UNIVERSIDAD DE GUADALAJARA

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

DEPARTAMENTO DE CIENCIAS COMPUTACIONALES

APUNTES DEL CURSO

PROGRAMACIÓN

ELABORADO POR

M. en Cs. Ing. Luis Alberto Muñoz Gómez

Actualización a Mayo 2022

Parte I. Programación Estructurada y Procedimental/Modular

Estructura de un programa C/C++

Conceptos de programa y programador

Algoritmo.- Conjunto finito y ordenado de instrucciones libres de ambigüedad y para resolver un problema

Programa.- Implementación de instrucciones basadas en un algoritmo, escritas en un lenguaje de programación y acompañadas de las estructuras de datos necesarias para cumplir con su objetivo

Programador.- Aquel (persona o software) que escribe un programa

La Codificación

Codificación.-Es la concretización de un programa en un código fuente interpretable

- hola mundo, cadena de caracteres ó constante de cadena
- formato del printf
- secuencias de escape: \n, \t, \", \\

Ciclo de vida del software: requerimientos, análisis, diseño, codificación, pruebas, implantación, explotación

Etapa de implantación: puesta en producción del software obtenido en la etapa de codificación y después de haber sido probado para detección de errores

Etapa de explotación: uso del software

Proceso.- Conjunto de tareas vinculadas ordenadamente que utilizan recursos para generar un producto

El entorno del compilador C/C++

- aplicación de consola
- nombre proyecto
- contenido plantilla
- librería
- compilar
- correr
- código fuente
- código objeto
- qué es un proyecto
- archivos generados
- modos insertar/sobrescribir
- guardar (ctrl+S), compilar (ctrl.+F9), correr (ctrl.+F10), compilar y correr (F9)

- marca (*) de archivo fuente modificado
- evitar los caracteres acentuados, ñ y otros “raros” en los nombres de archivos pues marca error
- código de página
- ctrl+z
- ctrl+y

Variables

Identificador

- nombre único (p. ej. retencionISR, a_1, etc.)
- camel case (p. ej. retencionISR o retencionIsr) iniciar en minúsculas
- sensible al caso
- no empezar con _ por las rutinas de biblioteca
- 31 caracteres significativos

Tipos de datos primitivos

- int entero de tamaño natural en la máquina; %d, %7d
- short int - entero corto
- long int - entero largo
- char - caracter de 1 byte; %c
- float, double; %3.0f, %6.1f, %6f, %.2f
- ANSI
- tamaños
- rangos
- signados y no signados
- enum boolean {NO, YES}

Tipo de dato	Cantidad de bytes	Combinaciones	Rango de Valores
	1	2 a la 8	-128 a 127
short int	2	2 a la 16	-32768 a 32767
Int	4	2 a la 32	-2147483648 a 2147483647
long int	8	2 a la 64	

Declaración y definición de datos Variables

- Variables contra constantes
- Solo declarar
- declarar una o varias
- en minúsculas

Inicialización de variables

- definir
- “evitar” declarar y definir al mismo tiempo
- Copiado en celdas de memoria y registros del procesador
- valor izquierdo

Reservación de memoria

- bits y bytes
- reservación de espacio
- imagen de bits en memoria

Constantes

Constantes const y #define

- hexadecimal
- const
- en mayúsculas
- #define, constantes simbólicas; **#define** *nombre texto de reemplazo*
- 12345U (sin signo)
- a=-0.1e-2f
- preprocesamiento para substitución de macros en el texto del programa

Entrada/Salida de datos

Escritura de datos

- printf
- cout

Entrada de datos

- getchar
- scanf
- cin
- valores por defecto a variables

Operadores y Expresiones

Aritméticos: +, -, *, /, división entera, módulo %
Incremento y decremento, antes y después, efectos colaterales: ++, --
Relacionales: >, >=, <, <=, ==, !=
Lógicos: &&, ||, !
De asignación: =, +=, -=, *=, /=
De condición: expr1?expr2:expr3

Prioridad de los operadores (p.58 Kernighan, Ritchie)

Operadores	Asociatividad
`() [] ->`	izquierda a derecha
`! ~ ++ -- + - * & (`*`tipo`*`) sizeof`	derecha a izquierda
`* / %`	izquierda a derecha
`+ -`	izquierda a derecha
`<< >>`	izquierda a derecha
`< <= > >=`	izquierda a derecha
`== !=`	izquierda a derecha
`&`	izquierda a derecha
`^`	izquierda a derecha
`|`	izquierda a derecha
`&&`	izquierda a derecha
`||`	izquierda a derecha
`?:`	derecha a izquierda
`= += -= *= /= %= &= ^= |= <<= >>=`	derecha a izquierda
`,`	izquierda a derecha

Expresiones

- los *operadores* especifican lo que se hará con las variables
- las *expresiones* se forman a partir de operadores y operandos; combinan variables y constantes para producir nuevos valores (generan un *valor izquierdo*)

Proposición: especificación de operaciones a realizar (ejemplo: de cálculo);

- una *proposición* puede ser:
- cualquier expresión,
- una asignación
- una llamada a función
- *proposición nula* ";"

Palabras Reservadas (p. 212 Kernighan, Ritchie)

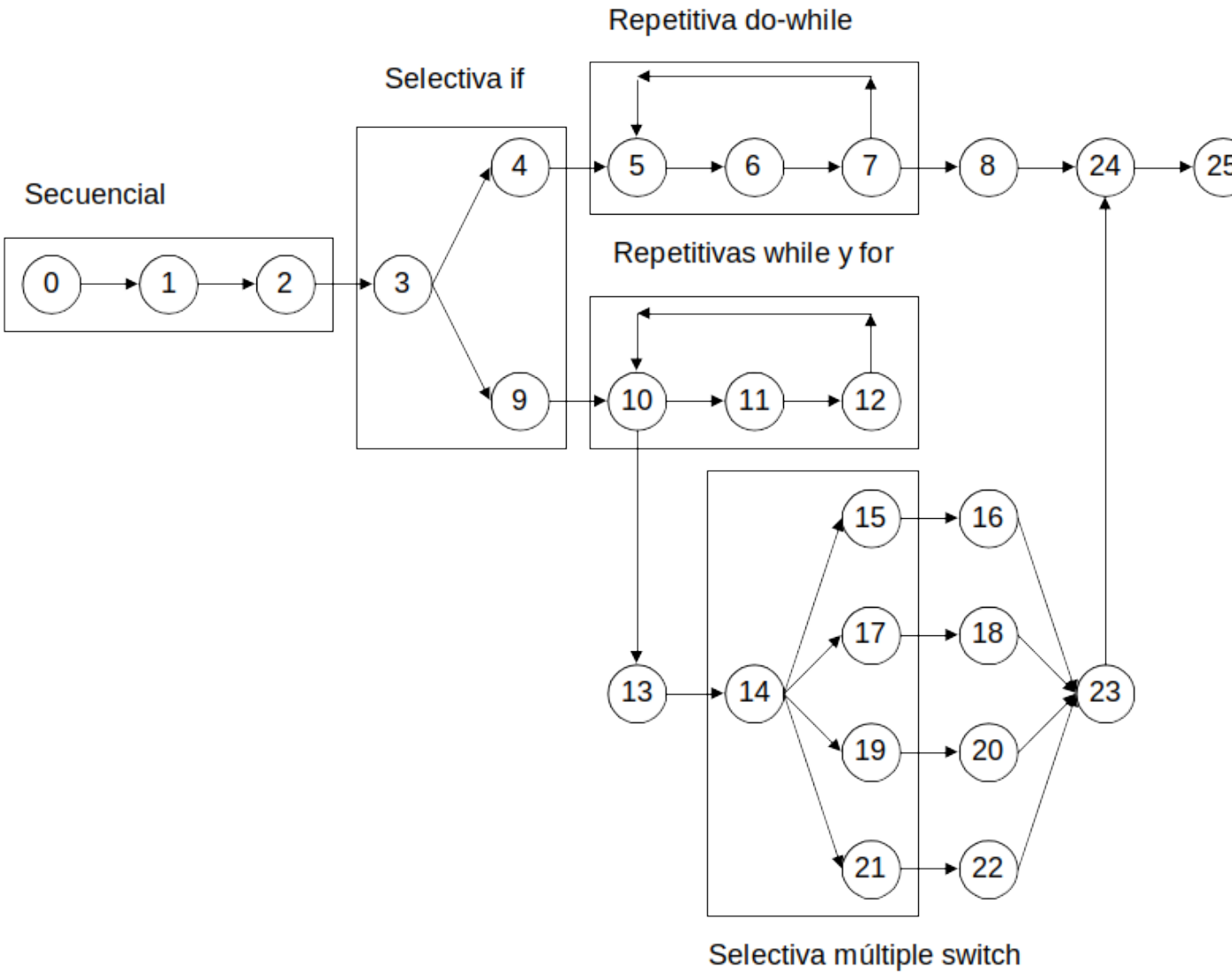
`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Comentarios

