Paradigmas de programación

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Agosto 2009

Outline

Outline

Lenguajes de programación

Existen docenas de lenguajes de programación, y se sigue creando nuevos. ¿Por qué ?

- Problemas muy diferentes : manipulaciones simbólicas, estructuras de datos complejas, programación cerca de la maquina, necesitad de representación lógica fuerte. . .
- Cada uno tiene sus propias preferencias en cuanto al estilo de programación.

Es interesante estudiar la filosofía de cada uno para :

- Entender las evoluciones del mundo de la computación.
- Poder mas fácilmente aprender nuevos lenguajes.
- Medir su propio sesgo debido a los lenguajes que uno conoce.

Lenguajes de programación

1290 variaciones de un programa, en varios lenguajes (el 27/08/2009)

`http://99-bottles-of-beer.net/`

"Implementa" la canción:

```
99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.
98 bottles of beer on the wall, 98 bottles of beer.
Take one down and pass it around, 97 bottles of beer on the wall.
...
No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.
```

Lenguajes de programación

```
/*
 * C (multithreaded) using POSIX Thread Library (pthread)
 * by Stefan Scheler <sts[at]synflood[dot]de>
 * Ilmenau, Germany - May 2005
 *
 * compile with: gcc -pthread -o 99bottles 99bottles.c
 * disable debug output with: ./99bottles 2>/dev/null
 *
 */

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

#define THREADS 10
#define PLURALS (bottles>1) ? "s" : ""

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int bottles = 99;
```

Lenguajes de programación

```
void *takeonedownandpassitaround(void *ptr) {
    struct timespec sleeptime;
    int *thread_id = (int *)ptr;
    while (1) {
        pthread_mutex_lock(&mutex);
        if (bottles>0) {
            fprintf(stderr, "Thread %d: ", *thread_id);
            printf("%d bottle%s of beer on the wall, %d bottle%s of b
                bottles, PLURALS);
            fprintf(stderr, "Thread %d: ", *thread_id);
            printf("Take one down and pass it around, ");
            if (bottles==1)
                printf("no more bottles of beer on the wall.\n");
            else
                printf("%d bottle%s of beer on the wall.\n", bottles-
        } else if (bottles==0) {
            fprintf(stderr, "Thread %d: ", *thread_id);
            printf("No more bottles of beer on the wall, no more bott
            fprintf(stderr, "Thread %d: ", *thread_id);
            printf("Go to the store and buy some more, 99 bottles of
        } else {
            pthread_mutex_unlock(&mutex);
            break;
```

Lenguajes de programación

```
int main(void) {

    pthread_t thread[THREADS];
    int threadid[THREADS];
    int i;

    srand(time(NULL));

    for (i=0; i<THREADS; i++) {
        threadid[i] = i;
        pthread_create(&thread[i], NULL, takeonedownandpassitaround,
         &threadid[i]);
    }

    for (i=0; i<THREADS; i++)
        pthread_join(thread[i], NULL);

    exit(EXIT_SUCCESS);

}
```

Lenguajes de programación

Cualidades para un lenguaje :

- Expresividad: que el numero máximo de problemas se puedan expresar con este lenguaje.
- Facilidad del aprendizaje (como lenguas...).
- Portabilidad: que pueda compilar en maquinas de bajo costo, en diferentes OS.
- Cualidad de la implementación.
- Factores económicos: inercia por el mercado actual.
- ...

Lenguajes de programación

Lenguajes de aprendizaje: KAREL

```
BEGINNING–OF–PROGRAM

  DEFINE turnright AS
  BEGIN
    turnleft
    turnleft
    turnleft
  END

  BEGINNING–OF–EXECUTION
    ITERATE 3 TIMES
      turnright

    turnoff
  END–OF–EXECUTION

END–OF–PROGRAM
```

http://mormegil.wz.cz/prog/karel/prog_doc.htm

Lenguajes de programación

Lenguajes de aprendizaje: KAREL

- conjunto muy reducido de instrucciones,
- excelente para la educación,
- pero llegan rápido los limites: expresividad no muy grande.

Lenguajes de programación

Estudiar lenguajes globalmente:

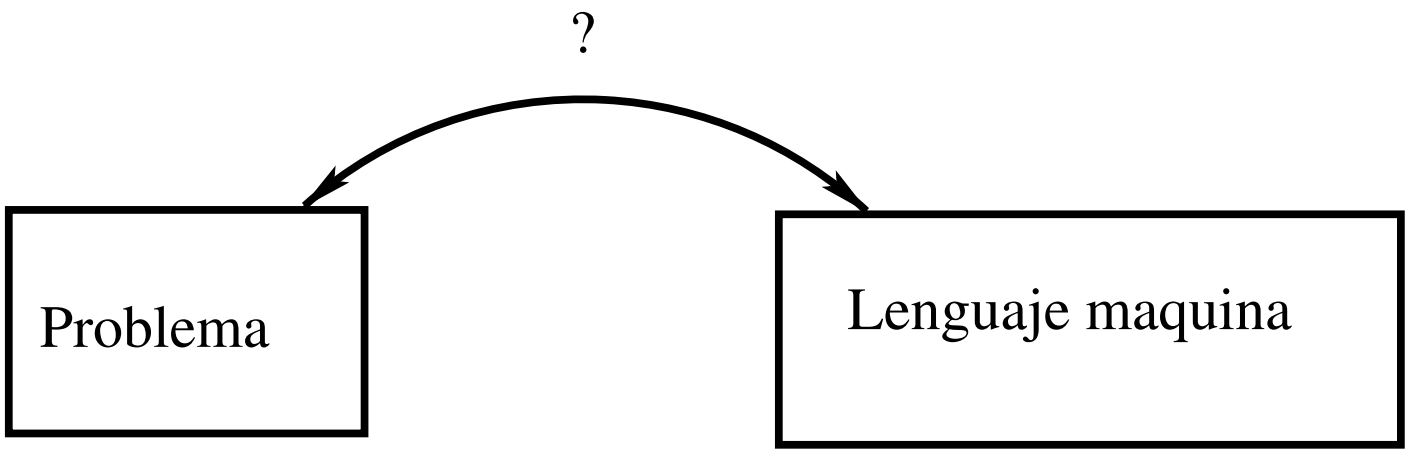
- Permite entender por medio de formulaciones generales elecciones muy especificas en cuanto a un lenguaje dado.
- Permite elegir el lenguaje mas apropiado para una tarea.
- Eventualmente permite "simular" un comportamiento genérico deseable pero ausento (explícitamente) del lenguaje en que trabajamos (ejemplo : POO en C).

Lenguajes de programación

En general los diferentes lenguajes tienen expresividad similar : los problemas pueden estar resueltos en varios lenguajes; aun en el mismo lenguaje, la resolución de un problema puede ser dada de maneras muy diferentes :

- Por el estilo.
- Por las elecciones algorítmicas.
- Por las representaciones mentales subyacentes a la resolución : paradigma de programación.

Paradigmas de programación



Entre todas las maneras posibles para programar la resolución de un problema, ¿cuál elegir? ¿qué esquema seguir?

