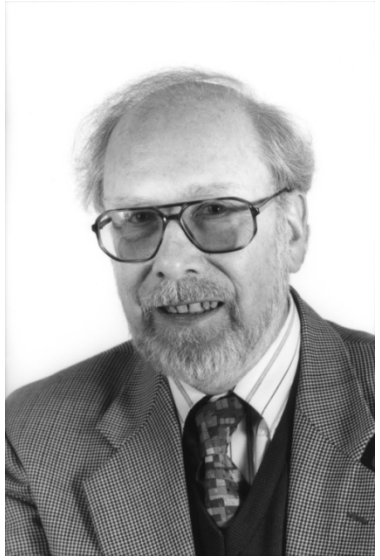
Modularidad y Programación Orientada a Objetos

Paradigmas de la Programación 2015
FaMAF-UNC
capítulos 9 y 10 de Mitchell
basado en filminas de Vitaly Shmatikov

temas a tratar

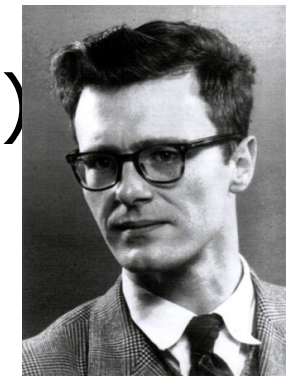
- desarrollo de programas modulares
  - refinamiento incremental
  - interfaz, especificación e implementación
- soporte de los lenguajes para la modularidad
  - abstracción procedural
  - tipos abstractos de datos
  - paquetes y módulos
  - abstracciones genéricas (con parámetros de tipo)

refinamiento incremental



- “… program … gradually developed in a sequence of refinement steps … In each step, instructions … are decomposed into more detailed instructions.”
  - Niklaus Wirth, 1971

ejemplo de Dijkstra (1969)