- explicación breve de lo que hace un programa
- una línea //
- al final de una línea //
- multilínea /* */
- útil también para depuración de un programa

- conversiones de tipo (casting)

Programación estructurada



Definición de Programación Estructurada

- Estilo de programación que permite la escritura de programas claros, fiables y eficientes mediante 3 estructuras de control de flujo: secuencial, condicional e iterativa y evitando el uso de instrucciones de transferencia incondicional (goto)

El Lenguaje C permite:

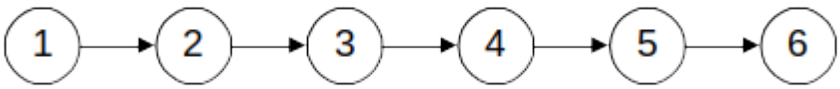
- control de flujo
- agrupación de proposiciones
- toma de decisiones (if-else)
- selección de un caso entre un conjunto de ellos (switch)
- iteración con la condición de paro en la parte superior (while, for) o en la parte inferior (do)

Estrategias de Diseño Descendente (Top-Down) y Ascendente (Botton-Up)

Ejemplo: Formación Académica

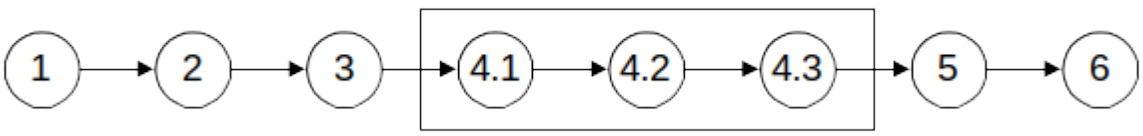
Estructura de control secuencial:

1 Cursar la Primaria
2 Cursar la Secundaria
3 Cursar la Preparatoria
4 Cursar la Licenciatura
5 Cursar la Maestría
6 Cursar el Doctorado



Diseño Descendente...

1 Cursar la Primaria
2 Cursar la Secundaria
3 Cursar la Preparatoria
4 Cursar la Licenciatura
  4.1 Cursar la carrera
  4.2 Graduarme de la carrera
  4.3 Titularme de la carrera
5 Cursar la Maestría
6 Cursar el Doctorado



Pseudocódigo:

```
<instruccion1>
<instruccion2>
<instruccion3>
```

Flujo de Programa (if y switch)

Estructura de control selectiva *if*

Pseudocódigo:

```
si <condición> entonces
inicio
  <instrucciones>
fin
```

En C/C++:

```
if (<condición>)
{
  <instrucciones>
}
```

Estructura de control selectiva *if-else*

Pseudocódigo:

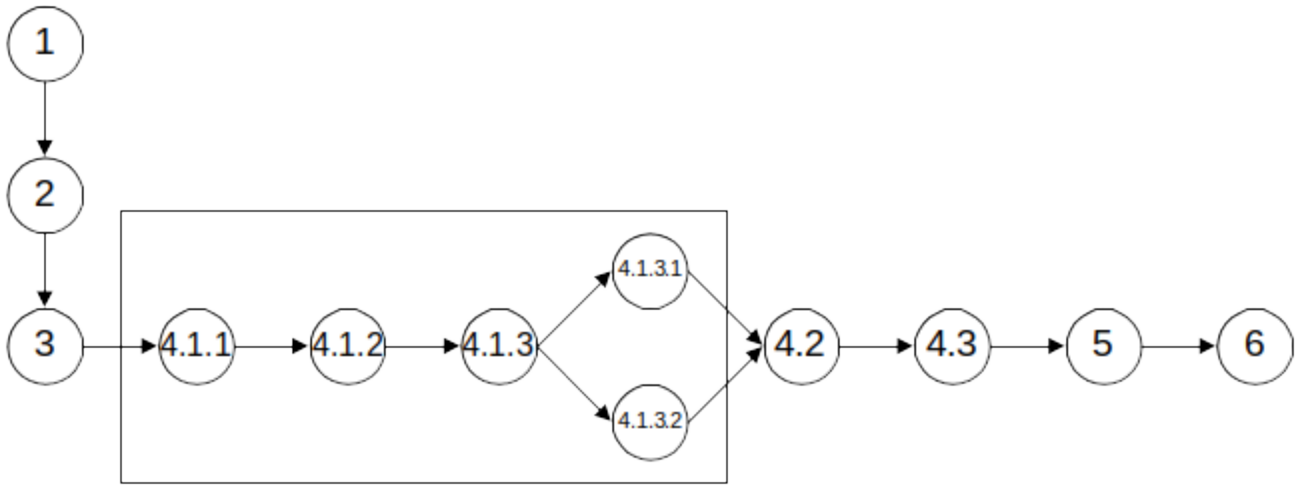
```
si <condición> entonces
inicio
  <instrucciones si verdadero>
fin
de lo contrario
inicio
  <instrucciones si falso>
fin
```

En C/C++:

```
if (<condición>)
{
  <instrucciones si verdadero>
}
else
{
  <instrucciones si falso>
}
```

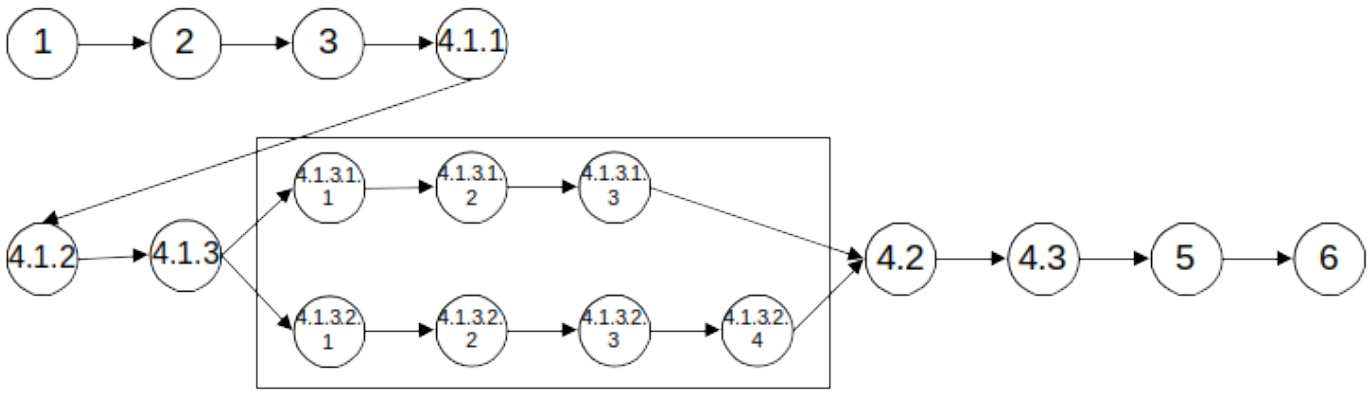
Ejemplo:

1 Cursar la Primaria
2 Cursar la Secundaria
3 Cursar la Preparatoria
4 Cursar la Licenciatura
4.1 Cursar la carrera
4.1.1 Seleccionar las 2 carreras que me más me gusten
4.1.2 Elegir una carrera (Ing. Química ó Ing. en Computación)
4.1.3 Verificar si estudiar Ingeniería Química *if(expresion){*
4.1.3.1 Estudiar Ingeniería Química
}
else{
4.1.3.2 Estudiar Ingeniería en Computación
}
4.2 Graduarme de la carrera
4.3 Titularme de la carrera
5 Cursar la Maestría
6 Cursar el Doctorado



Diseño descendente...