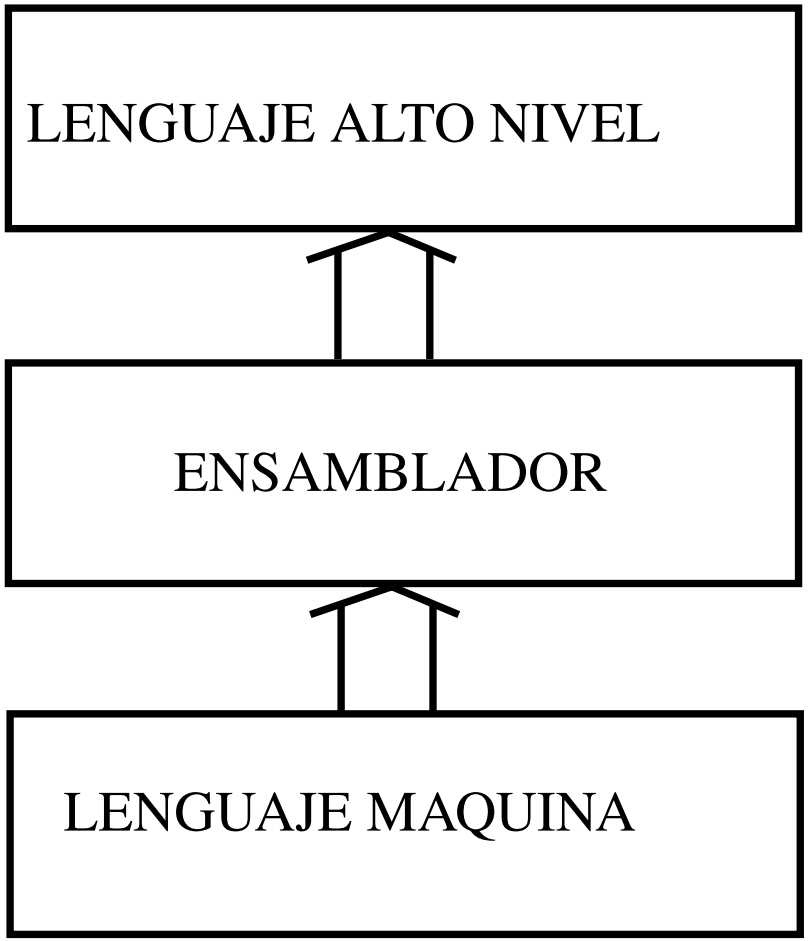
Programación $\neq$ Lenguajes de programación

Definir un paradigma de programación se hace independientemente del lenguaje en sí mismo : es darse una manera de pensar un programa (concebirlo, analizarlo...) a partir de un conjunto de conceptos.

Ahora, según el lenguaje que uno toma, es mas o menos fácil de implementar este paradigma. Es mas fácil implementar un programa concebido de manera OO (Orientada Objeto) en C++ o Java; pero se podría hacerlo en C.

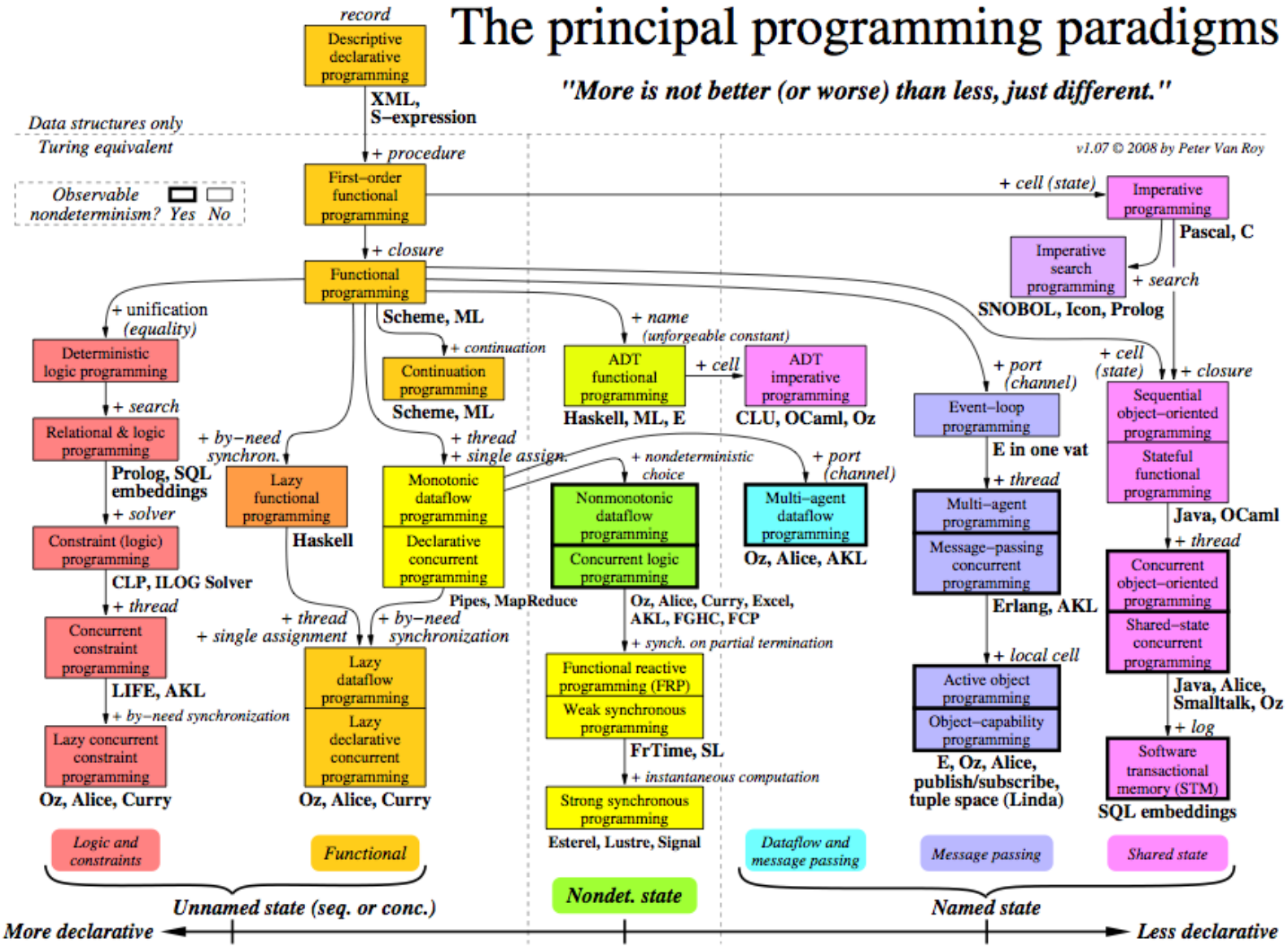
Además, en un mismo lenguaje/programa, varios paradigmas pueden coexistir.

Paradigmas y abstracción



Nuevos paradigmas aparecen mientras mas se aleja de la programación maquina, y mientras la industria se forma: necesidad de reutilizar código!

Paradigmas y abstracción (P. Van Roy)



Programación imperativa

Es el paradigma que surgió mas naturalmente al desarollo de las computadoras : las maquinas estan físicamente construidas para ejecutar series de instrucciones maquinas ($\neq$ cerebro) ; se concibieron abstraciones de esas instrucciones o combinaciones de elllas, para dar lugar a lenguajes de mas alto nivel.

Surgen naturalmente de los ensambladores: describen la resolución del problema en una serie de operaciones. Usan componentes abstractos elementales que son asignaciones, ramificaciones condicionales, ramificaciones incondicionales...

Al centro de la idea es el estado del programa (memoria...) que va siendo modificado por las operaciones.

Programación imperativa

Palabras llaves: estado, asignaciones, ciclos de control.

Ejemplos: ensambladores...

Problemas:

- difícil de reutilizar código de manera simple,
- tendencia a producir "código espagueti" en particular al usar goto.

Programación procedural

Para cumplir una tarea dada, descomponemos todo en sub-tareas de calculo que se cumplen sucesivamente para alcanzar la meta: rutinas, sub-rutinas, funciones... Mas modularidad, permite mejor re-utilización del código.

La gran mayoría de los lenguajes que conocen (Fortran, Basic, C, Java...) permiten seguir este paradigma.

En su versión mas radical (sin Goto en particular): programación estructurada

Programación procedural con C

- Secuencias de instrucciones.
- Bloques equivalentes a instrucciones individuales.
- Funciones.
- Estructuras de control : iteraciones, elecciones condicionales.
- Variables (globales o locales) que son el elemento característico del paradigma.

Programación procedural con C

+ El programa se entiende fácilmente gracias a las estructuras de control.
+ Permite una forma de modularidad : se puede separar interfaces y implementaciones; se hace estructuras especificas a un problema y se separa el código propio a esa estructura.
- La modularidad podría ir aun mas adelante.

Programación procedural

Palabras llaves: estructuras de datos, rutinas,

Ejemplos: Fortran, Basic, C...

Problemas:

- faltarían mas conceptos para mas modularidad.
- rutinas = funciones matemáticas?

Programación funcional

En la programación funcional, como lo indica su nombre, la entidad básica es la función. Se intenta evitar referencias a variables
Lenguajes asociados : Lisp, Scheme...

```
(define (factorial n)
    (if (<= n 1)
        1
        (* n (factorial (- n 1)))))
```

- Mucha importancia en el desarrollo de IA.
- Consenso en que es muy poderoso y permite implementar muchos conceptos.