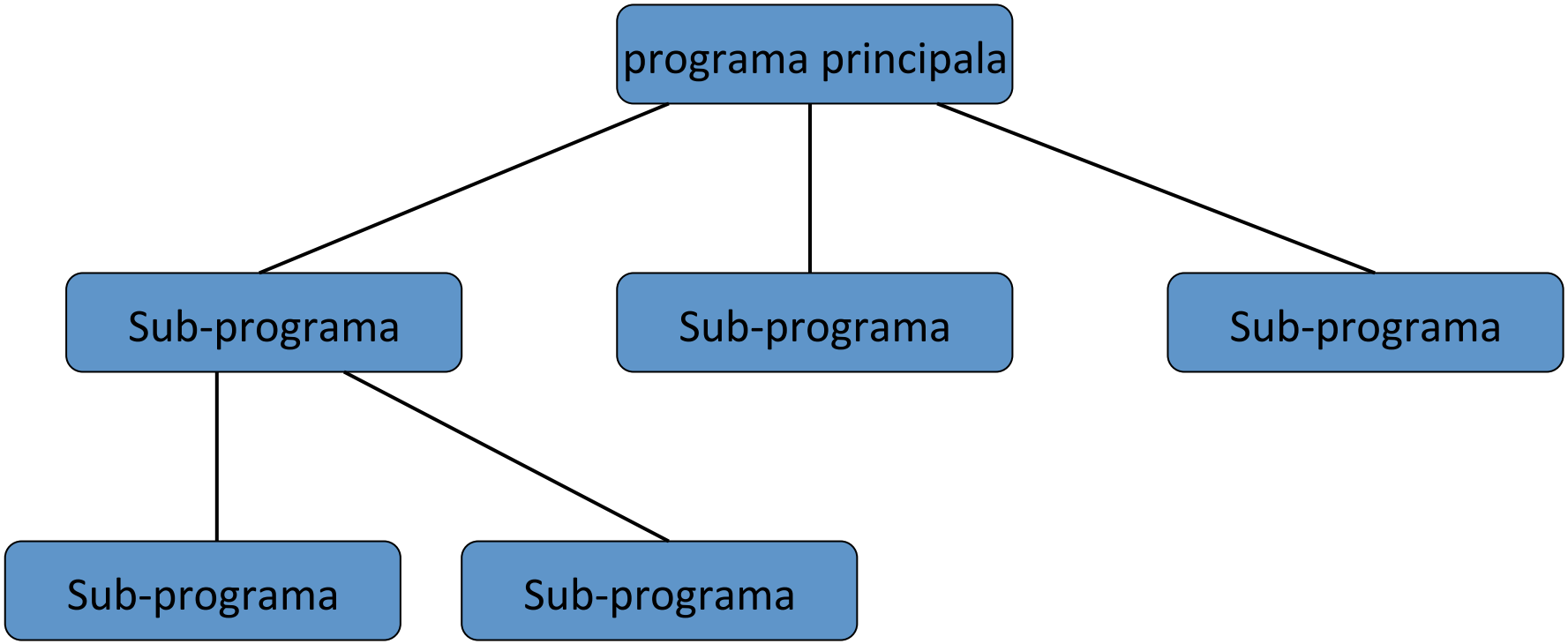


```
begin
    print first 1000 primes
end
```

```
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)

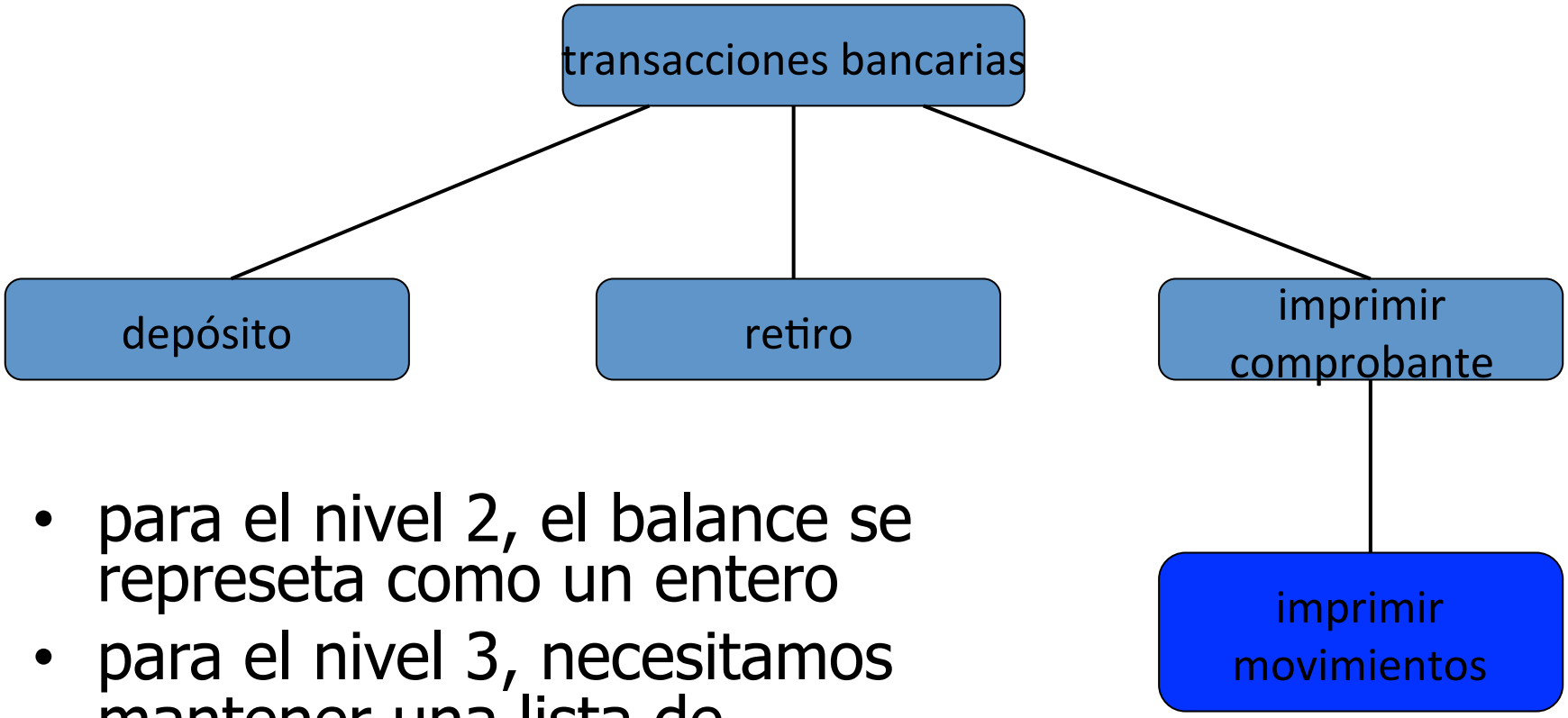


refinamiento de datos

- “As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel”
  – Wirth, 1971

  Así como se van refinando las tareas, también así los datos pueden tener que refinarse, descomponerse o estructurarse, y es natural refinar un programa y las especificaciones de datos en paralelo

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

- **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

usar la cola de prioridad

- 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
```