1 Cursar la Primaria
2 Cursar la Secundaria
3 Cursar la Preparatoria
4 Cursar la Licenciatura
  4.1 Cursar la carrera
    4.1.1 Seleccionar las 2 carreras que me más me gusten
    4.1.2 Elegir una carrera (Ing. Química ó Ing. en Computación)
    4.1.3 Verificar si estudiar Ingeniería Química *if(expresion){*
      4.1.3.1 Estudiar Ingeniería Química
        4.1.3.1.1 Cursar las materias más fáciles
        4.1.3.1.2 Cursar la materia más difícil
        4.1.3.1.3 Cursar las materias optativas
    }
    else{
      4.1.3.2 Estudiar Ingeniería en Computación
        4.1.3.2.1 Cursar las materias más fáciles
        4.1.3.2.2 Evaluar un posible descanso de la carrera
        4.1.3.2.3 Cursar la especialidad de la carrera
        4.1.3.2.4 Cursar las materias optativas
    }
  4.2 Graduarme de la carrera
  4.3 Titularme de la carrera
5 Cursar la Maestría
6 Cursar el Doctorado



Estructura de control selectiva *if-else if-else*

Pseudocódigo:

```
si <condición1> entonces
inicio
  <instrucciones si verdadera condición1>
fin
de lo contrario si <condición2> entonces
inicio
  <instrucciones si verdadera condición2>
fin
de lo contrario
inicio
  <instrucciones por defecto>
fin
```

En C/C++:

```
if (<condición1>)
{
  <instrucciones si verdadera condición1>
}
else if (<condición2>)
{
  <instrucciones si verdadera condición2>
}
else
{
  <instrucciones por defecto>
}
```

Estructura de control selectiva múltiple *switch*

Pseudocódigo:

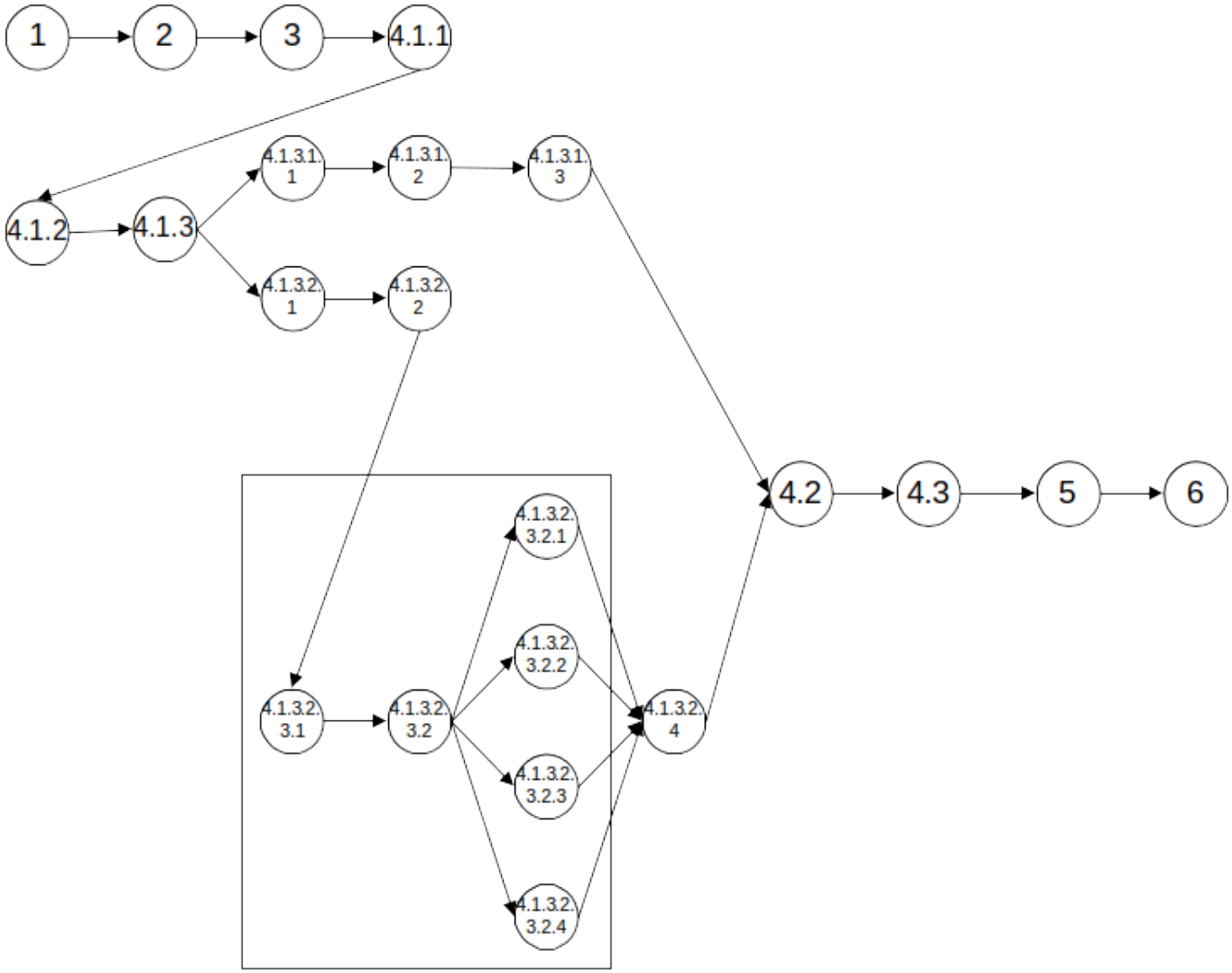
```
según sea <variable>
inicio
  caso <CONSTANTE_1>
    inicio
      <instrucciones si variable = CONSTANTE_1>
    fin
  caso <CONSTANTE_2>
    inicio
      <instrucciones si variable = CONSTANTE_2>
    fin
  <...más casos...>
  por defecto
    inicio
      <instrucciones por defecto>
    fin
fin
```

En C/C++:

```
switch(<variable>)
{
  case <CONSTANTE_1>:
    <instrucciones si variable = CONSTANTE_1>
    break;
  case <CONSTANTE_2>:
    <instrucciones si variable = CONSTANTE_2>
    break;
  <...más casos...>
  default:
    <instrucciones por defecto>
}
```

Ejemplo:

1 Cursar la Primaria
2 Cursar la Secundaria
3 Cursar la Preparatoria
4 Cursar la Licenciatura
  4.1 Cursar la carrera
    4.1.1 Seleccionar las 2 carreras que me más me gusten
    4.1.2 Elegir una carrera (Ing. Química ó Ing. en Computación)
    4.1.3 Verificar si estudiar Ingeniería Química *if(expresion){*
      4.1.3.1 Estudiar Ingeniería Química
        4.1.3.1.1 Cursar las materias más fáciles
        4.1.3.1.2 Cursar la materia más difícil
        4.1.3.1.3 Cursar las materias optativas
    }
    else{
      4.1.3.2 Estudiar Ingeniería en Computación
        4.1.3.2.1 Cursar las materias más fáciles
        4.1.3.2.2 Evaluar un posible descanso de la carrera
        4.1.3.2.3 Cursar la especialidad de la carrera
          4.1.3.2.3.1 Elegir una de las especialidades
          4.1.3.2.3.2 Verificar por la opción a seguir *switch(var){*
            4.1.3.2.3.2.1 Cursar por Software de Sistemas
            4.1.3.2.3.2.2 Cursar por Sistemas Computacionales
            4.1.3.2.3.2.3 Cursar por Sistemas Digitales
            4.1.3.2.3.2.4 Cursar por Sistemas de Información
          }
        4.1.3.2.4 Cursar las materias optativas
    }
  4.2 Graduarme de la carrera
  4.3 Titularme de la carrera
5 Cursar la Maestría
6 Cursar el Doctorado


1
2
3
4.1.1
4.1.2
4.1.3
4.1.3.1.1
4.1.3.1.2
4.1.3.1.3
4.1.3.2.1
4.1.3.2.2
4.1.3.2.3.1
4.1.3.2.3.2
4.1.3.2.3.2.1
4.1.3.2.3.2.2
4.1.3.2.3.2.3
4.1.3.2.3.2.4
4.1.3.2.4
4.2
4.3
5
6

Pseudocódigo

```
anio<-2000
anio/400
multiplo de 400 --- (anio%400)==0

si (es un año múltiplo de 400) entonces{
  es un año bisiesto
}
sino si ((es un año múltiplo de 4) y
         (no es un año múltiplo de 100)) entonces{
  es un año bisiesto
}
sino{
  no es un año bisiesto
}

si ((es un año múltiplo de 400) ó
    ((es un año múltiplo de 4) y (no es un año múltiplo de
100))){
  es un año bisiesto
}
sino{
  no es un año bisiesto
}
ó es ||
y es &&
```