Programación funcional pura

- No side-effect : las funciones aquí no son simplemente ramificaciones que comparten datos con otras partes del programa, son entidades independientes (en particular valores dadas para los argumentos daran siempre el mismo argumento); no afectaciones. Mas “puro”, matemáticamente.
- No noción de variable.
- Lenguaje funcional puro: HASKELL .

Programación funcional

Palabras llaves: funciones matemáticas, $\lambda-$cálculo

Ejemplos: Lisp, Scheme, Haskell...

Problemas:

- menos flexible y práctico.

Programación lógica

En este paradigma, se consideran principalmente predicados y reglas lógicas que alimentan una base de conocimientos. Ejecutar un programa, en este caso, es evaluar o un predicado (sí o no) o dar valores para que un predicado es verdadero (todo eso deducido de la base de conocimientos y de las reglas).
Ejemplo típico, Prolog :

```
factorial(0,1).
factorial(N,F) :-
    N>0,
    N1 is N-1,
    factorial(N1,F1),
    F is N * F1.

?-  factorial(3,W).
W=6
```

Programación lógica

Aunque pueda parecer "torcido", sirve enormemente este paradigma, en particular en la implementación de sistemas complejos donde planear tareas se puede hacer mediante reglas

Lenguajes de programación

Cualidades necesarias para un programa :

- Corrección.
- Eficiencia.
- Legibilidad, concisión
- Robustez : entradas no previstas, limitaciones software o hardware
- Adaptabilidad : portabilidad hacia otras arquitectura
- Reutilización : necesita código suficientemente general, y claramente especificado.
- ...

Retos de la programación : robustez

Para que un programa complejo sea robusto es útil poder separar interfaces e implementaciones y modularizar :

- el usuario "cliente" de las librerías desarrolladas no se preocupará de las implementaciones (no es su problema).
- si se descubre un *bug*, sólo la implementación del modulo que falla estará cambiada.

Separación interfaz/implementación en C

En midi.h :

```
midiFile *readMidiFile(const char *fileName);
midiFile *createMidiFile();
int readMidiTrack(FILE *f, midiTrack *track);
midiTrack *createMidiTrack(midiTrack **nxt);
```

Separación interfaz/implementación en C

En midi.c :

```
//////////////////////
// Midi track I/O
int readMidiTrack(FILE *f, midiTrack *track) {
  char firstchain[4];
  unsigned char curStatus;
  unsigned char curChannel;
  // read first 4 bytes
  fread(&firstchain[0],1,4,f);
  if (firstchain[0]!='M'||
      firstchain[1]!='T'||
      firstchain[2]!='r'||
      firstchain[3]!='k') {
    fprintf(stderr,"This is not a MIDI track header")
    return 1;
  }
```

Separación interfaz/implementación en C

Lo que se da al cliente es :

- El archivo $\mathrm{midi.h}$.
- Una biblioteca $\mathrm{midi.dll}$ (por ejemplo).

Separación interfaz/implementación en C

Limites :

- El cliente tiene acceso a la estructura interna de la estructura.
- Podría aprovechar para hacer cosas que no queremos que haga.
- ¿Cómo saber si los apuntadores que regresamos son bien apuntadores a estructuras validas? Por un lado perdemos control de las estructuras al pasar por apuntadores...
- Los mecanismos con que aislamos las cosas no son impuestos por el lenguaje, es auto-disciplina.

Programación orientada a objetos

La meta es buscar la modularidad máxima a través de la separación del código en unidades conceptuales para :

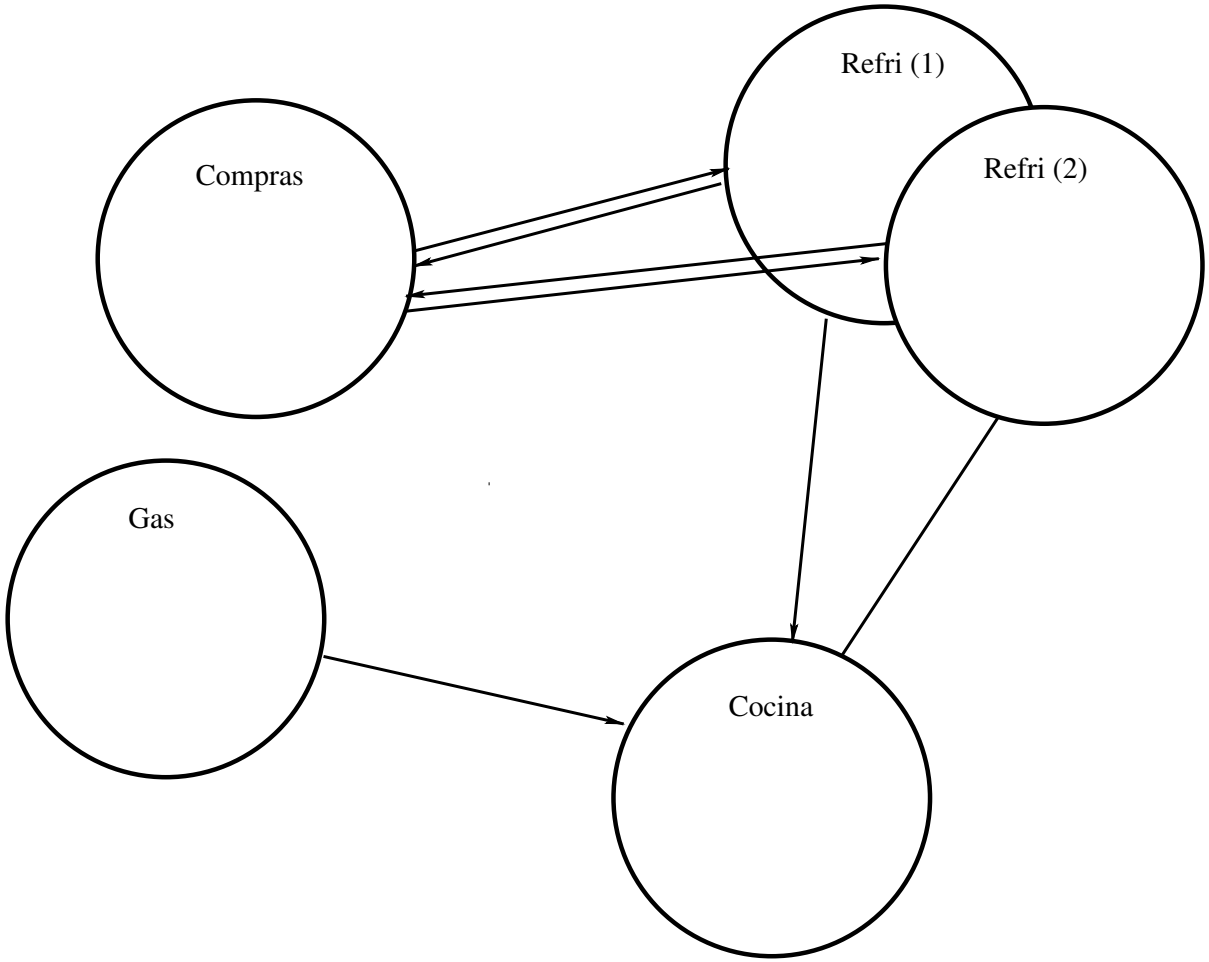
- Desarrollar partes de código independientemente de la evolución de la implementación o de la estructura interna de las otras partes.
- No necesitar conocer los detalles de implementación de las partes que usamos como clientes.
- Poder reutilizar el código en programas completamente distintos.

Outline

POO : principios

Se intenta hacer abstracción de la implementación física del programa (memoria...) para representarlo a través de entidades, los objetos, que interactuan a través de demandas (mensajes) transmitidas para efectuar operaciones.

POO : principios



Objetos comunican entre ellos por mensajes y actúan según su "nivel de responsabilidad"; aceptan la responsabilidad de una tarea al recibir mensajes; pueden delegar trabajo a otras clases.

