Programación Orientada a Objetos

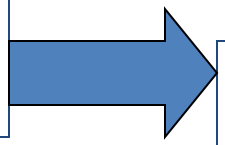
Paradigmas de la Programación 2021
FaMAF-UNC
capítulos 9 y 10 de Mitchell
basado en filminas de Vitaly Shmatikov

objetivos

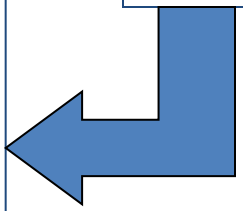
- desarrollo de programas modulares
  - refinamiento incremental
  - interfaz, especificación e implementación
- soporte de los lenguajes para la modularidad
  - abstracción procedural
  - tipos abstractos de datos
  - paquetes y módulos
  - abstracciones genéricas (con parámetros de tipo)

ejemplo de Dijkstra (1969)

```
begin
    print first 1000 primes
end
```

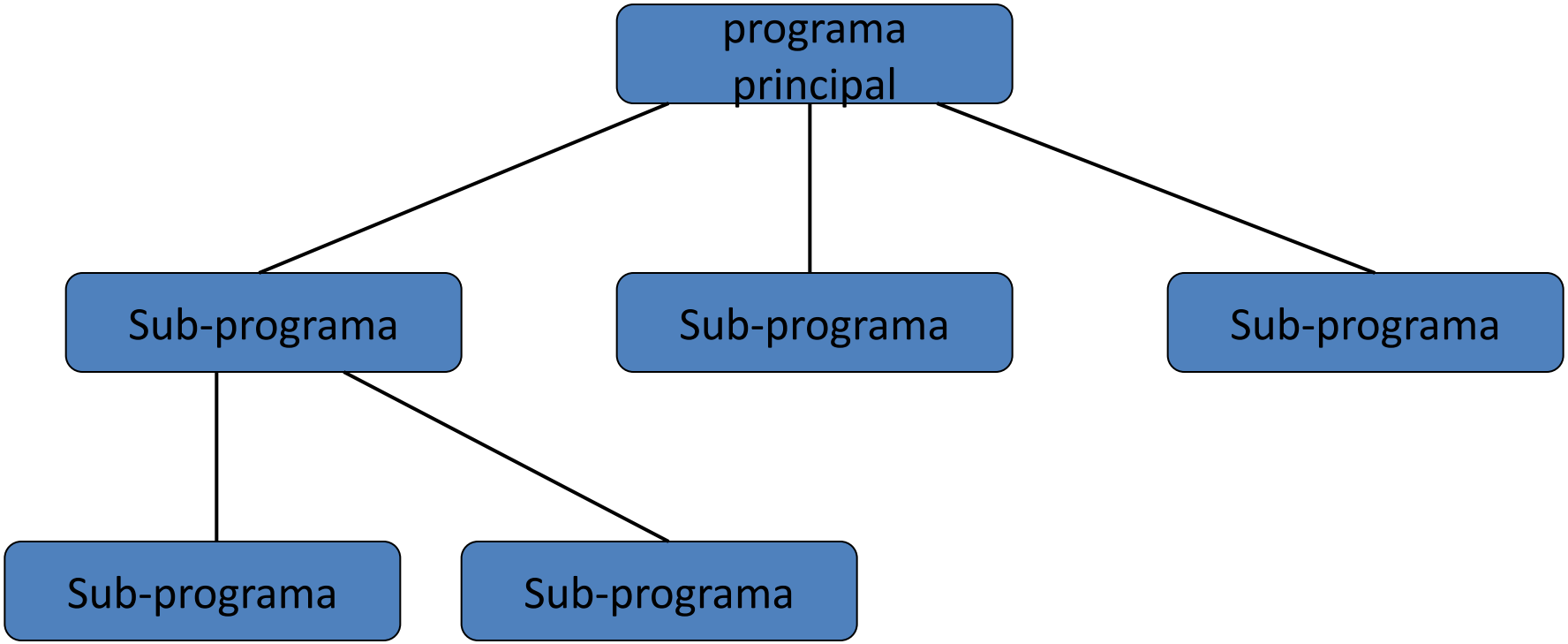


```
begin
    variable table p
    fill table p with first 1000
      primes
    print table p
end
```