Lógica Proposicional

multi4 = ”el año es un múltiplo de 4” //   =(anio%4)==0
multi100 = ”el año es un múltiplo de 100”
multi400 = ”el año es un múltiplo de 400”
esFebrero = mes==2
diaEntreUnoVeintiocho = (1<=dia) && (dia<=28)
dia29 = dia==29

((multi4 Λ ¬multi100) V multi400) => anioBisiesto

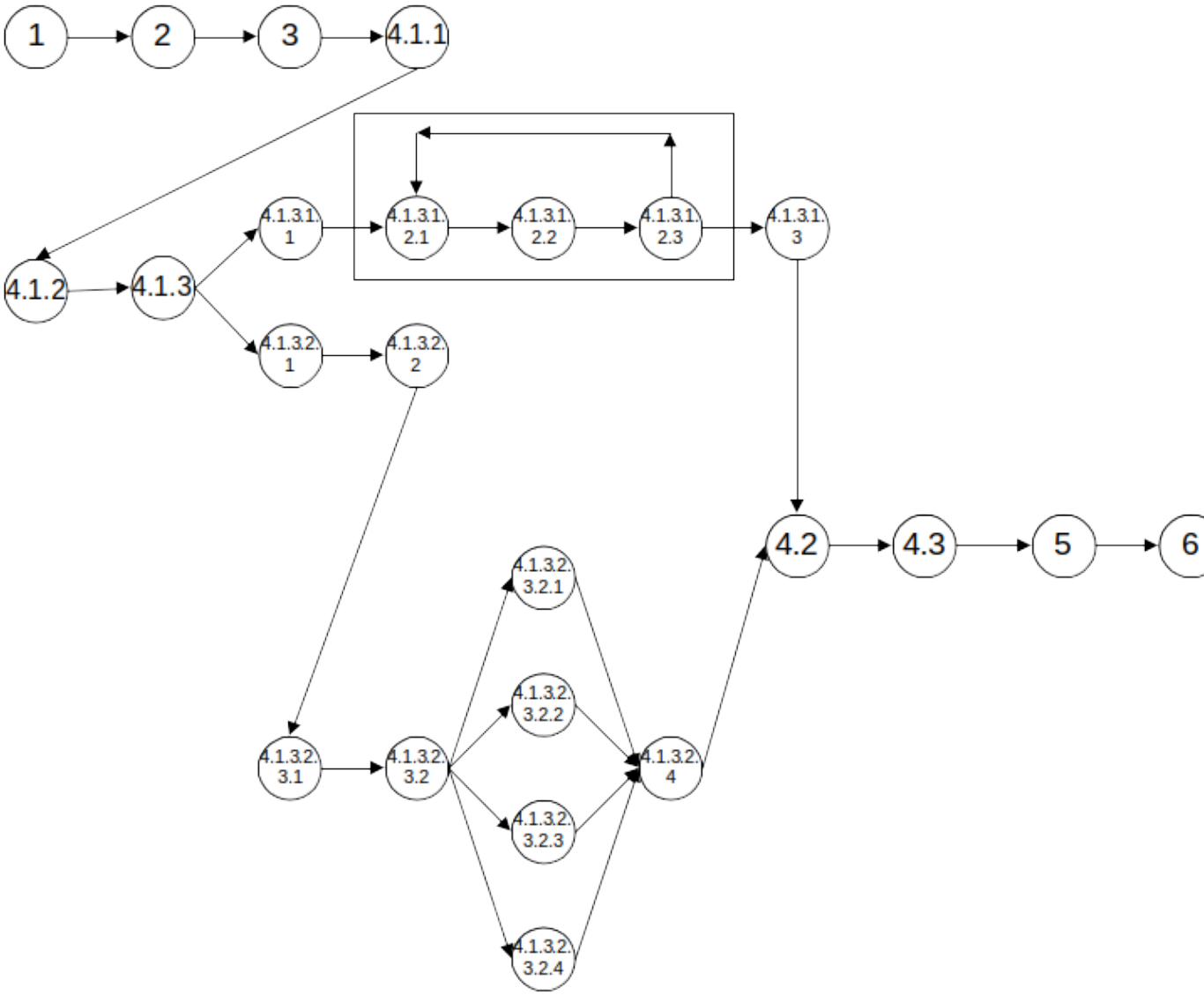
(esFebrero Λ (diaEntreUnoVeintiocho V (dia29 Λ anioBisiesto))) => fechaValida

Λ en C es &&
V en C es ||
=> en C es un if (...){…}

Flujo de Programa (do-while, while y for)

Estructura de control iterativa con condición de paro en la parte inferior (verificación al final) *do-while*

1 Cursar la Primaria
2 Cursar la Secundaria
3 Cursar la Preparatoria
4 Cursar la Licenciatura
  4.1 Cursar la carrera
    4.1.1 Seleccionar las 2 carreras que me más me gusten
    4.1.2 Elegir una carrera (Ing. Química ó Ing. en Computación)
    4.1.3 Verificar si estudiar Ingeniería Química *if(expresion){*
      4.1.3.1 Estudiar Ingeniería Química
        4.1.3.1.1 Cursar las materias más fáciles
        4.1.3.1.2 Cursar la materia más difícil
          4.1.3.1.2.1 Iniciar *do{*
          4.1.3.1.2.2 Evaluar mi desempeño en el curso
          4.1.3.1.2.3 Verificar si cursar de nuevo *}while(expr)*
        4.1.3.1.3 Cursar las materias optativas
    }
    else{
      4.1.3.2 Estudiar Ingeniería en Computación
        4.1.3.2.1 Cursar las materias más fáciles
        4.1.3.2.2 Evaluar un posible descanso de la carrera
        4.1.3.2.3 Cursar la especialidad de la carrera
          4.1.3.2.3.1 Elegir una de las especialidades
          4.1.3.2.3.2 Verificar por la opción a seguir *switch(var){*
            4.1.3.2.3.2.1 Cursar por Software de Sistemas
            4.1.3.2.3.2.2 Cursar por Sistemas Computacionales
            4.1.3.2.3.2.3 Cursar por Sistemas Digitales
            4.1.3.2.3.2.4 Cursar por Sistemas de Información
          }
        4.1.3.2.4 Cursar las materias optativas
    }
  4.2 Graduarme de la carrera
  4.3 Titularme de la carrera
5 Cursar la Maestría
6 Cursar el Doctorado



Contadores, Acumuladores y Banderas

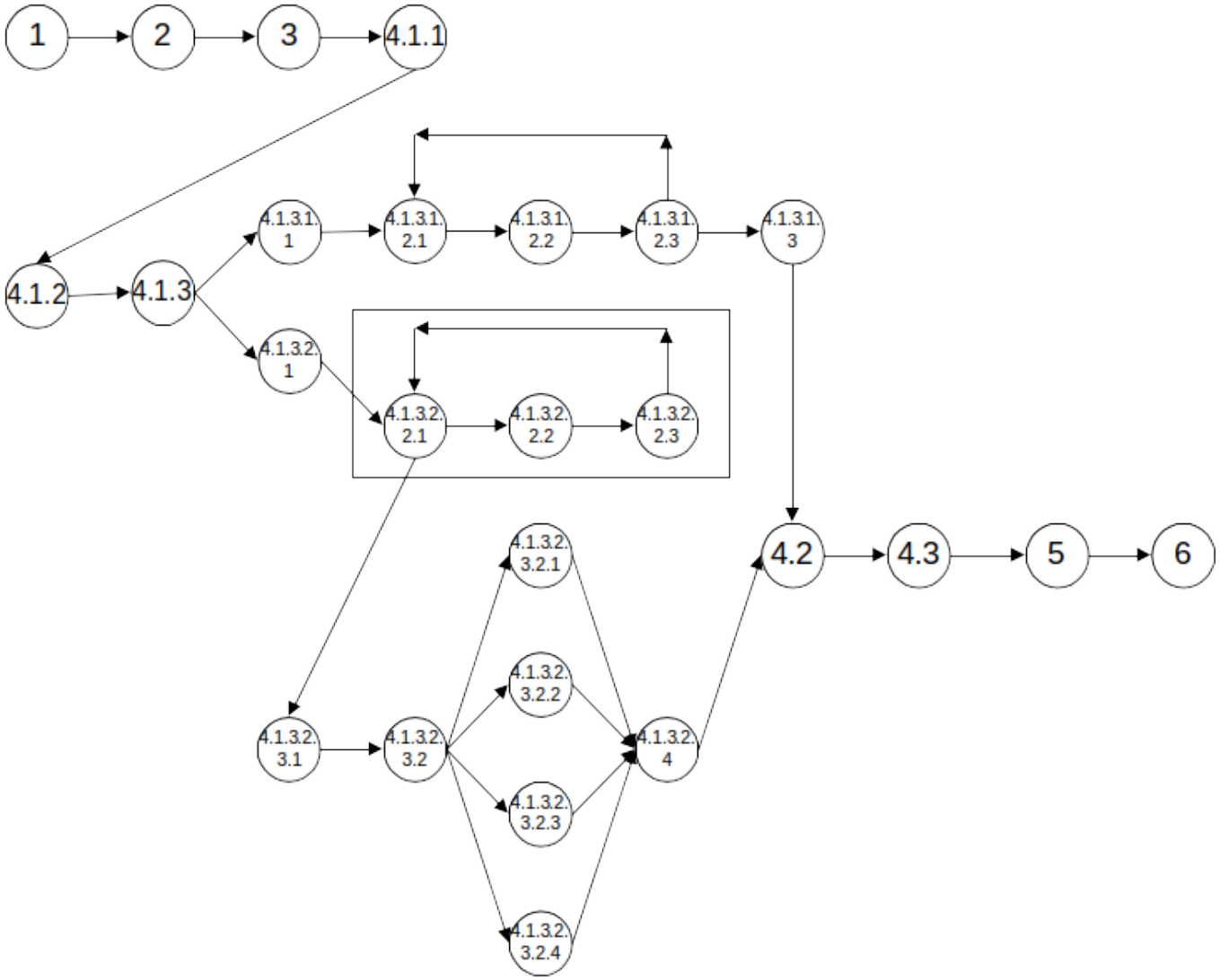
Contador.- Variable entera cuya función es contar de 1 en 1 ya sea de forma incremental ó decremental; generalmente se usa al interior de una estructura de control iterativa