POO : principios

Ejemplo : un programa de simulación de robot humanoide; el objeto robot recibe un mensaje para que el robot se sienta; el robot pide la trayectoria y la secuencia de comandos a un objeto planificador ; luego envía mensajes a los objetos actuatores (motores brazos, piernas. . .) que a su vez envían su estado corriente (para eventualmente corregirlos); objetos sensores envian sus datos a un objeto calculador de posición que también puede corregir las trayectorias/comandos.

POO : principios

Dixit A. Kay (Smalltalk) :

1. Everything is an object.
2. A program is a bunch of objects telling each other what to do by sending messages.
3. Each object has its own memory made up of other objects.
4. Every object has a type.
5. All objects of a particular type can receive the same messages

POO : concepción

Una ventaja de la POO es que la conceptualización de los problemas y de su resolución se hace mas fácil, en particular con herramientas gráficas (UML).
La concepción en POO consiste primero a bien identificar los objetos que permiten representar mejor el problema.
Otra ventaja que es tal representación permite resolver toda clase de problemas involucrando los objetos presentes.

POO : en practica

- Objetos son estructuras de datos en memoria.
- Un objetos lleva dos tipos de cosa : datos (o atributos) y código ejecutable (operaciones que este objeto puede hacer, como funciones C).
- Los objetos tienen una interfaz y una implementación; solo la interfaz esta visible por los otros objetos del entorno.

POO : en practica

Datos :

- Son guardados en variables propias al objeto

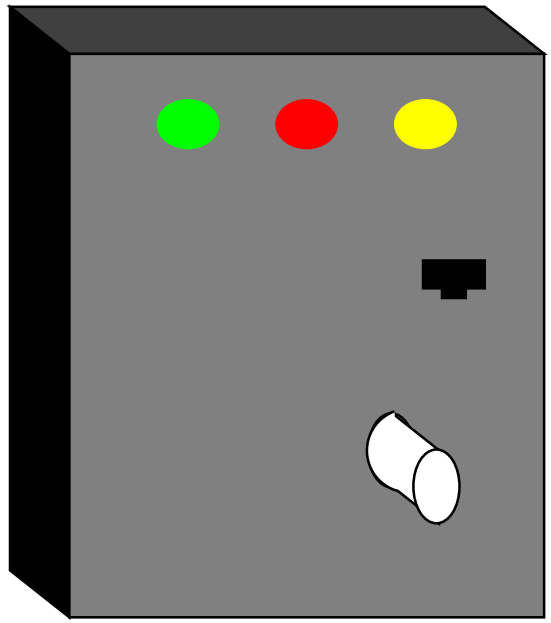
Operaciones :

- Las operaciones que puede hacer el objeto se llaman métodos.
- La interfaz precisa el prototipo de cada método : nombre, valor de regreso, nombre y tipos de los argumentos.
- Los métodos pueden cumplir tareas diversas : crear objetos, enviar mensajes, hacer manipulación sobre las variables internas del objeto.

POO : encapsulación

Lo que me sirve

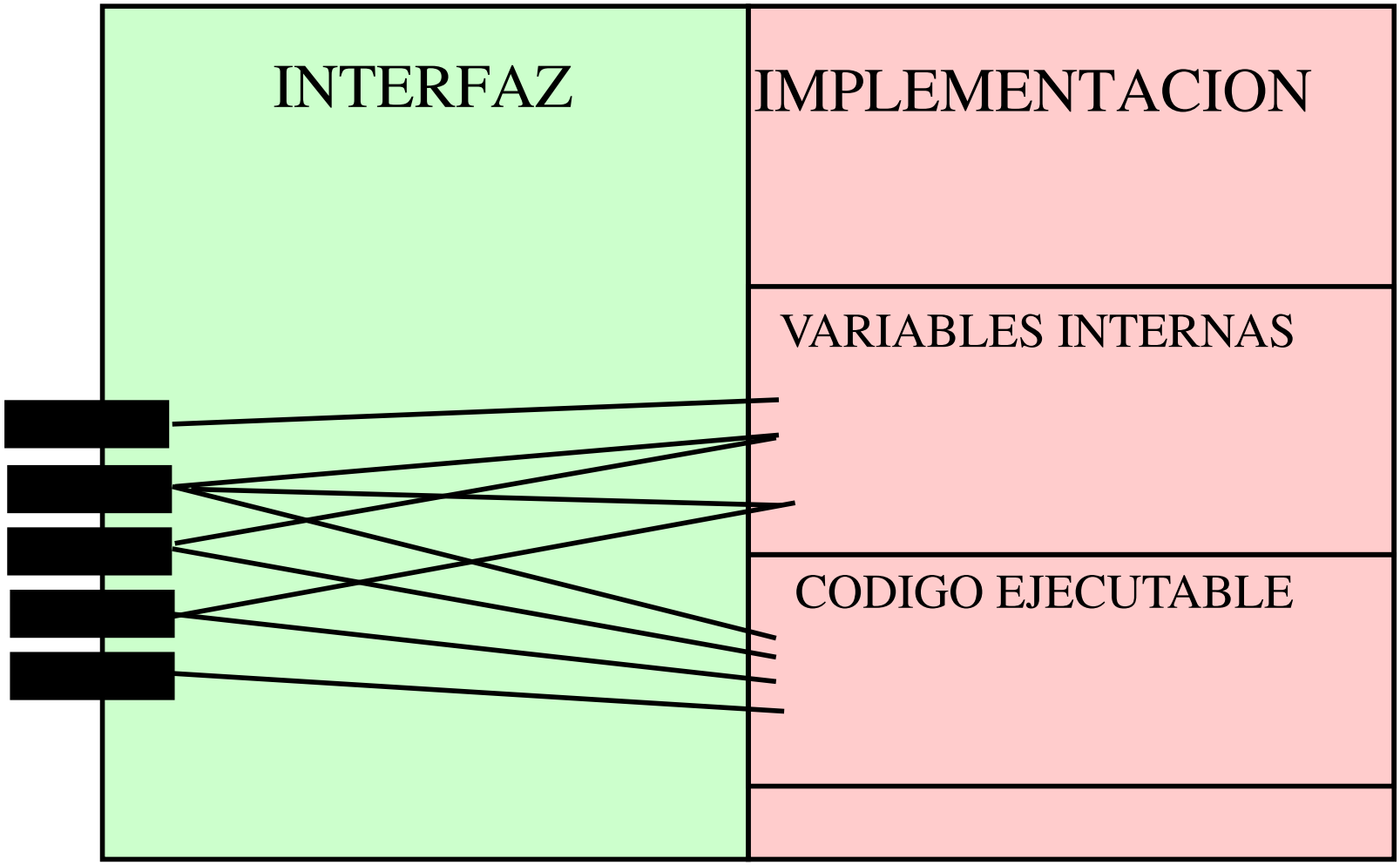
-prender()
- apagar()
- actuar()