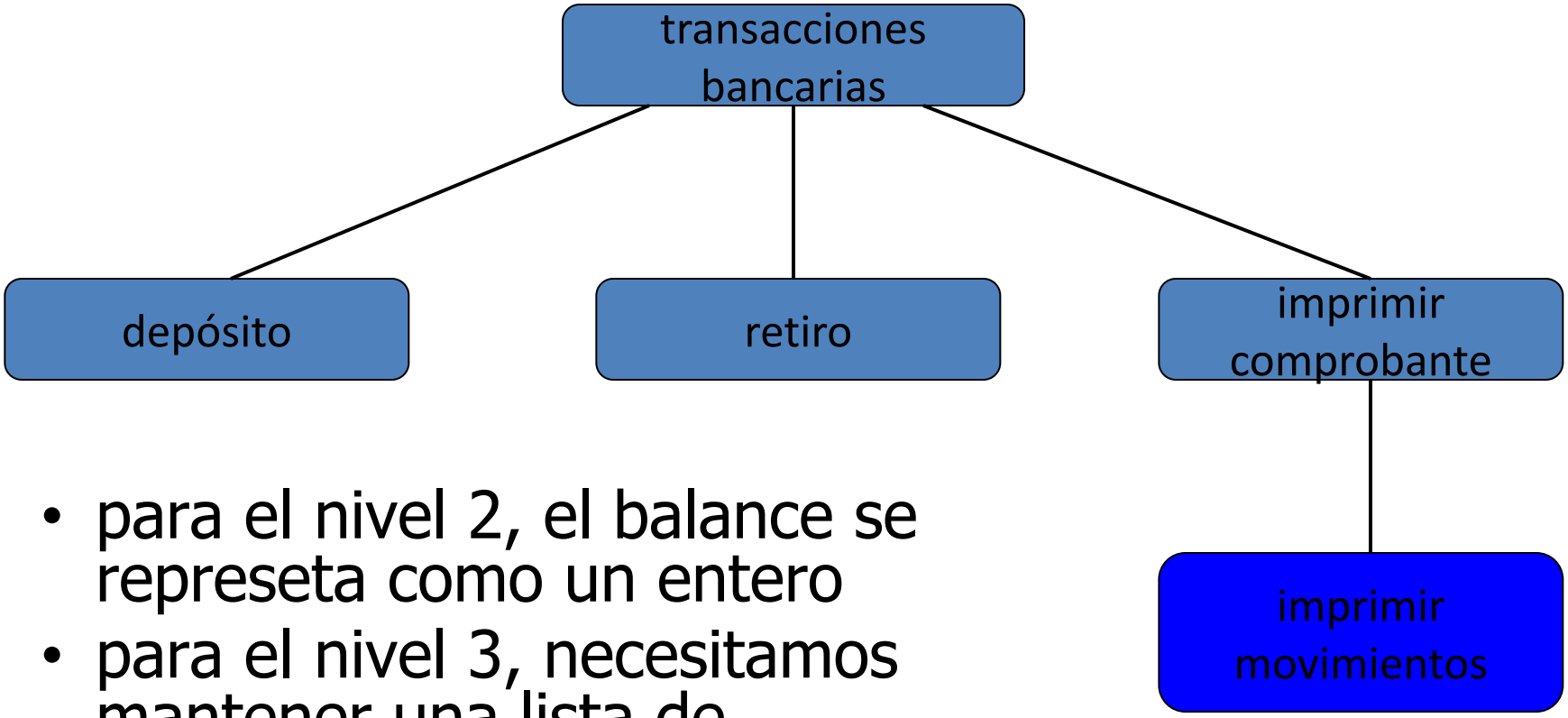


```
begin
    int array p[1:1000]
    make for k from 1 to 1000
      p[k] equal to k-th prime
    print p[k] for k from 1 to 1000
end
```