Acumulador.-Variable de tipo entero ó flotante que sirve para almacenar de forma incremental distintos valores

Bandera.-variable (en C es de tipo entero en C++ tipo bool) que puede asumir un valor de verdad (verdadero ó falso) y que representa ya sea la realización de un bloque de instrucciones ó bien la continuidad en la realización del mismo, según sea la estructura de control a la que se le asocie

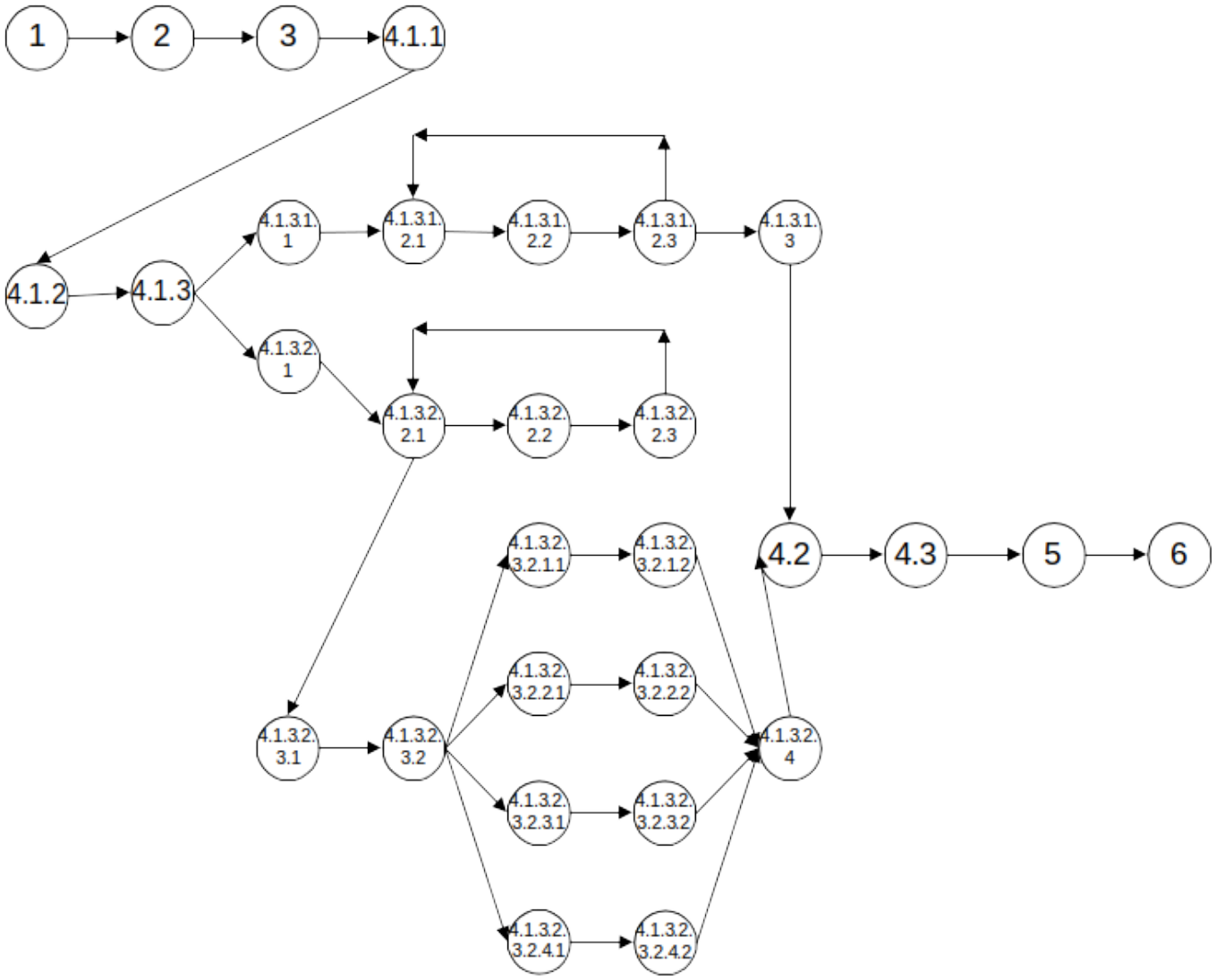
Estructura de control iterativa con condición de paro en la parte superior (verificación al inicio) *while ó for*

1 Cursar la Primaria
2 Cursar la Secundaria
3 Cursar la Preparatoria
4 Cursar la Licenciatura
  4.1 Cursar la carrera
    4.1.1 Seleccionar las 2 carreras que me más me gusten
    4.1.2 Elegir una carrera (Ing. Química ó Ing. en Computación)
    4.1.3 Verificar si estudiar Ingeniería Química *if(expresion){*
      4.1.3.1 Estudiar Ingeniería Química
        4.1.3.1.1 Cursar las materias más fáciles
        4.1.3.1.2 Cursar la materia más difícil
          4.1.3.1.2.1 Iniciar *do{*
          4.1.3.1.2.2 Evaluar mi desempeño en el curso
          4.1.3.1.2.3 Verificar si cursar de nuevo *}while(expr)*
        4.1.3.1.3 Cursar las materias optativas
    }
    else{
      4.1.3.2 Estudiar Ingeniería en Computación
        4.1.3.2.1 Cursar las materias más fáciles
        4.1.3.2.2 Evaluar un posible descanso de la carrera
          4.1.3.2.2.1 Verificar si falto un semestre *while(expr){*
          4.1.3.2.2.2 Trabajar un tiempo
          4.1.3.2.2.3 Regularizarme en conocimientos necesarios
          }
        4.1.3.2.3 Cursar la especialidad de la carrera
          4.1.3.2.3.1 Elegir una de las especialidades
          4.1.3.2.3.2 Verificar por la opción a seguir *switch(var){*
            4.1.3.2.3.2.1 Cursar por Software de Sistemas
            4.1.3.2.3.2.2 Cursar por Sistemas Computacionales
            4.1.3.2.3.2.3 Cursar por Sistemas Digitales
            4.1.3.2.3.2.4 Cursar por Sistemas de Información
          }
        4.1.3.2.4 Cursar las materias optativas
    }
  4.2 Graduarme de la carrera
  4.3 Titularme de la carrera
5 Cursar la Maestría
6 Cursar el Doctorado



...y el programa completo quedaría con algo más de diseño descendente al interior del switch...