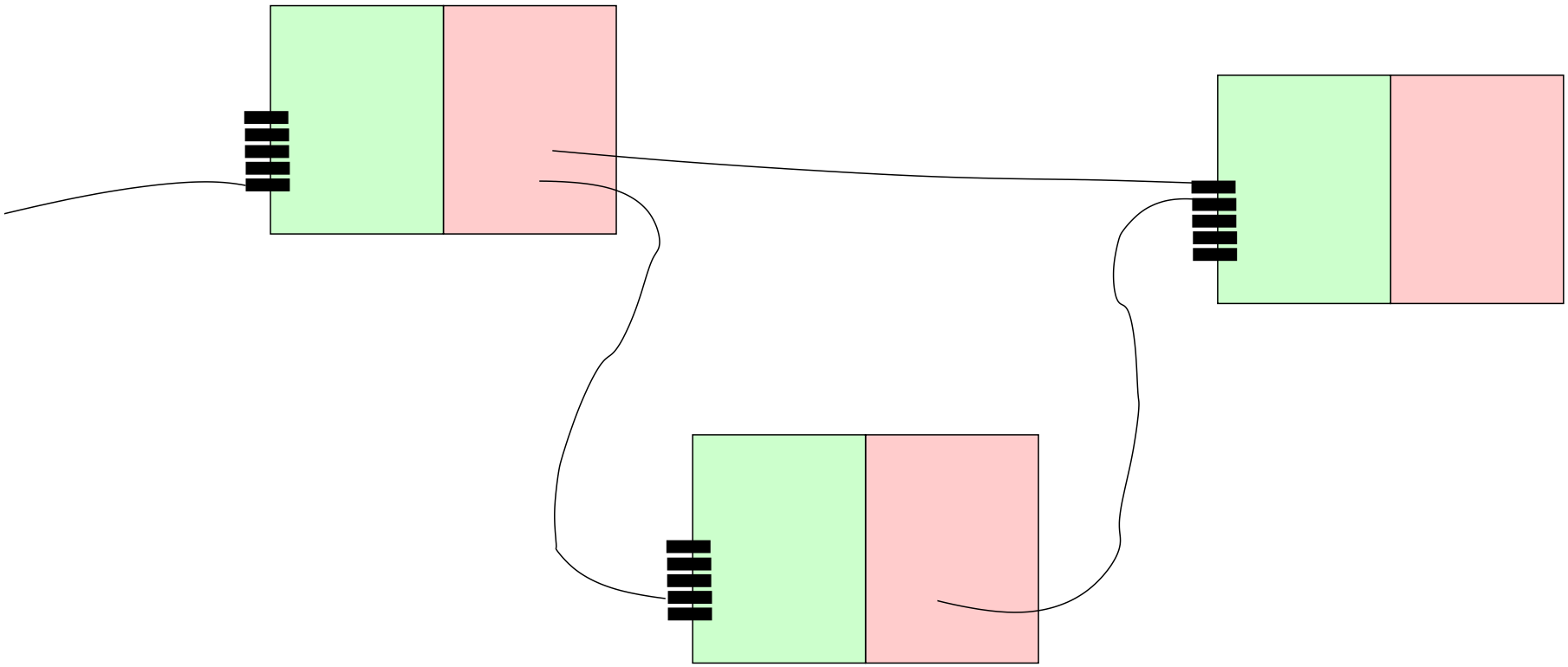
Lo que no veo

- electronica
- programacion interna

POO : encapsulación



POO : encapsulación



POO : encapsulación

Es una de los principales conceptos en POO : escondes del exterior el estado interno del objeto, típicamente atributos, a los que sólo puedes acceder por mensajes,

- de acceso al valor (getCosa()),
- de modificación de valor (setCosa(valor)).

Ejemplos : robot.getTrajectory(), robot.getName()

POO : encapsulación

Ventajas :

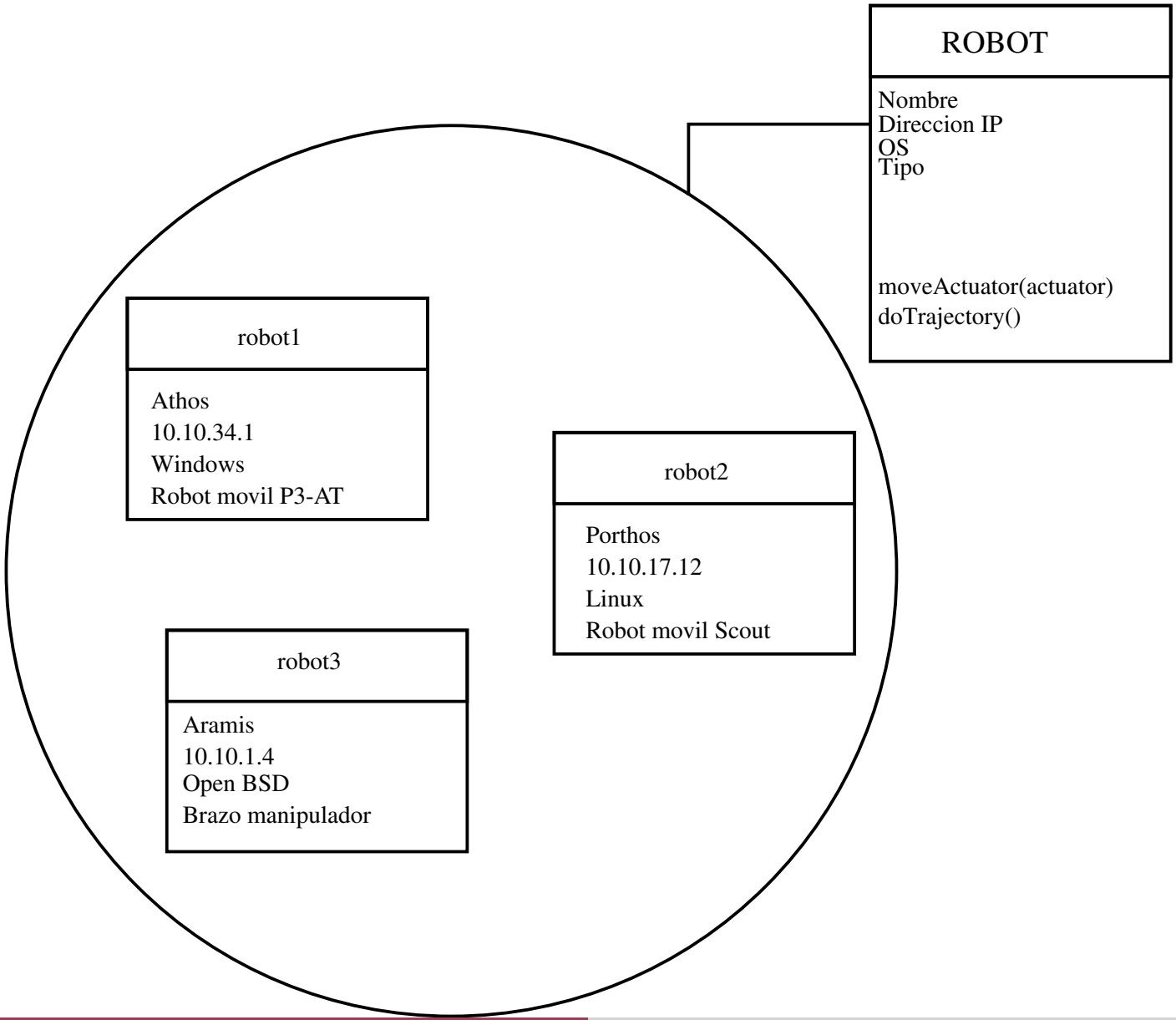
- para el "cliente" : solo tiene que manipular lo esencial, no se arriesga a nada más que lo que especifica el desarrollador a través de la interfaz,
- para el desarrollador : puede modificar largas porciones de código sin molestar al "cliente".

POO : clase

Para poder crear un objeto, se necesita saber cual es el tipo de sus variables, su interfaz, el código de sus métodos . . . : el conjunto de todas esas características es lo que se llama una clase (abstracción de un objeto).

Se dice que un objeto instancia una clase; el programa, en POO, es una serie de definición de clases. Clase = tipo, pero mas enfocado al problema que a la representación memoria (también incluyendo funciones. . .).

