Universidad Complutense de Madrid
Facultad de Informática
Grado en Ingeniería Informática, Grupo A

Tecnología de la Programación

Introducción a la Programación Orientada a Objetos

Simon Pickin,
Alberto Díaz, Puri Arenas

Introducción a la programación orientada a objetos

1. Paradigmas de programación
   - Programación imperativa
     - Programación modular
     - Programación orientada a objetos (OO)
2. Evolución de los enfoques de programación imperativa
3. Historia de la programación orientada a objetos
4. Un primer vistazo a la OO
   - Fecha en C++
   - Fecha en Java

Paradigmas de programación

- Un paradigma o modelo de programación es
  - una forma de pensar acerca del proceso de
    - descomposición de problemas
    - desarrollo de soluciones
- Los lenguajes de programación pueden soportar
  - un solo paradigma
  - múltiples paradigmas
- Paradigmas de programación clásicos
  - La programación imperativa
  - La programación lógica
  - La programación funcional

Características de la programación imperativa

- El programa consta de un conjunto de instrucciones paso a paso
  - que indican como solucionar el problema
- Las instrucciones hacen uso de *variables de programa*
  - El valor de una variable de programa se obtiene mediante el uso de su nombre
  - Se da un valor a una variable de programa mediante una sentencia de asignación
  - El valor de una variable de programa puede cambiar varias veces durante una ejecución
  - El estado del programa en un punto dado de una ejecución se define como el conjunto de valores de sus variables de programa en ese punto
- Una instrucción ejecutada por el ordenador
  - depende del estado del programa antes de la ejecución de la instrucción
  - puede dejar al programa en otro estado después de la ejecución de la instrucción
    - mediante la modificación de las variables de programa

Características de la programación imperativa

- La programación imperativa es una traducción casi directa de las capacidades hardware de las máquinas (*)
  - La memoria representa el estado del programa
  - Las instrucciones máquina son las operaciones.
- Se puede dividir la programación imperativa en:
  - programación modular
  - programación orientada a objetos

* Aún más antes de finales de los 60 cuando hubo un amplio uso de la sentencia `goto`, que corresponde directamente a la instrucción jump del ensamblador

- El famoso (y, en su época, polémico) artículo de Djikstra "*Goto statement considered harmful*" (1968) marcó el alza de la "Programación Estructural", que abogaba por la prohibición de la sentencia `goto` en favor del uso exclusivo de sentencias de *secuencia*, de *selección* y de *iteración*, agrupadas en subrutinas de entrada única y salida única y luego en modulos, aunque muchos lenguajes mantenían un tipo de `goto` para salir de bucles.

Evolución de la programación imperativa

- Código máquina
- Lenguaje ensamblador:
  - Pequeña abstracción
- Primeros lenguajes de alto nivel:
  - Tipos asociado a variables
- Programación procedimental:
  - Subprogramas
- Programación modular:
  - Módulos que agrupan procedimientos y funciones
- Programación con TAD:
  - Datos + operaciones
- Programación orientada a objetos: Clases y objetos

Características de la programación modular

- Los programas se dividen en subprogramas / subrutinas
  - Funciones: reciben datos como argumentos y devuelven datos como resultado
  - Procedimientos: reciben datos como argumentos pero no devuelven resultado
    - En C, C++ y Java, los procedimientos son funciones que devuelven `void`
- Los datos y los subprogramas puede considerarse de forma separada
  - Las funciones transmiten datos entre ellas
    - Los datos fluyen entre las funciones / los procedimientos
- Los subprogramas se agrupan en módulos
- Se puede usar los módulos para crear tipos abstractos de datos (TAD)
  - Un TAD reúne una estructura de datos y las operaciones que manipulan los valores conformes con esa estructura.
    - *Encapsulación* y *ocultación de información*
  - En la asignatura EDA se trata en profundidad este tema

El uso de la programación modular en el grado en I.I.