estructura de un programa (vs. un script)



ejemplo



- para el nivel 2, el balance se represeta como un entero
- para el nivel 3, necesitamos mantener una lista de transacciones pasadas

modularidad: conceptos básicos

modularidad: conceptos básicos

- **componente**
  - **unidad** de programa **con sentido**: función, estructura de datos, módulo,…
- **interfaz**
  - tipos y operaciones definidos dentro de un componente que son **visibles** fuera del componente
- **especificación**
  - **comportamiento** esperado de un componente, expresado como una propiedad **observable** a través de la interfaz
- **implementación**
  - estructuras de datos y funciones **dentro** del componente

ejemplo: componente función

- **componente**
  - función que calcula la raíz cuadrada
- **interfaz**

```
float sqroot (float x)
```

- **especificación**

```
si x>1, entonces sqrt(x)*sqrt(x) ≈ x
```

- **implementación**

```
float sqroot (float x){
   float y = x/2; float step=x/4; int i;
   for (i=0; i<20; i++){if ((y*y)<x) y=y+step; else y=y-step; step
  = step/2;}
   return y;
}
```

ejemplo: tipo de datos

- **componente**
  - cola de prioridad: estructura de datos que devuelve elementos en orden de prioridad descendente
- **interfaz**
  - tipo pq
  - operaciones empty : pq
    insert : elt * pq → pq
    deletemax : pq → elt * pq
- **especificación**
  - Insert añade un elemento al conjunto de elementos guardados
  - Deletemax devuelve el elemento máximo y compone el resto de elementos en una cola de prioridad

ejemplo: tipo de datos

- tres operaciones

```
empty     : pq
insert          : elt * pq → pq
deletemax   : pq →  elt * pq
```

- un algoritmo que usa la cola de prioridad (heap sort)

```
begin
 create empty pq s
 insert each element from array into s
 remove elts in decreasing order and
 place in array
end
```

herramientas de los lenguajes para la abstracción

tipos abstractos de datos (TADs)

- desarrollo de lenguaje de los 1970s
- Idea 1: separar la interfaz de la implementación

  ejemplo:

  los conjuntos tienen las operaciones: `empty, insert, union, is_member?, …`

  los conjuntos se implementan como … `lista enlazada` …
- Idea 2: usar comprobación de tipos para forzar la separación
  - el programa cliente sólo tiene acceso a las operaciones de la interfaz
  - la implementación encapsulada en el constructo TAD

módulos

- construcción general para ocultar información
  - módulos (Modula), paquetes (Ada), estructuras (ML), …
- interfaz:
  - conjunto de nombres y sus tipos
- implementación:
  - declaración para cada entrada en la interfaz
  - declaraciones extra que están ocultas

abstracciones genéricas

parametrizar los módulos por tipos

- implementaciones generales, que se pueden instanciar de muchas formas: la misma implementación para múltiples tipos
- paquetes genéricos en Ada, templates en C++ (especialmente las de la STL – Standard Template Library), functores en ML functors …

templates de C++

- mecanismo de parametrización de tipos

  `template<class T>` ... indica el parámetro de tipo T
  - C++ tiene templates de clase y de función
- se instancian en tiempo de ligado
  - se crea una copia del template generado para cada tipo
  - por qué duplicar código?
    - tamaño de variables locales en el activation record
    - Ligado a las operaciones del tipo instanciado
- ej: función swap (sobrecarga y polimorfismo)