1 Cursar la Primaria
2 Cursar la Secundaria
3 Cursar la Preparatoria
4 Cursar la Licenciatura
4.1 Cursar la carrera
4.1.1 Seleccionar las 2 carreras que me más me gusten
4.1.2 Elegir una carrera (Ing. Química ó Ing. en Computación)
4.1.3 Verificar si estudiar Ingeniería Química *if(expresion){*
4.1.3.1 Estudiar Ingeniería Química
4.1.3.1.1 Cursar las materias más fáciles
4.1.3.1.2 Cursar la materia más difícil
4.1.3.1.2.1 Iniciar *do{*
4.1.3.1.2.2 Evaluar mi desempeño en el curso
4.1.3.1.2.3 Verificar si cursar de nuevo *}while(expr)*
4.1.3.1.3 Cursar las materias optativas
}
else{
4.1.3.2 Estudiar Ingeniería en Computación
4.1.3.2.1 Cursar las materias más fáciles
4.1.3.2.2 Evaluar un posible descanso de la carrera
4.1.3.2.2.1 Verificar si falto un semestre *while(expr){*
4.1.3.2.2.2 Trabajar un tiempo
4.1.3.2.2.3 Regularizarme en conocimientos necesarios
}
4.1.3.2.3 Cursar la especialidad de la carrera
4.1.3.2.3.1 Elegir una de las especialidades
4.1.3.2.3.2 Verificar por la opción a seguir *switch(var){*
4.1.3.2.3.2.1 Cursar por Software de Sistemas
4.1.3.2.3.2.1.1 Cursar las materias más fáciles
4.1.3.2.3.2.1.2 Cursar la materia más difícil
4.1.3.2.3.2.2 Cursar por Sistemas Computacionales
4.1.3.2.3.2.2.1 Cursar las materias más fáciles
4.1.3.2.3.2.2.2 Cursar la materia más difícil
4.1.3.2.3.2.3 Cursar por Sistemas Digitales
4.1.3.2.3.2.3.1 Cursar las materias más fáciles
4.1.3.2.3.2.3.2 Cursar la materia más difícil
4.1.3.2.3.2.4 Cursar por Sistemas de Información
4.1.3.2.3.2.4.1 Cursar las materias más fáciles
4.1.3.2.3.2.4.2 Cursar la materia más difícil
}
4.1.3.2.4 Cursar las materias optativas
}
4.2 Graduarme de la carrera
4.3 Titularme de la carrera
5 Cursar la Maestría
6 Cursar el Doctorado


1
2
3
4.1.1
4.1.2
4.1.3
4.1.3.1.1
4.1.3.1.2.1
4.1.3.1.2.2
4.1.3.1.2.3
4.1.3.1.3
4.1.3.2.1
4.1.3.2.2.1
4.1.3.2.2.2
4.1.3.2.2.3
4.1.3.2.3.1
4.1.3.2.3.2
4.1.3.2.3.3
4.1.3.2.3.4
4.1.3.2.4.1
4.1.3.2.4.2
4.1.3.2.4.3
4.1.3.2.4.4
4.2
4.3
5
6

Arreglos

Definiciones

Estructura de Datos.- Colección de datos organizados de un modo particular; pueden ser de dos tipos:

a) estáticas: son aquellas en las que se asigna una cantidad fija de memoria cuando se declara la variable
b) dinámicas: son aquellas cuya ocupación de memoria puede aumentar y disminuir en tiempo de ejecución a medida que el programa progresa

Arreglo.-Estructura de datos en la que se almacena una colección de datos del mismo tipo; ejemplo: los salarios de los empleados de una empresa. Es una lista de un número finito de n elementos del mismo tipo que se caracteriza por:

1) almacenar los elementos del array en posiciones de memoria continua
2) tener un único nombre de variable; ejemplo: salarios

Vectores

- Arreglos de una dimensión

Matrices

- Arreglos de dos dimensiones

Programación Procedimental/Modular: Funciones

Qué son

- Módulos de programa
- Pueden regresar valores de tipos básicos, estructuras, uniones o apuntadores
- Las *funciones* contienen proposiciones encerradas todas entre llaves

- Funciones sin parámetros
- Prototipos de funciones
- Funciones con parámetros por valor
- Funciones predefinidas (matemáticas y de cadena)

Declaración de Variables

- Variables locales automáticas
- Declaradas en modalidad estructurada por bloques
- Pueden ser internas a una función o visibles al programa completo

Recursividad

Es la forma en la cual se especifica un proceso basado en su propia definición.

Factorial:

$$f(n)=\begin{cases} n=0: 1 \\ n>0: n*f(n-1) \\ \text{otro: indefinido} \end{cases}$$

Sucesión de Fibonacci:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \text{ para } n = 2, 3, 4, 5, \ldots$$

Apuntadores

Apuntador.-Variable que contiene la dirección de memoria donde se almacena alguna información

Estructuras

Tipo de Dato Estructurado.-Colección de 2 o más tipos de datos primitivos que generan un nuevo tipo de abstracción: Persona(código, nombre, rfc, salario)

Parte II. Programación Orientada a Objetos

Strings

string.-Tipo de dato que representa a una secuencia de caracteres, también utilizado como "constante de cadena".

Introducción a la Programación Orientada a Objetos

Programación Procedimental contra la Orientada a Objetos

Programación Procedimental

El paradigma de programación original (y probablemente aun el más comúnmente usado) es:

- Decidir cuales procedimientos quieres; usar los mejores algoritmos que puedas encontrar

El foco está en el procesamiento, el algoritmo necesario para ejecutar la computación deseada.

Programación Orientada a Objetos

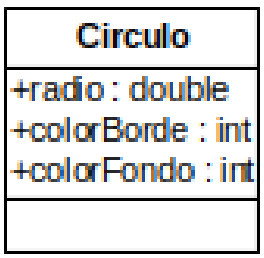
El paradigma de programación es:

- Decidir cuales clases deseas; proveer un conjunto completo de operaciones para cada clase; vuelve explícito lo común mediante el uso de la herencia

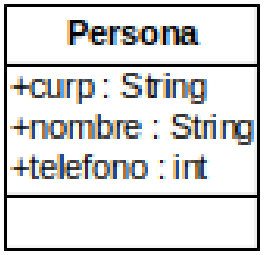
Introducción al Pseudocódigo Orientado a Objetos

Abstracción

```
Circulo{
  radio:real
  colorBorde:entero
  colorFondo:entero
}
```



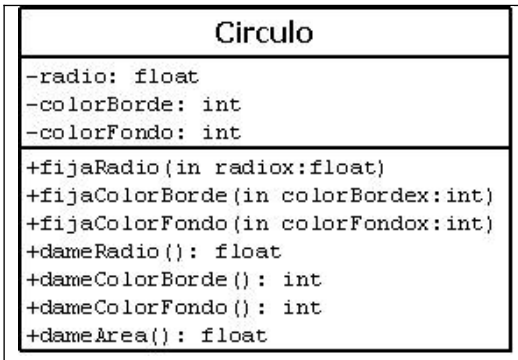

```
class Circulo{ //como registro
    public:
        double radio;
        int colorBorde;
        int colorFondo;
};

int main()
{
    Circulo c;
    c.radio=10;
    c.colorBorde=11;
    c.colorFondo=14;
    cout << "radio es " << c.radio << endl;
    cout << "colorBorde es " << c.colorBorde << endl;
    cout << "colorFondo es " << c.colorFondo << endl;
    return 0;
}
```



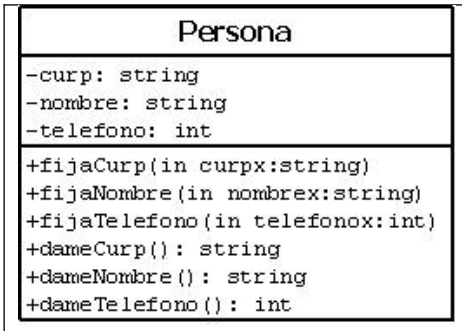

```
class Persona{ //como registro
    public:
        string curp;
        string nombre;
        int telefono;
};

int main()
{
    Persona p;
    p.curp="MUGL790912IC7";
    p.nombre="Luis Alberto";
    p.telefono=1234567;
    cout << "curp es " << p.curp << endl;
    cout << "nombre es " << p.nombre << endl;
    cout << "telefono es " << p.telefono << endl;
    return 0;
}
```