POO : clase

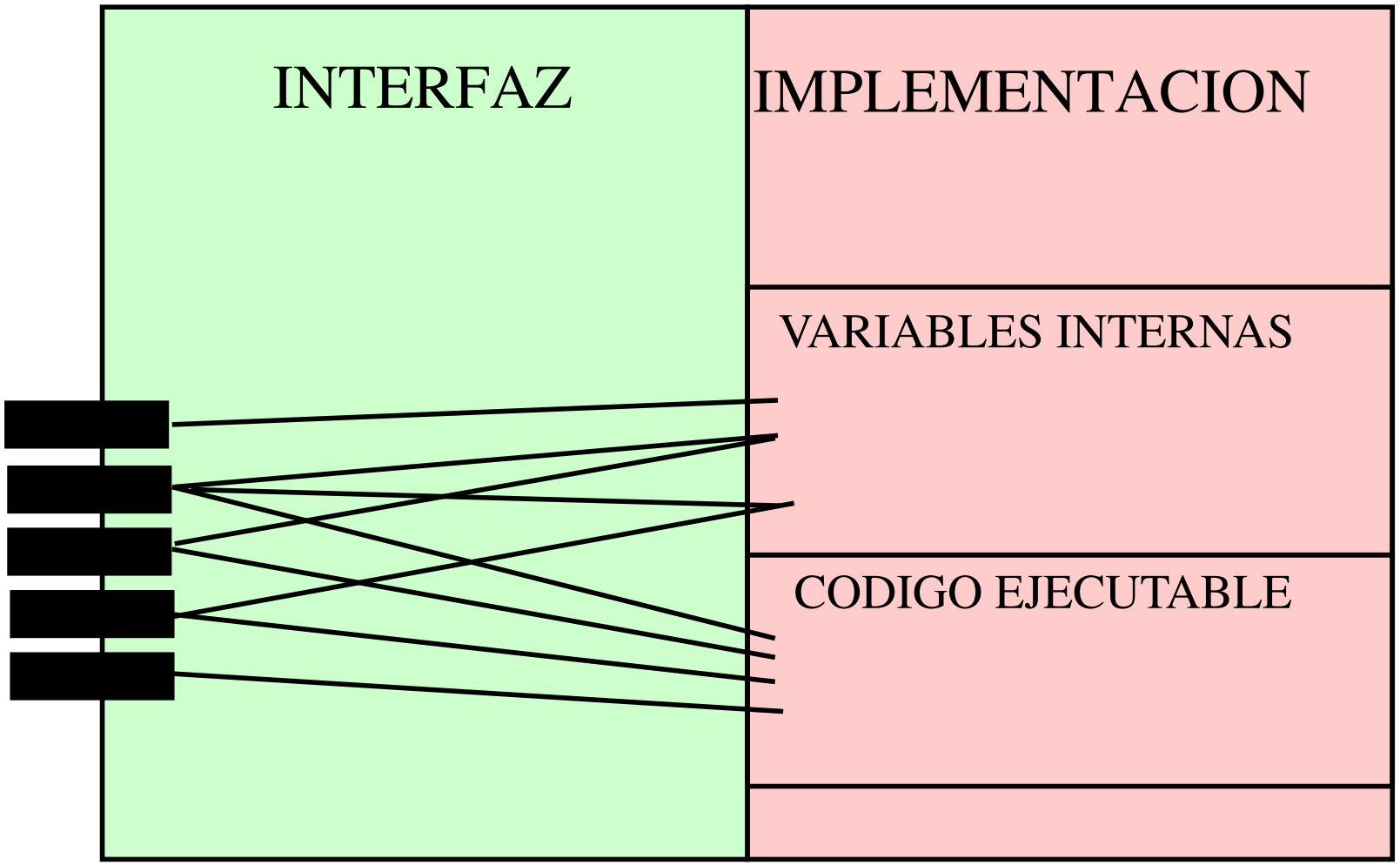

ROBOT
Nombre
Direccion IP
OS
Tipo
moveActuator(actuator)
doTrajectory()
robot1
Athos
10.10.34.1
Windows
Robot movil P3-AT
robot2
Porthos
10.10.17.12
Linux
Robot movil Scout
robot3
Aramis
10.10.1.4
Open BSD
Brazo manipulador

POO : clase

Dos objetos instanciando una misma clase :

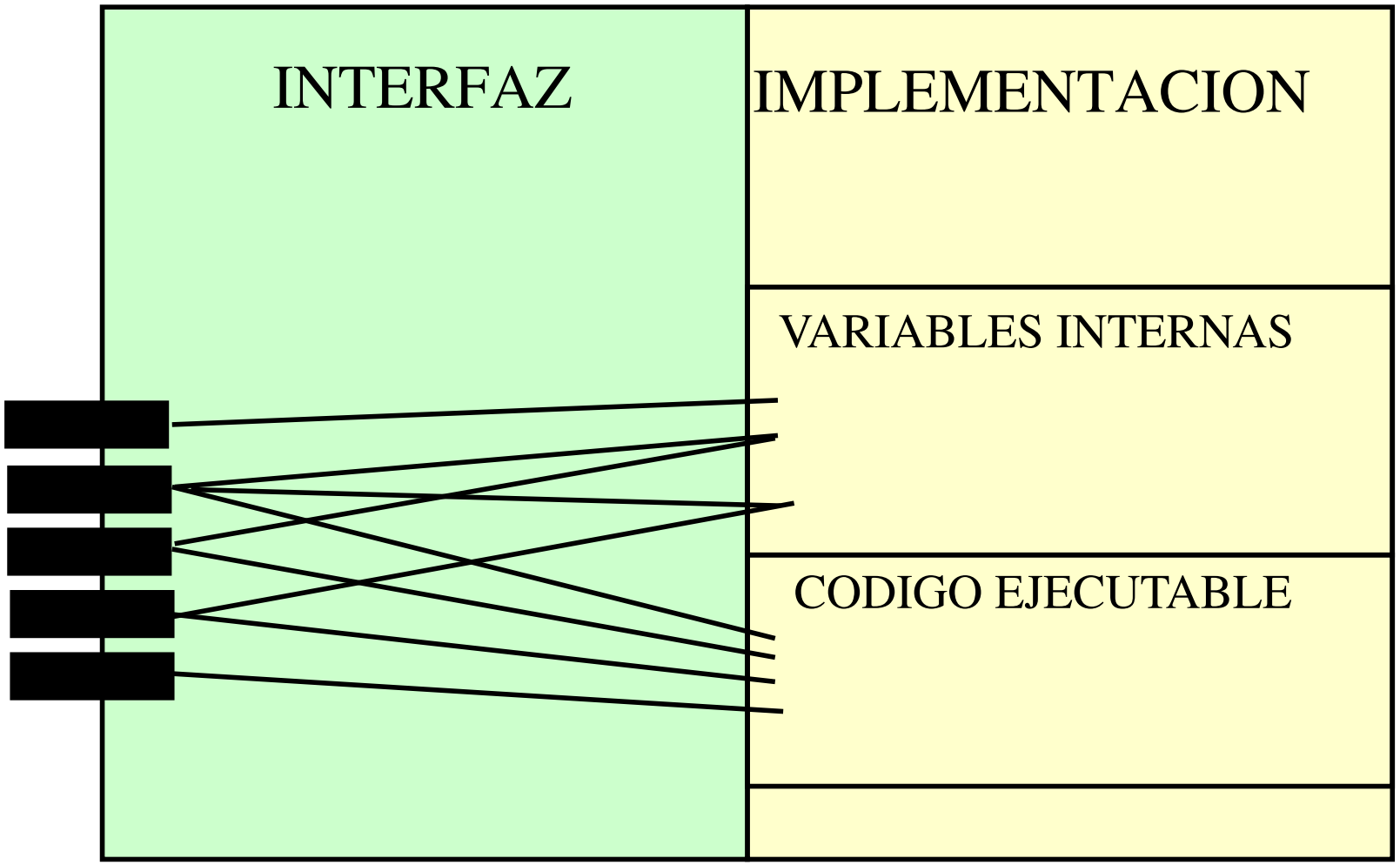
- Tienen variables internas distintas (a priori) : como instancias de estructuras.
- Comparten el código de sus métodos.
- Los métodos llamados en el objeto A “ven” todas las variables del objeto A.

POO : interfaz/implementación



La clase roja hace la implementación de la interfaz de una manera (con unas elecciones algorítmicas dadas).

POO : interfaz/implementación



La clase amarilla hace la implementación de la misma interfaz con otras elecciones algorítmicas.