- La programación modular es el paradigma utilizado en la asignatura del primer curso FP
  - con el lenguaje C++
- Sin embargo, C++ es un lenguaje multi-paradigma
  - También se puede utilizar para realizar programación orientada a objetos

Programación orientada a objetos (POO)

- Surgen dos nuevos conceptos:
  - las clases y los objetos.
- Los objetos se declaran por medio de clases
- Una clase es como una plantilla para crear objetos
  - Una clase define un tipo
    - Se podría decir que una clase es a un objeto lo que un tipo es al valor de una variable de programa en un lenguaje de programación imperativa
  - Un objeto es un ejemplar (o "instancia") de una clase
    - Cada ejemplar tiene su propia identidad, es decir, se distingue de otros ejemplares

Programación orientada a objetos (POO)

- En vez de tener por un lado las funciones y procedimientos y por otro los datos
  - ambos se juntan en clases y objetos que se tratan como una unidad
  - proporcionando la *encapsulación* y *ocultación de información* de los TAD
- En OO, se utiliza el término *atributos* para las variables
  - que pertenecen a alguna clase
  - Otros términos usados a veces en vez de "atributos": *campos*, *variables miembro*
- En OO, se utiliza el término *método* para las operaciones
  - que pertenecen a alguna clase
  - Contienen las instrucciones imperativas conocidas (bucles, condicionales, etc.)
  - Los atributos de la clase están visibles para todos los métodos de la clase
    - Pueden verse como parámetros implícitos
  - Otro término usado a veces en vez de "métodos": *funciones miembro*

La ejecución de un programa OO

- Consta de
  - un conjunto de objetos, cada uno de una cierta clase
  - interactuándose mediante llamadas a métodos entre unos y otros, pasándose datos a través de los argumentos de esos métodos
  - Muchos lenguajes OO permiten a los objetos leer y/o escribir los atributos de otros objetos. Sin embargo:
    - esa posibilidad debería evitarse donde sea posible ya que rompe la encapsulación / ocultación de información (*).

(*) Los datos primitivos de un objeto que se leerán/se escribirán por otros objetos deberían almacenarse en *propiedades*, donde una propiedad, en este contexto, es un atributo de un tipo primitivo cuya clase contiene un método para devolver su valor (un método *get*) y posiblemente también un método para poner su valor (un método *set*).

Características principales de la POO

- *Programación imperativa*
- Clases como unidades de *encapsulación* y *ocultación de información*
  - Una clase es un TAD (¡normalmente!)
  - Programación basada en prototipos: encapsulación sin clases
    - p.ej. Javascript
- Clases y objetos; la noción de ID de objeto
  - Se puede distinguir entre distintos ejemplares de la misma clase
- Clases y tipos: una clase define un tipo en el sistema de tipos
- *Herencia*
- *Polimorfismo* (*de subtipo*): solo en lenguajes de tipado estático
  - considerando que subclases definen subtipos
- V*inculación dinámica*

Historia de la POO

- Finales de los años 60: Simula-67 en Noruega
  - Objetos & clases, herencia & subclases, sobreescritura de métodos, recolección de basura
- Años 70: Smalltalk en Xerox PARC, CA, EEUU
  - Reflexión structural y computacional
- Años 80
  - Congresos internacionales sobre OO (p.ej. OOPSLA, ECOOP)
  - Técnicas de ingeniería del software OO
  - Nuevos lenguajes (p.ej. Eiffel) y se extienden otros (C => C++)
- Años 90
  - El OO es omnipresente
    - Sistemas operativos, entornos de desarrollo visual, ..
  - Sun Microsystems (comprado luego por Oracle) diseñan el lenguaje Java
- Siglo 21
  - Muchos lenguajes son OO o incluyen características OO:
    - PHP, LUA, C#, Python, …