Objetos y Clases

```
Circulo{
  radio:real
  colorBorde:entero
  colorFondo:entero
  PI:constante<-3.1416

  fijaRadio(radiox){
    radio<-radiox
  }
  fijaColorBorde(colorBordex){
    colorBorde<-colorBordex
  }
  fijaColorFondo(colorFondox){
    colorFondo<-colorFondox
  }
  <-dameRadio(){
    regresa radio
  }
  <-dameColorBorde(){
    regresa colorBorde
  }
  <-dameColorFondo(){
    regresa colorFondo
  }
  <-dameArea(){
    regresa PI*radio*radio
  }
}
```




```
class Circulo{
    static const float PI=3.1416;
    double radio;
    int colorBorde;
    int colorFondo;
public:
    void fijaRadio(double radiox){
        radio=radiox;
    }
    void fijaColorBorde(int colorBordex){
        colorBorde=colorBordex;
    }
    void fijaColorFondo(int colorFondox){
        colorFondo=colorFondox;
    }
    double dameRadio(){
        return radio;
    }
    int dameColorBorde(){
        return colorBorde;
    }
    int dameColorFondo(){
        return colorFondo;
    }
    float dameArea(){
        return PI*radio*radio;
    }
};

int main()
{
    Circulo c;
    c.fijaRadio(10);
    c.fijaColorBorde(11);
    c.fijaColorFondo(14);
    cout << "radio es " << c.dameRadio() << endl;
    cout << "colorBorde es " << c.dameColorBorde() << endl;
    cout << "colorFondo es " << c.dameColorFondo() << endl;
    cout << "area es " << c.dameArea() << endl;
    return 0;
}
```




```
class Persona{ //como clase
    string curp;
    string nombre;
    int telefono;
public:
    void fijaCurp(string curpx){
        curp=curpx;
    }
    void fijaNombre(string nombrex){
        nombre=nombrex;
    }
    void fijaTelefono(int telefonox){
        telefono=telefonox;
    }
    string dameCurp(){
        return curp;
    }
    string dameNombre(){
        return nombre;
    }
    int dameTelefono(){
        return telefono;
    }
};

int main()
{
    Persona p;
    p.fijaCurp("MUGL790912IC7");
    p.fijaNombre("Luis Alberto");
    p.fijaTelefono(1234567);
    cout << "curp es " << p.dameCurp() << endl;
    cout << "nombre es " << p.dameNombre() << endl;
    cout << "telefono es " << p.dameTelefono() << endl;
    return 0;
}
```

Ocultamiento de Información

El *ocultamiento de información* permite evitar que el código de una *aplicación cliente* tenga acceso a los datos de una *clase* que bien pueden requerir un trato cuidadoso, así cualquier solicitud para modificar o consultar el estado de un *objeto* deba ser mediante el uso de *métodos de interfaz* (públicos) y bajo las restricciones que en ellos se especifiquen.

Instanciación

Creación, inicialización y limpiado de objetos

- invocación a métodos setters

Fortalezas y debilidades de la POO

Fortalezas:

- Modelado en la dimensión estructural de los objetos
- Modelado semántico entidad-relacionamiento
- Identificación y especificación de objetos, clases, métodos, atributos, asociaciones dependientes de dominio
- Jerarquías de herencia para reutilización de código

Debilidades:

- Criterios consistentes de particionamiento de la complejidad. Deben existir criterios objetivos y prácticos para agregar clases o particionar sistemas
- Reutilización de la especificación. La máxima reutilización ocurre al interior de dominios específicos de aplicación.
- Modelado funcional vs end-to-end. Por la descomposición funcional top-down, las funciones se deben subordinar a los objetos.
- Validación del usuario en el modelado del dominio del problema
- Estimación o dimensionamiento de los sistemas. Está bajo investigación.

Introducción al modelado Orientado a Objetos y el Lenguaje Unificado de Modelado (UML)

UML es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software. Se usa para entender, diseñar, configurar, mantener y controlar la información sobre los sistemas a construir.

Breve historia de UML (Unified Modelling Language)

UML nació como una notación estándar de la construcción de modelos. UML comenzó a gestarse en Octubre de 1994. UML es el resultado de la fusión de dos métodos, el OMT de Rumbaugh y el método de Booch para diseño de software orientado a objetos. El primer borrador nació en Octubre de 1995.

Quién usa UML?

Problemas complejos requieren particularmente suficiente planeación para evitar problemas durante el desarrollo y más en las etapas de mantenimiento y actualización donde los cambios son más arduos y costosos. UML permite a personas de diferentes disciplinas, trabajar juntos en identificar y resolver problemas antes de que ocurran.

Los desarrolladores y analistas de negocios pueden mapear los requerimientos en un lenguaje. No hay necesidad de que consideren la tecnología subyacente en esta etapa, que permite a no-programadores estar más envueltos en el proceso de desarrollo.

UML es utilizado en sistemas complejos para capturar no solo la información del software orientado a objetos, sino también las reglas de negocio. UML es una elección natural como lenguaje independiente de plataforma.

Muchos desarrolladores solo usan la aproximación de “solo código”. Esto causa severos problemas cuando el sistema se expande o los desarrolladores originales ya no trabajan más en el proyecto.

Ventajas

1. Es un lenguaje formal, cada elemento tiene un significado bien definido de modo que no se presta a malentendidos.
2. Es conciso, dado que el lenguaje tiene una notación, simple y sencilla
3. Es comprensible, por describir lo más importante del sistema
4. Es escalable, por ser suficientemente formal para manejar proyectos de modelado masivo de sistemas masivos así como pequeños proyectos, evitando excesos
5. Se construyó sobre lecciones aprendidas, por ser la culminación de las mejores prácticas de la comunidad orientada a objetos
6. Es el estándar, dado que UML es controlado por un grupo de manejadores de estándares con contribuciones activas de un grupo mundial de proveedores y

académicos. El estándar asegura la habilidad de transformación de UML y su interoperabilidad, a modo de no estar casado con un producto particular.

Objetivos

- Ofrecer a programadores un lenguaje de modelado visual listo para usarse que ayude en el desarrollo e intercambio de modelos significativos
- Independencia de lenguajes de programación y procesos de desarrollo
- Proporcionar mecanismos de extensión y especialización
- Proporcionar una base formal para entender el lenguaje de modelado
- Fomentar el crecimiento en el mercado las herramientas para orientado a objetos
- Soportar conceptos de desarrollo de alto nivel como pueden ser colaboraciones, frameworks, patterns y componentes
- Integrar las mejores prácticas utilizadas hasta el momento

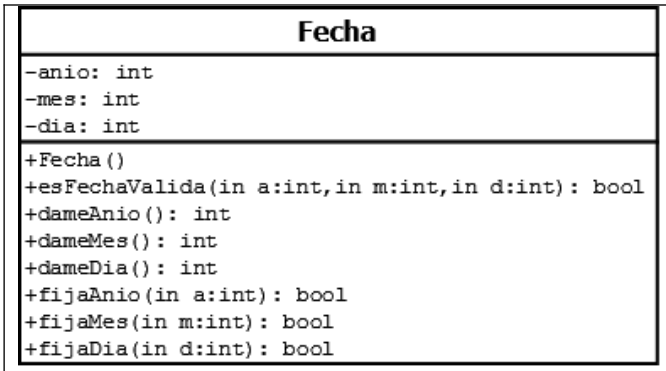
Apuntador this →

```
class Persona{
…
   fijaNombre(string nombre){
      this→nombre=nombre;
   }
…
}
```

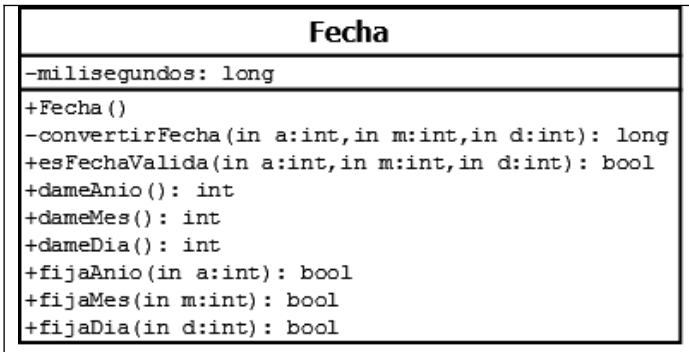
Encapsulamiento

El *encapsulamiento* permite que ocultemos los detalles de implementación de una clase y que si necesitamos mejorar su funcionamiento, siempre y cuando se mantengan sin cambios sus *métodos de interfaz*, las aplicaciones cliente continuarán funcionando correctamente.


```
class Fecha{  //Fecha como registro
public:
    int dia;
    int mes;
    int anio;
}

//el problema: libre acceso del exterior
int main() {
    Fecha f;
    //asignarle mes y año
    f.dia=31;  //es posible, pero de qué mes?
    cout << "dia=" << f.dia;
}
```