POO : interfaz/implementación

Es importante poder separar interfaz e implementación para poder implementar interfaces de varias maneras; es posible hacerle explícitamente en Java, hay que usar desvíos en C++ (clases de base abstractas).
En Java:

```
public interface CocinarOmelette {
        void preparar(int nInvitados);
        void servir();
}
```

POO : interfaz/implementación

En Java:

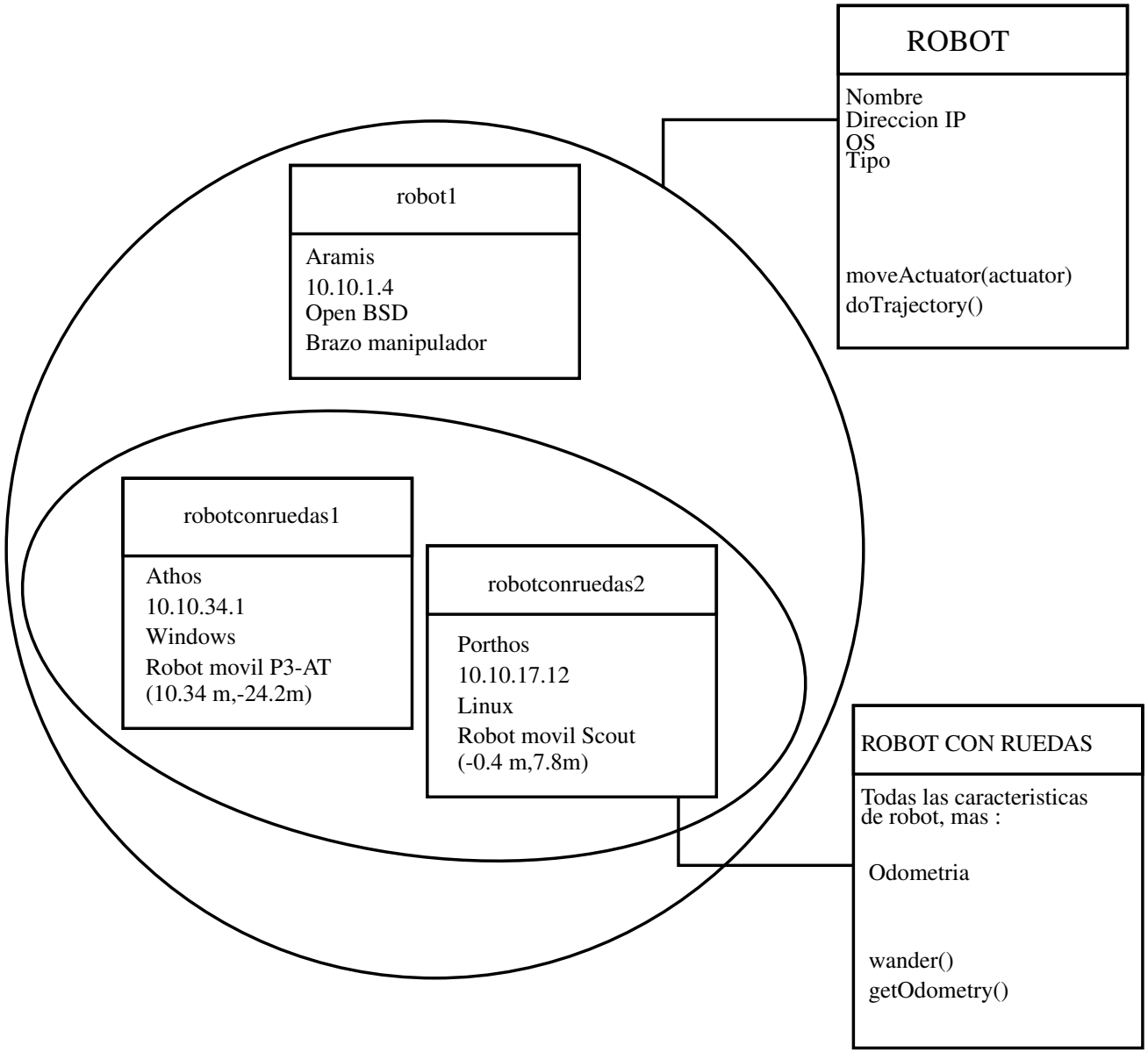
```
public class recetaJB implements CocinarOmelette {
        private int huevosDeGallina;
        private int chile;
        void preparar(int nInvitados) {
                ... // A la manera de JB
        }
        void servir() {
                ....
        };
}
public class recetaClaudia implements CocinarOmelette {
        private int huevosDeAvestruz;
        private int papas;
        void preparar(int nInvitados) {
                ... // A la manera de Claudia
        }
        void servir() {
                ....
        };
}
```

POO : herencia

Es un concepto fundamental también en POO : dado una clase, puedo imaginar clases más especificas o mas generales : que tengan subconjuntos o sobreconjuntos de los atributos/métodos de la clase considerada.
Ejemplos :

- Una clase $\mathrm{Robot\ con\ ruedas}$ sería una clase heredando de la clase Robot.
- Una clase $\mathrm{Computadora}$ generalizaría la clase Robot, que heredaría de esa.

POO : herencia



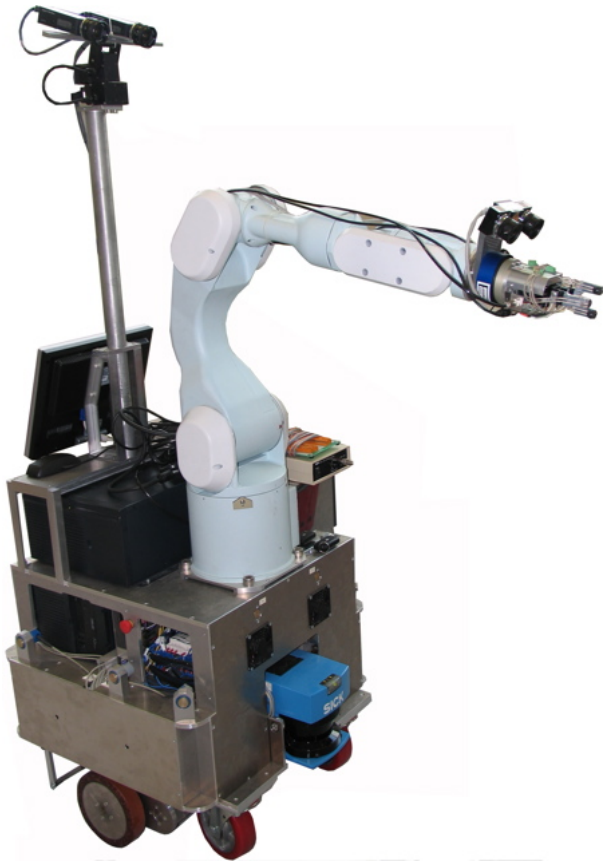
POO : herencia

Permite establecer una jerarquía de niveles de abstracción muy útil : en función del problema, o del lugar en la resolución del problema, se podrá usar el nivel de abstracción mas adecuado.

Permite polimorfismo.

POO : herencia múltiple

En unos lenguajes se puede usar herencias múltiples, o sea que la clase que se define herede de varias clases padres (C++).
Ejemplo : una clase para los robots con ruedas y con brazo manipulador



POO : herencia múltiple

Problemas :

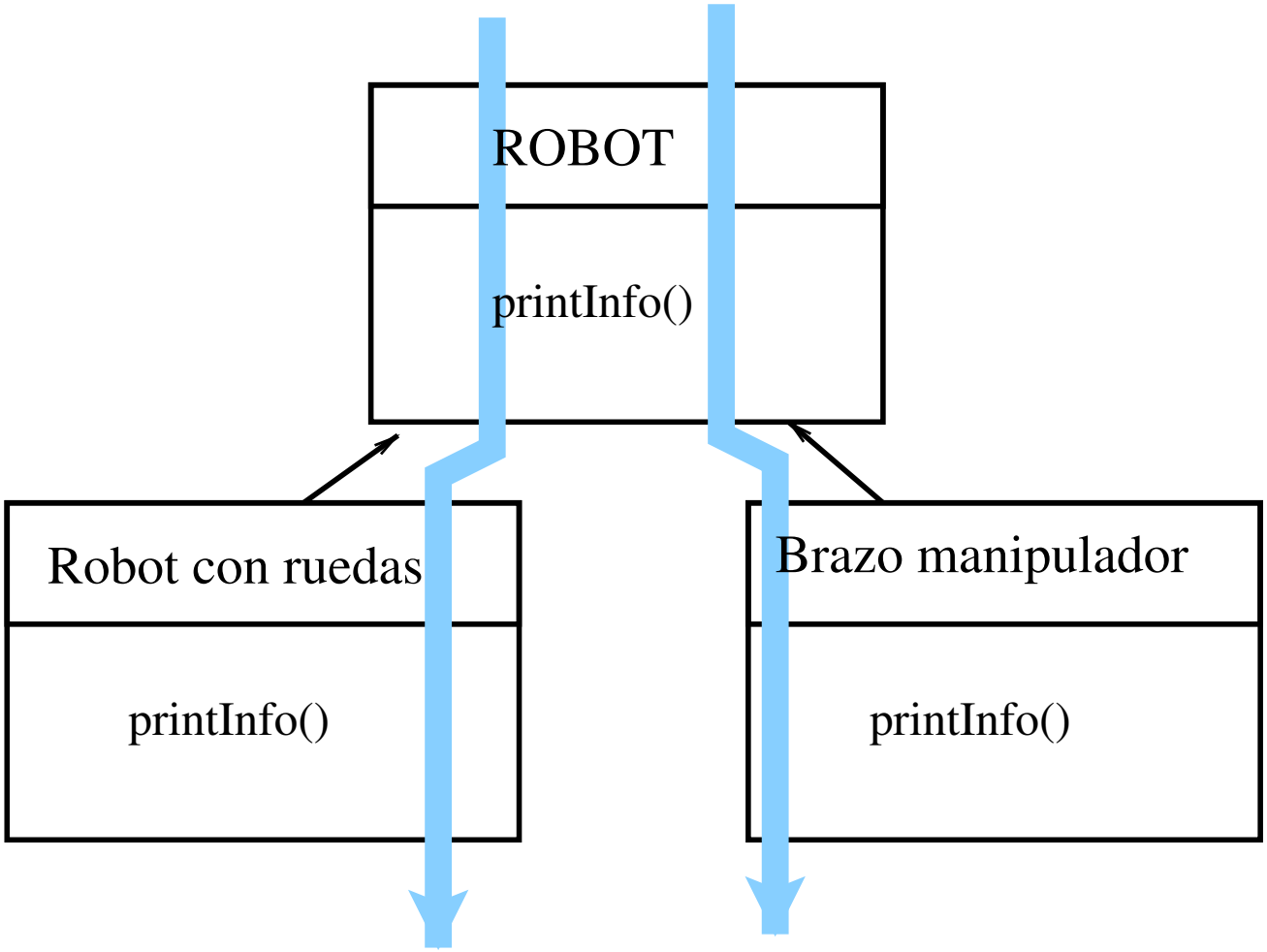
- Para entitades presentes en varias clases padres : fusionarlas ? separarlas ? (por ejemplo, el estado state del robot)
- Problemas a la compilación/ejecución : qué método ejecutar ?