Ejemplo 1: declaración de Fecha en C++ sin OO

```
// Fichero Fecha.h
// Definición del tipo de datos

struct Fecha {
    int dia;
    int mes;
    int anyo;
};

// Declaración de funciones relacionados con el tipo Fecha.
// Las implementaciones aparecerían en Fecha.cpp


// Construye una fecha
Fecha construye(int day, int month, int year);

// Dada una fecha la suma el número de días pasado como parámetro
void suma(Fecha &fecha, int numDias);

// Dadas 2 fechas devuelve el número de días que hay entre ellas
int diferencia(const Fecha &fecha1, const Fecha &fecha2);

// Escribe por pantalla la fecha
void escribe(const Fecha &fecha);
```

Ejemplo 1: uso de Fecha en C++ sin OO

```
int main() {

    // f1 y f2 son variables del tipo Fecha
    Fecha f1, f2;

    f1 = construye(12, 10, 1942);
    f2 = construye(11, 11, 1970);

    escribe(f1);
    escribe(f2);
    suma(f1, 3);
    std::cout << diferencia(f1, f2) << std::endl;

    return 0;

}
```


Ejemplo 1: declaración de Fecha en C++ con OO

```
// Fichero Fecha.h
// Definición del tipo de datos

class Fecha {
    public:
        // Declaración de métodos relacionados con el tipo Fecha.
        // Las implementaciones aparecerían en outro ficheiro: Fecha.cpp

        // Constructor de Fecha: método que construye un objeto Fecha
        Fecha (int day, int month, int year);

        void suma(int numDias);
        int diferencia(const Fecha &fecha);
        void escribe();

    private:
        int dia;
        int mes;
        int anyo;
};
```

Ejemplo 1: uso de Fecha en C++ con OO

```
int main() {

        // se crean y se instancian dos variables f1 y f2 de tipo Fecha
        Fecha f1(12, 10, 1492);
        Fecha f2(1, 1, 1970);

        f1.escribe();
        f2.escribe();
        f1.suma(3);
        std::cout << f1.diferencia(f2) << std::endl;

        return 0;
    }
 }
```


Ejemplo 1: declaración de Fecha en Java con OO

```
// Fichero Fecha.java
// Incluye la definición del tipo de datos
// y la implementación de los métodos

public class Fecha {

       public Fecha(int day, int month, int year) { // ... }
       public void suma(int numDias) { // ... }
       public void escribe() { // ... }
       public int diferencia(Fecha fecha) { // ... }

       private int dia;
       private int mes;
       private int anyo;

};
```

Ejemplo 1: uso de Fecha en Java con OO

```
public class Ejemplo {

    public static void main(String [] args) {

        // se crean y se instancian dos variables f1 y f2 de tipo Fecha
        Fecha f1 = new Fecha(12, 10, 1492);
        Fecha f2 = new Fecha(1, 1, 1970);

        f1.escribe();
        f2.escribe();
        f1.suma(3);
        System.out.println(f1.diferencia(f2));

        }
}
```


Ejemplo 2: números racionales

- Consideremos un número racional.
- Podemos representar un número racional como:

```
int num; // numerador del racional
int den; // denominador del racional
```

- Individualmente los datos son enteros
  - Sin embargo juntos componen un racional, con su numerador y denominador

Ejemplo 2: operaciones sobre los racionales en C++

```
// inicia los valores de r.num a n y r.den a d
void crear(Rational &r, int n, int d):

// multiplica el numerador del racional r por n
void productoEscalar(Rational &r, int n)

// calcula la versión irreducible del racional r
void simplificar(Rational &r)

// calcula r1 = r1 * r2
void multiplicar(Rational &r1, const Rational &r2)
```


Ejemplo 2: implementación de Rational en C++

```
// Modulo Rational.h

struct Rational {
    int num;
    int den;
};

void crear(Rational &r, int n, int d){…}
void productoEscalar(Rational &r, int n){…}
void simplificar(Rational &r){…}
void multiplicar(Rational &r1, const Rational &r2){…}
```

