C#`, `D`, etc...
- Si retomamos el concepto del principio de sustitución y una declaración de clases con herencia como esta:

```
class Robot : public Automaton {...}; // Es Un
                                      // Un robot es un automata.
```

- El principio de sustitución nos dice que en cualquier lugar de nuestro código donde podamos usar un dato de clase o tipo `Automata`, podremos usar uno de clase `Robot`, p.e.:

```
Robot* r = new Robot;
r->compute ();
Automaton* aa[100];
aa[0] = r;
aa[1] = new Automaton;
```

- La decisión del tipo de objeto con el que se va a trabajar no se puede saber hasta el mismo instante de la ejecución.
- A esto es a lo que llamamos *enlace dinámico*.
- Pero el *enlace dinámico* no afecta solo a la decisión sobre el tipo real del objeto al que se refiere una variable.
- Tiene consecuencias, p.e., con el código a invocar (*método*) en respuesta a un mensaje recibido.

```
class Automaton {
  public:
    virtual void compute () {...} // compute tiene ahora enlace dinámico (virtual)
};
class Robot : public Automaton {
    void compute () {...}         // Se redefine en clases derivadas
                                  // no es necesario virtual, ya se sabe.
};
class ChessPlayer : public Automaton {
    void compute () {...}
};
```

- ¿Que crees que pasará con este código?

```
for (auto ai = 0; ai < 10; ai++) aa[ai]->compute();
```

- ¿Y si es el usuario el que elige el tipo de automáta en tiempo de ejecución desde un menú?
- ¿Si tan bueno es el *enlace dinámico* por qué no está habilitado por defecto en C++?
- De hecho algunos lenguajes orientados a objetos sí que lo hacen: Smalltalk, Java, C#, D, etc...
- La respuesta está en la eficiencia en tiempo de ejecución. Piensa a qué puede deberse esto.

Paso de mensajes.

- Cuando trabajamos con un objeto instancia de una clase que tiene una parte visible y otra no-visible desde el exterior de la misma, las funciones *normales* (externas a una clase) no tienen capacidad para acceder a la parte no-visible de la clase.
- Sólo las operaciones definidas dentro de la *clase* pueden acceder a las variables de instancia o de clase.
- Por tanto debemos invocar una de estas operaciones definidas en la clase sobre un determinado objeto: `obj.operation()` (*envío del mensaje*).
- **C++** permite levantar esta restricción mediante las llamadas *funciones amigas*. En otros lenguajes se puede hacer de otras formas.
- Tradicionalmente en la POO se denomina a esta acción *enviar un* ***mensaje*** a un objeto, p.e.: `robot.avanzarLineaRecta(20);`
- Del mismo modo, el código que ejecuta el objeto en respuesta a este mensaje se le llama ***método***.
- Un *mensaje* no es exactamente igual a una *función libre*:
  - Siempre tendrá un parámetro más. Normalmente es un parámetro *oculto*.

- En C++ este parámetro oculto se llama **this**.

Encapsulación.

- Es la característica que nos permite agrupar bajo una misma entidad los datos y las funciones que trabajan con esos datos.
- Es un mecanismo más de *abstracción*.
  Hasta ahora conoceis dos:
  1. *División en funciones*
  2. *Tipos Abstractos de Datos*.
- Permite establecer zonas de visibilidad desde el exterior al interior de esta entidad.
- ¿Qué ganamos al unir bajo una misma entidad datos y funciones?:
  - La independencia del código que usa esta *entidad* de los cambios que pueda haber en su representación interna.
- Normalmente los *datos* suelen estar en la parte *no-visible* mientras que las funciones o métodos están en la parte *visible*. De este modo reforzamos el punto anterior.
- A esta parte *visible* se le conoce como el **interfaz** de la clase o también *parte pública*. A la parte *no-visible* se le llama **implementación** o también *parte privada*.
- En los LOO la *entidad* que aporta la capacidad de *encapsulación* es la **clase**.
- En los LOO a una variable cuyo tipo sea una *clase* se le llama **objeto**. También nos podemos referir a ella como una **instancia de la clase** o simplemente **instancia**.
- A las funciones definidas dentro de una clase se les llama **métodos**.
- Al igual que con las *variables*, existen **métodos de clase** y **métodos de instancia**.
- Un *método de instancia* definido en una clase se invoca a través del operador "\(\bullet\)", o del operador "**\(\to\)**" (si la variable que representa al objeto es un puntero):
  1. `object.method (parameters)`

- En el caso de un *método de clase*, si, p.e., el identificador `DashBoard` representa un nombre de una clase, usar un método de clase con ese identificador lo haríamos así:

  ```
  DashBoard::classMethod (parameters)
  ```

C++ como lenguaje orientado a objetos.

- **C++** es un *LOO*, es decir, soporta:
  - Herencia
  - Enlace Dinámico
  - Paso de Mensajes
  - Encapsulación
- Entre los temas del 5 al 9 iremos estudiando en detalle estos conceptos y algún que otro más (*excepciones, genericidad*), así como de que manera se incorporan en **C++**:
  - **Tema 5**: *Clases y objetos*.
  - **Tema 7**: *Herencia, polimorfismo y enlace dinámico*.
  - **Tema 8**: *Genericidad*.
  - **Tema 9**: *Excepciones*.