ejemplo de template

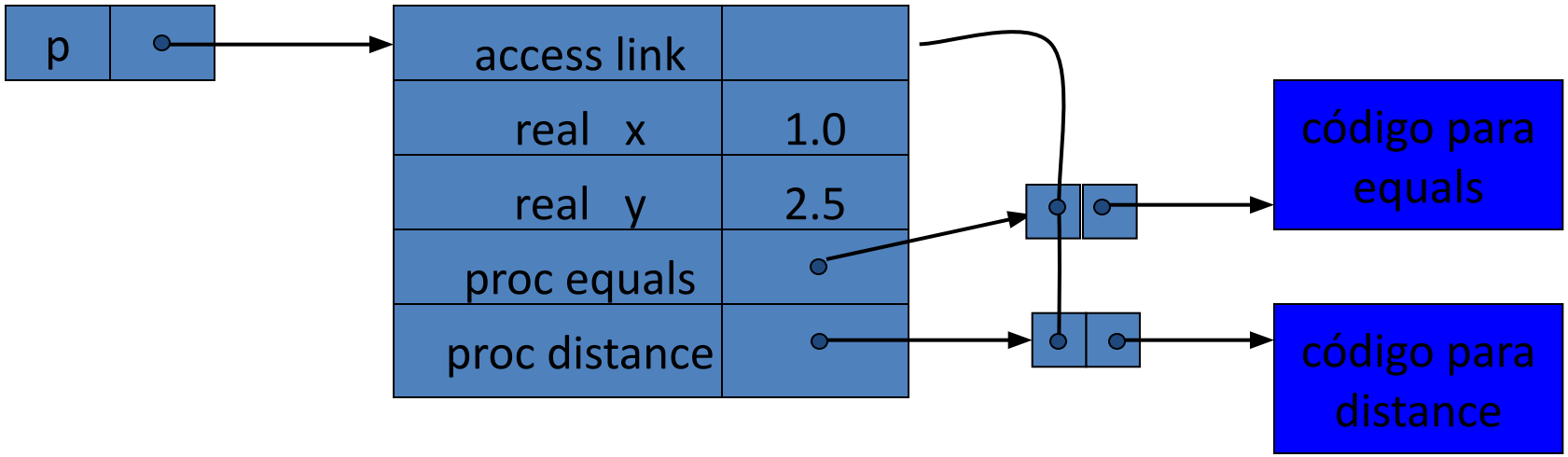
```
template <typename T>
void swap(T& x, T& y){
      T tmp = x;  x=y;  y=tmp;
}
```

diferencia entre ML y C++

- ML
  - *swap* se compila a una función, y el typechecker (comprobador de tipos) determina cómo se puede usar
- C++
  - *swap* se compila a formato linkeable, y el linker duplica el código para cada tipo con el que se usa
- por qué la diferencia?
  - ML: la x local es un puntero a un valor en el heap, con tamaño constante.
  - C++: la x local es un puntero a un valor en el stack, su tamaño depende del tipo.

propiedades importantes de la orientación a objetos

la abstracción procedural



propiedades importantes de la orientación a objetos

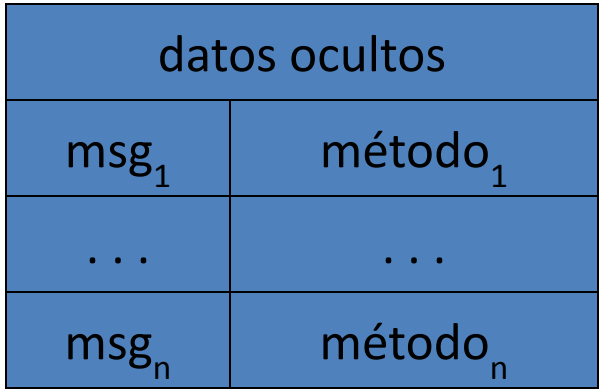
- objetos
- lookup dinámico
- encapsulación
- herencia
- subtipado

objetos

un objeto consiste de …

- datos ocultos
  - variables de la instancia (datos del miembro)
  - posiblemente funciones ocultas
- operaciones públicas
  - métodos (funciones del miembro)
  - puede tener variables públic en algunos mensajes

un programa orientado a objetos envía mensajes a los objetos



construcción de encapsulación <u>universal</u>

(se puede usar para estructuras de datos, sistemas de archivos, bases de datos, etc.)