POO : polimorfismo

Otro concepto fundamental es el del polimorfismo, consecuencia de la herencia : es la propiedad de poder usar el mismo nombre de método para formas mas o menos especializadas, eventualmente sin necesitar tomar en cuenta el tipo exacto, sino el tipo "padre".

Ejemplo : un método printInfo() de la clase Robot podría ser especializado (con más informaciones especificas) en clases hijas.

POO : polimorfismo



POO : polimorfismo

- Este polimorfismo también se llama polimorfismo de herencia
- Otros tipos de polimorfismo (pero no tanto ligado a la POO):
  - polimorfismo ad-hoc: existencia de métodos de mismo nombre para objetos sin relación particular (ejemplo, el operador +)
  - polimorfismo paramétrico: los métodos tienen mismo nombre pero los parametros son diferentes

POO : en resumen

Cualidades necesarias para un programa :

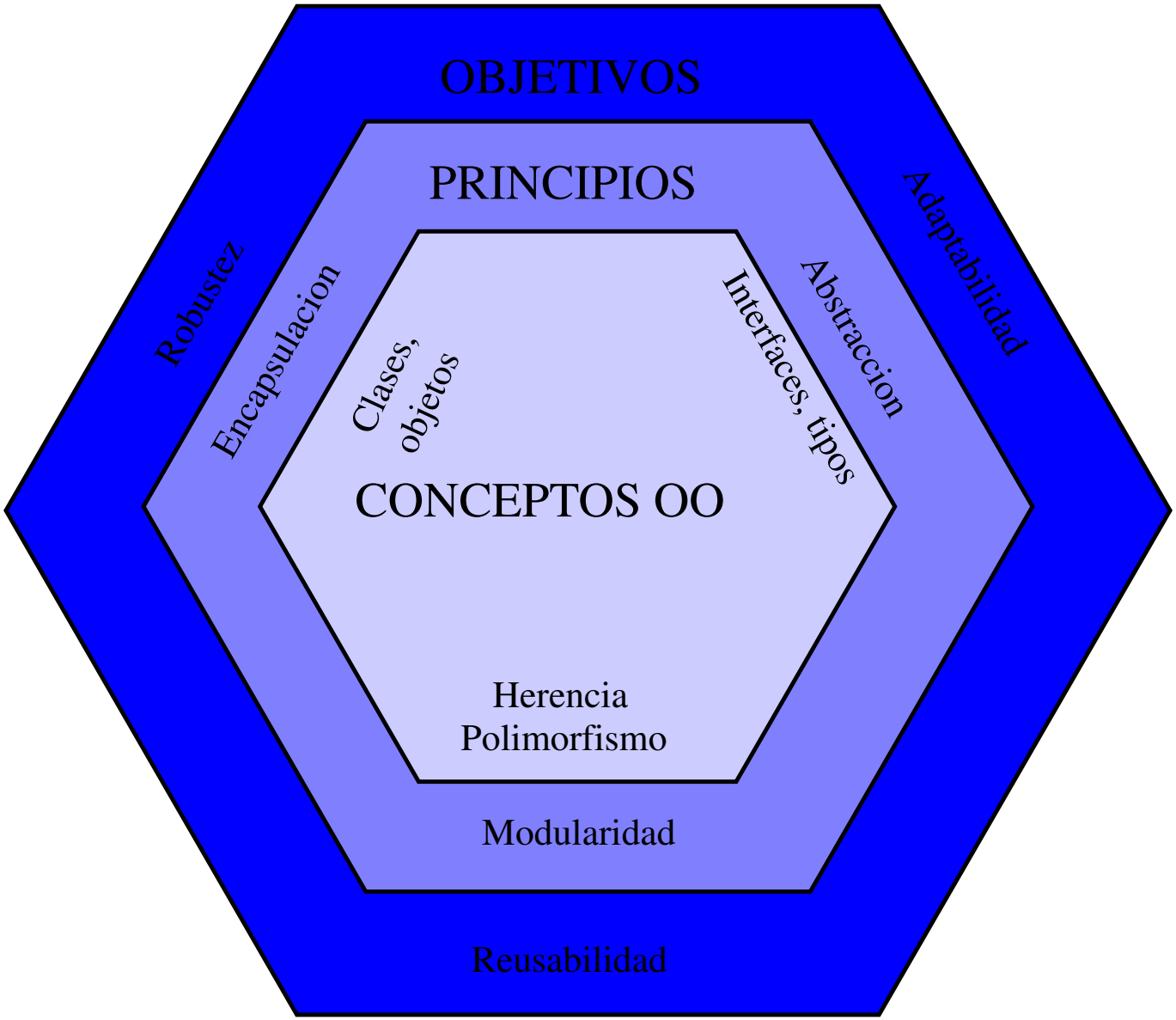
- Corrección
- Efficacidad
- Lisibilidad, concisión
- Robustez : entradas no previstas, limitaciones software o hardware...
- Adaptabilidad : portabilidad hacia otras arquitectura...
- Reutilización : necesita código suficientemente general, y claramente especificado...
- ...

POO : en resumen

Respuestas de la POO :

- Robustez → Encapsulación : Restringir el acceso a la representación interna; posibilidad de cambiar la implementación.
- Adaptabilidad → Abstracción : especificaciones de funcionalidad (interfaces : se puede conservarlas aunque la implementación cambie de un sistema a otro).
- Reutilización → Modularidad : división del software, jerarquización; reutilización por composición (objetos dentro de nuevas clases).
- ...

POO : en resumen



POO : en practica

Lenguajes permitiendo usar confortablemente el paradigma OO :

- C++
- C#
- Objective-C
- Java
- Eiffel
- Smalltalk

POO pura : SmallTalk

Tal vez el ejemplo de un lenguaje completamente OO. Todo es objeto en SmallTalk, hasta las estructura de control se construyen con mensajes pasados a objetos (booleanos)

```
|inter|
inter := Interval from:0 to:5 by:1.
inter do:[ :e | Transcript show:'ok '.].
```

POO : en resumen

Conceptos que recordar:

- clase, objeto, instancia,
- método, interfaces,
- herencia, polimorfismo,
- encapsulación, abstracción.

Proximas clases

El C++ ! *Thinking in C++*, Bruce Eckel

`http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`