Ejemplo de enlace dinámico en C++. Comparativa con C.

Código C++.

```
// Copyright (C) 2020-2023  Programacion-II

// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.

// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
```

```cpp
#include <iostream>
#include <cstdint>

class FiguraGeometrica {
public:
  FiguraGeometrica() {
    // std::cout << "Hola, soy una FG tambien\n";

    inc_count();
  }

  virtual ~FiguraGeometrica() { std::cout << "FiguraGeometrica::DESTRUCTOR\n"; }

  // Prueba a usar esta definición de dibujar en lugar de la que hay
  // mas adelante. Si se produce algún error de compilación trata de
  // resolverlo.
  // virtual void dibujar() = 0;

  // FiguraGeometrica::dibujar tiene enlace dinámico
  virtual void dibujar() { std::cout << "FiguraGeometrica::dibujar\n"; };

  static uint32_t get_count() { return count; }
  static void inc_count() { count++; }

private:
  //static uint32_t count;  // Probar a inicializar aqui.

  // O de este otro mode en C++17 o superior
  // inline static uint32_t count = 0;
};

uint32_t FiguraGeometrica::count = 0;

using FiguraGeometricaPtr = FiguraGeometrica*;

class Circulo : public FiguraGeometrica {
public:
  Circulo() { std::cout << "Hola, soy un circulo\n"; }
```

```
};

class Rectangulo : public FiguraGeometrica {
public:
  Rectangulo() { std::cout << "Hola, soy un rectangulo\n"; };

  virtual ~Rectangulo() { std::cout << "Rectangulo::DESTRUCTOR\n"; }

  void dibujar() { std::cout << "Rectangulo::dibujar\n"; }
};

// Comprueba lo sencillo que es añadir una nueva clase al
// diseño. Comparalo con lo que tendrias que hacer para añadir una
// nueva clase en la versión de C no orientada a objetos.

// class Triangulo : public FiguraGeometrica {
//     public:
//         Triangulo() {
//             std::cout << "Hola, soy un triangulo\n";
//         };

//         virtual ~Triangulo() {
//             std::cout << "Triangulo::DESTRUCTOR\n";
//         }

//         void dibujar() {
//             std::cout << "Triangulo::dibujar\n";
//         }
// };

int main () {

  FiguraGeometricaPtr vfg[5] = {nullptr,nullptr,nullptr,nullptr,nullptr};

  vfg[0] = new Circulo;
  vfg[1] = new Circulo;
  vfg[2] = new Rectangulo;
  vfg[3] = new Rectangulo;
```

```
  for (int f = 0; f < 5; f++) {
    std::cout << "vfg[" << f << "]: ";
    vfg[f]->dibujar();
  }

  for (int f = 0; f < 5; f++) {
    delete vfg[f];
  }

  std::cout << "total de FG creadas: " << FiguraGeometrica::get_count() << '\n';

  return 0;
}
```

Código C.

```
/*
 * Copyright (C) 2020-2022  Programacion-II
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <stdio.h>
#include <stdlib.h>
```

```
 */

enum TipoFigura { FG, CIRCULO, RECTANGULO };

typedef struct FiguraGeometrica {
    enum TipoFigura t;
} FiguraGeometrica;

typedef struct Circulo {
    enum TipoFigura t;
} Circulo;

typedef struct Rectangulo {
    enum TipoFigura t;
} Rectangulo;

typedef FiguraGeometrica* FiguraGeometricaPtr;
typedef Circulo* CirculoPtr;
typedef Rectangulo* RectanguloPtr;

void FiguraGeometrica_dibujar(FiguraGeometricaPtr this) {
    printf("FiguraGeometrica::dibujar\n");
}

void Circulo_dibujar(CirculoPtr this) {
    printf("Circulo::dibujar\n");
}

void Rectangulo_dibujar(RectanguloPtr this) {
    printf("Rectangulo::dibujar\n");
}

int main(int argc, char *argv[]) {
    FiguraGeometricaPtr vfg[5] = {NULL, NULL, NULL, NULL, NULL};

    vfg[0] = malloc (sizeof (Circulo));
    vfg[0]->t = CIRCULO;
```

```
    vfg[2]->t = RECTANGULO;

    vfg[3] = malloc (sizeof (Rectangulo));
    vfg[3]->t = RECTANGULO;

    vfg[4] = malloc (sizeof (Circulo));
    vfg[4]->t = CIRCULO;

    for (int f = 0; f < 5; f++) {
        switch (vfg[f]->t) {
            case FG:
                FiguraGeometrica_dibujar(vfg[f]);
              break;
            case CIRCULO:
                Circulo_dibujar((CirculoPtr) vfg[f]);
                break;
            case RECTANGULO:
                Rectangulo_dibujar((RectanguloPtr) vfg[f]);
                break;
        }
    }

    for (int f = 0; f < 5; f++) {
        free(vfg[f]);
    }

    return 0;
}
```

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.

← Tema 3: Tipos Abstractos de Datos: Árboles, Grafos.

Tema 5: Clases y objetos. Espacios de nombres. →