lookup dinámico

- en programación convencional, el significado de una operación con los mismos operandos es siempre el mismo

  `operación(operandos)`

- en programación orientada a objetos,

  `object → message (arguments)`

el código depende del objeto y el mensaje

diferencia fundamental
entre TADs y objetos

sobrecarga vs. lookup dinámico

- en programación convencional `add (x, y)`
  la función `add` tiene significado fijo
- para sumar dos números `x → add (y)`
  tenemos un `add` distinto si `x` es entero, complejo, etc.
- semejante a la sobrecarga, con una diferencia crítica: la sobrecarga se resuelve en tiempo de compilación, mientras que el lookup dinámico se resuelve **en tiempo de ejecución**

encapsulación

- el constructor de un concepto tiene una vista detallada
- el usuario de un concepto tiene una vista abstracta
- la encapsulación separa estas dos vistas, de forma que el código de cliente opera con un conjunto fijo de oparciones que provee el implementador de la abstracción

subtipado y herencia

- la interfaz es la vista externa de un objeto
- el subtipado es una relación entre interfaces

- la implementación es la representación interna de un objeto
- la herencia es una relación entre implementaciones, de forma que nuevos objetos se pueden definir reusando implementaciones de otros objetos

interfaces de objeto

- interfaz
  - los mensajes que entiende un objeto
- ej: `Punto`
  - `x-coord` : devuelve la coordenada x de un punto
  - `y-coord` : devuelve la coordenada y de un punto
  - `move` : método para cambiar de ubicación
- la interfaz de un objeto es su tipo

subtipado

- si la interfaz A contiene todos los elementos de la interfaz B, entonces los objetos de tipo A también se pueden usar como objetos de tipo B

Punto

x-coord
y-coord
move

Punto_coloreado

x-coord
y-coord
color
move
change_color

la interfaz de `Punto_coloreado` contiene la de `Punto`, por lo tanto `Punto_coloreado` es un subtipo de `Punto`

ejemplo

```
class Point
   private
     float x, y
   public
      point move (float dx, float dy);
class Colored_point
   private
     float x, y; color c
   public
      point move(float dx, float dy);
      point change_color(color newc);
```

- **Subtipado:** `Colored_point` se puede usar en lugar de `Point`: propiedad que usa el **cliente**
- **Herencia**: `Colored_point` se puede implementar reusando la impelementación de `Point`: propiedad que usa el **implementador**

estructura de un programa orientado a objetos

- agrupar datos y funciones
- clase
  - define el comportamiento de todos los objetos que son instancias de la clase
- subtipado
  - organiza datos semejantes en clases relacionadas
- herencia
  - evita reimplementar funciones ya definidas

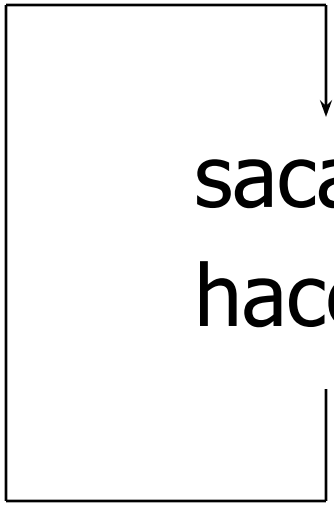
ejemplo: biblioteca geometría

- definimos el concepto general forma
- implementamos dos formas: círculo, rectángulo
- funciones sobre formas: centro, mover, rotar, imprimir
- anticipar cómo podría evolucionar la biblioteca

formas

- la interfaz de cada forma debe incluir
  centro, mover, rotar, imprimir
- las diferentes formas se implementan distinto
  - Rectángulo: cuatro puntos que representan las esquinas
  - Círculo: punto central y radio

ejemplo de uso: ciclo de procesamiento



sacar forma de una cola

hacer algo

el loop de control no conoce el tipo de cada forma

Subtipado ≠ Herencia