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 está 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

módulos y abstracción de datos

```
module Set
 interface
  type set
  val empty : set
  fun insert : elt * set -> set
  fun union : set * set -> set
  fun isMember : elt * set -> bool
 implementation
  type set = elt list
  val empty = nil
  fun insert(x, elts) = ...
  fun union(…) = ...
  ...
end Set
```

- con módulos se puede definir un ADT con tipo privado y operaciones públicas
- los módulos son más generales, pueden incluir varios tipos y operaciones relacionados
- algunos lenguajes separan interfaz e implementación, de forma que una interfaz puede tener múltiples implementaciones

abstracciones genéricas

parameterizar 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 las variables locales en el activation record
    - Ligado a las operaciones del tipo instanciado
- ejemplo: la función swap (overloading 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 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?
  - la x local es un puntero a un valor en el heap, con tamaño constante.
  - la x local es un puntero a un valor en el stack, su tamaño depende del tipo.

C++ Standard Template Library

- muchas abstracciones genéricas
  - operaciones tipos abstractos polimórficos
  - ejemplo de programación genérica
- eficiente en tiempo de ejecución (pero no siempre en espacio)
- escrito en C++
  - usa el mecanismo de templates y sobrecarga
  - no usa objetos – no hay funciones virtuales

Arquitecto: Alex Stepanov

principales entidades en la STL

- Contenedor: colección de objetos tipados
  - arreglo, lista, diccionario asociativo, ...
- Iterador: generalización de puntero o dirección
- Algoritmo
- Adaptador: convertir de uno a otro
  - ej: producir un iterador a partir de un contenedor actualizable
- Objeto función: forma de clausura (“manual”)
- Allocador: encapsulación de un pool de memoria

ejemplo de la aproximación de la STL

función para unir dos listas ordenadas (conceptual)

```
merge: rango(s) × rango(t) × comparación(u)
        →  rango(u)
```

- `rango(s)` – "lista" ordenada de elementos de tipo s, dada por punteros al primer y último elementos
- `comparación(u)` – función que devuelve un booleano sobre el tipo u
- s y t deben ser subtipos de u

merge en STL

- los rangos se representan con iteradores
  - el iterador es una generalización de un puntero
  - tiene ++ (moverse al siguiente elemento)
- el operador `comparación` es un objeto de la clase `Compare`
- se expresa el polimorfismo usando una template

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator1 last2,
                     OutputIterator result, Compare comp)
```

STL vs. C y C++ crudos

- C:

```
qsort( (void*)v, N, sizeof(v[0]), compare_int );
```

- C++, usando arreglos de C:

```
int v[N];
sort( v, v+N );
```

- C++, usande una clase vector:

```
vector v(N);
sort( v.begin(), v.end() );
```

programación orientada a objetos

varios conceptos de lenguaje importantes

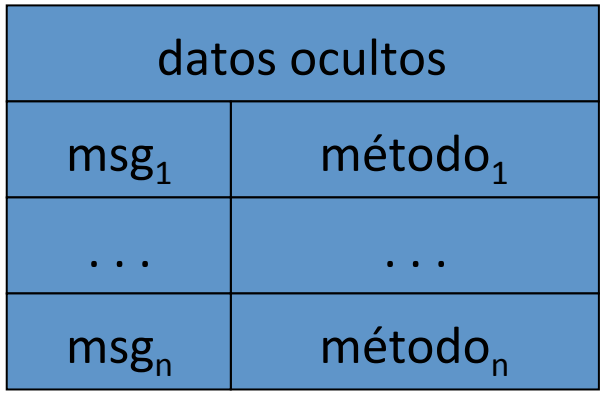
- lookup dinámico
- encapsulación
- herencia
- subtipado

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 públicas en algunos mensajes

un programa orientado a objetos envía mensajes a los objetos



construcción de encapsulación universal
(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: el overloading 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

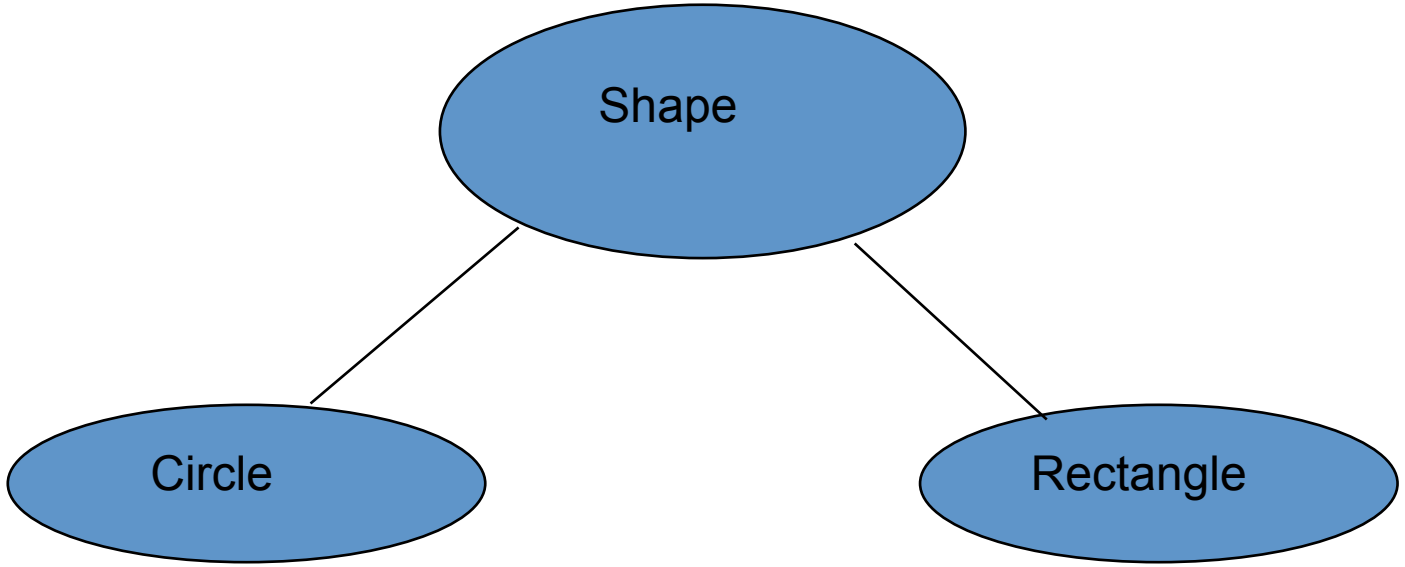
- definimos el concepto general forma
- implementamos dos formas: círculo, rectángulo
- funciones sobre formas: centro, mover, rotar, imprimir
- anticipar cómo podría evolucionar la biblioteca

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

Subtype Hierarchy



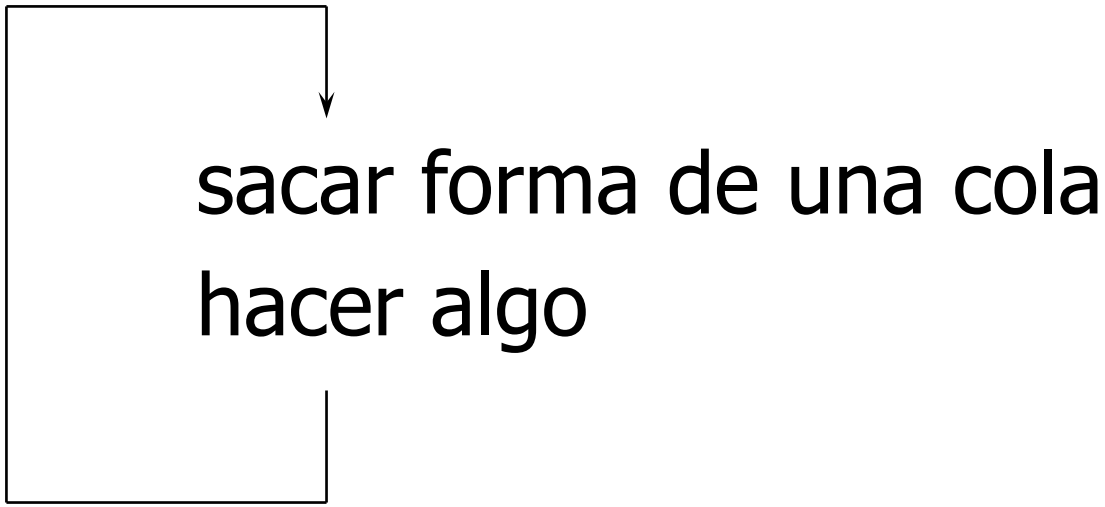
- General interface defined in the shape class
- Implementations defined in circle, rectangle
- Extend hierarchy with additional shapes

Código de cada clase

	centro	mover	rotar	imprimir
Círculo	c_center	c_move	c_rotate	c_print
Rectángulo	r_center	r_move	r_rotate	r_print

- Dynamic lookup
  - circle → move(x,y) calls function c_move
- Conventional organization
  - Place c_move, r_move in move function

ejemplo de uso: ciclo de procesamiento



el loop de control no conoce el tipo de cada forma

Subtipado ≠ Herencia