Ejemplo 2: uso de la implementación de Rational

```
// módulo Main.h
#include "Rational.h"

...

Rational r;
crear(r,3,4);              // r = 3/4
productoEscalar(r,2);      // r=6/4
simplificar(r);            // r=3/2
cout << r.num << '/' << r.den << '\n';

...
```


Ejemplo 2: otra implementación de Rational en C++

- El programador del tipo Rational decide cambiar su implementación

```
// Modulo Rational.h

typedef int Rational[2]; // usar array en vez de struct

// Y adaptar la implementación de las funciones
void crear(Rational &r, int n, int d){…}
void productoEscalar(Rational &r, int n){…}
void simplificar(Rational &r){…}
void multiplicar(Rational &r1, const Rational &r2){…}
```

Ejemplo 2: uso de la otra implementación de Rational

```
// módulo Main.h
#include "Rational.h"

// ¿Qué ocurre al código que utiliza Rational?
...
Rational r;
crear(r,3,4);              // r = 3/4
productoEscalar(r,2);      // r=6/4
simplificar(r);            // r=3/2
cout << r.num << '/' << r.den << '\n';  // problema
...
```


Ejemplo 2: conclusiones

- ¿Por qué se ha roto la abstracción?
  - Porque hemos utilizado la implementación del tipo racional y no los métodos que ofrece para manipularlo.
- ¿Cómo escribir un racional y mantener la abstracción de datos?
  - O bien introducir funciones para acceder al numerador y denominador del racional.
  - O bien definir un método para escribir un racional.
- ¿Qué nos ofrece la POO?
  - Una manera de encapsular y abstraer:
    - La implementación del tipo de datos es privada.
    - Las funciones y/o procedimientos para manipular el tipo de datos son públicas.

Ejemplo 2: adición de funciones de acceso

```
// añadir funciones de acceso
int getNum(const Rational& r){
   return r.num;  // versión para implementación con struct
}
int getDen(const Rational& r){
   return r.den;  // versión para implementación con struct
}
Rational r;
crear(r,3,4);              // r = 3/4
productoEscalar(r,2);      // r=6/4
simplificar(r);            // r=3/2
cout << getNum(r) << '/' << getDen(r) << '\n';
```


Ejemplo 2: implementación de Rational con OO (Java)

```
public class Rational {
   private int num;
   private int den;

   // en vez de void crear(Rational& r, int n, int d){…}
   public Rational(int n, int d){…}
   // en vez de void productoEscalar(Rational& r, int n){…}
   public void productoEscalar(int n){…}
   // en vez de void simplificar(Rational& r){…}
   void simplificar(){…}
   // en vez de void multiplicar(Rational& r){…}
   void multiplicar(Rational r2){…}

   public int getNum(){…}
   public int getDen(){…}
}
```

Ejemplo 2: construcción de entidades Rational

`Rational&` , tal como usado en argumentos de función, equivale a `this` (que es un argumento implícito de todos los métodos en OO)

```
// Función que crea structs de tipo Rational
void crear(Rational &r, int n, int d){
      r.num = n;
      r.den = d;
}

// Constructor que crea objectos de la clase Rational
public Rational(int n, int d){
      this.num = n;  // uso de "this" no es obligatorio
      this.den = d;  // uso de "this" no es obligatorio
}
```


Resumen: encapsulación y ocultación de información

- Encapsulación
  - El uso de componentes / módulos que agrupan datos y operaciones sobre estos datos.
- Ocultación de Información (a veces incluido en “encapsulación”)
  - Asegurar que los detalles de implementación de un componente / modulo encapsulado no son visibles en código fuera de este componente / modulo.
  - Algunos lenguajes, p.ej. Python, tienen un mecanismo para la encapsulación pero no para la ocultación de información
- Este mecanismo:
  - reduce la fragilidad (las modificaciones no rompen el código fácilmente)
  - limita las interdependencias entre componentes software
  - protege la integridad de los datos