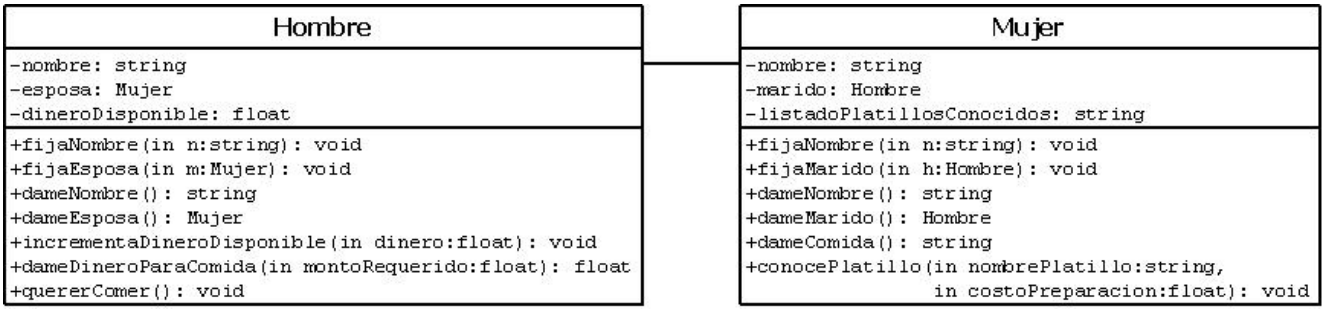

```
class Fecha{  //Fecha como clase
  int dia;
  int mes;
  int anio;
public:
  bool esFechaValida(int a,int m,int d){
    //pendiente implementar
  }
  bool fijaDia(int d){
    dia=d;    //requiere validar antes
    return true;//no siempre es válido
  }
  int dameDia(){
    return dia;
  }
  //completar otros dame() y fija()
}

//la solución
int main() {
  Fecha f;
  //f.dia=31; //no compila por ocultamiento
  f.fijaDia(31);
  cout << "dia=" << f.dameDia();
  return 0;
}
```




```
class Fecha{  //Fecha como clase
  int dia;
  int mes;
  int anio;
  long convertirFecha(int a,int m,int d){
    //pendiente implementar
  }
public:
  bool esFechaValida(int a,int m,int d){
    //pendiente implementar
  }
  bool fijaDia(int d){
    dia=d;    //requiere validar antes
    return true;//no siempre es válido
  }
  int dameDia(){
    return dia;
  }
  //completar otros dame() y fija()
}

//la solución
int main() {
  Fecha f;
  //f.dia=31; //no compila por ocultamiento
  f.fijaDia(31);
  cout << "dia=" << f.dameDia();
  return 0;
}
```

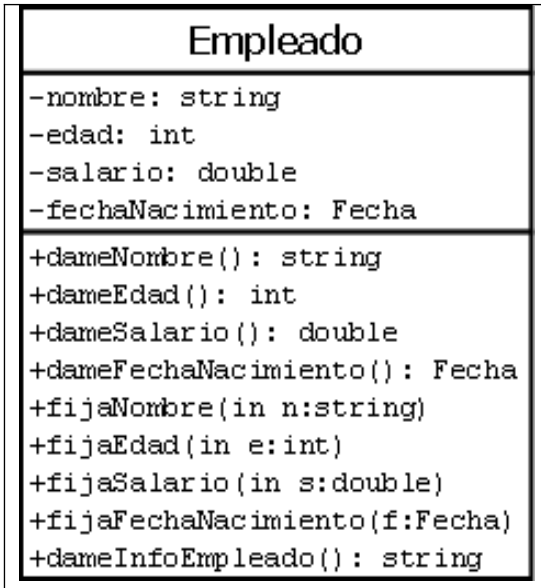
Comunicación entre objetos

Comunicación entre dos objetos



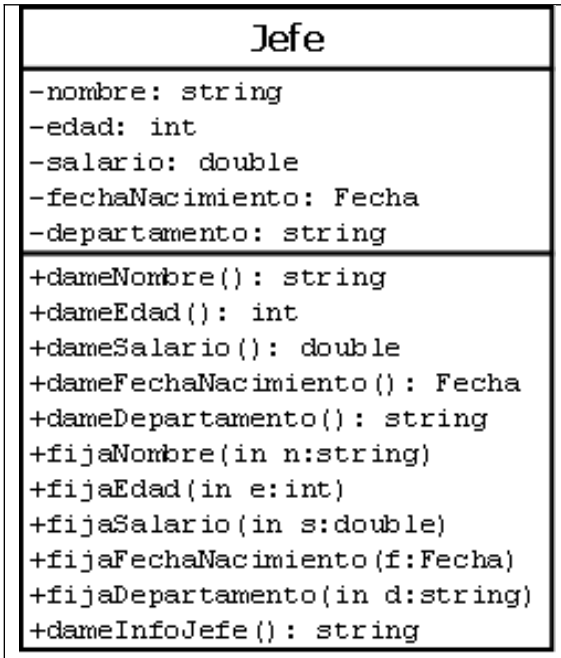
Herencia

Abstracción base:



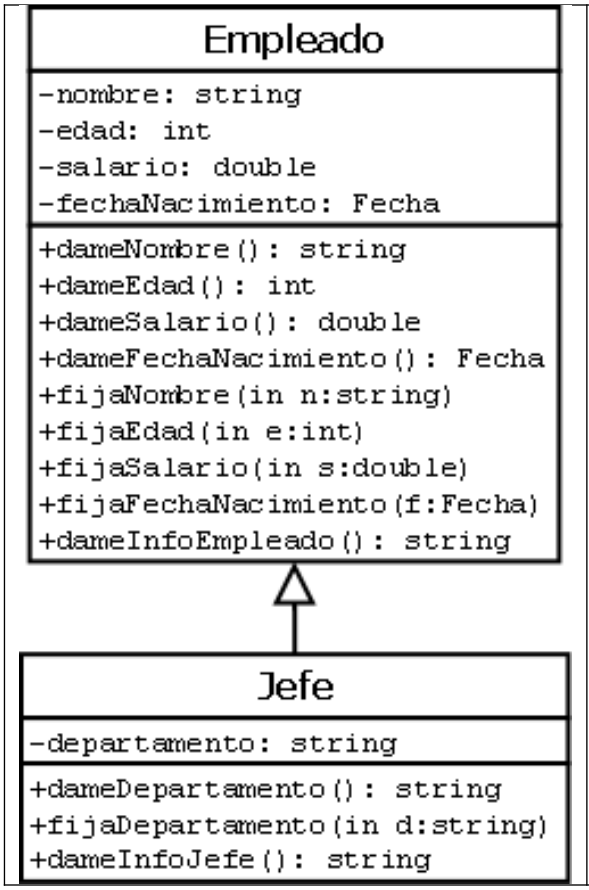

```
class MiFecha{
public:
    int anio;
    int mes;
    int dia;
};

class Empleado{
    string nombre;
    int edad;
    double salario;
    Fecha fechaNacimiento;
public:
  Empleado(string nombre, int edad, double salario){
      this->nombre = nombre;
      this->edad = edad;
      this->salario=salario;
      fechaNacimiento.anio=1979;
      fechaNacimiento.mes=9;
      fechaNacimiento.dia=12;
  }
  string dameInfoEmpleado(){
      stringstream info;
      info << "Nombre:" << nombre <<
          "\nEdad:" << edad <<
          "\nSalario:" << salario <<
          "\nFechaNacimiento:" <<
          fechaNacimiento.anio << "/" <<
          fechaNacimiento.mes << "/" <<
          fechaNacimiento.dia << "\n";
      return info.str();
    }
};
int main()
{
    Empleado luis("Luis",31,18000);
    cout << "---\n" << luis.dameInfoEmpleado();
    return 0;
}
```

Problema:




```
class MiFecha{
public:
    int anio;
    int mes;
    int dia;
};

class Jefe{
    string nombre;
    int edad;
    double salario;
    Fecha fechaNacimiento;
    string departamento;
public:
    Jefe(string nombre, int edad,double salario,
        string departamento){
        this->nombre = nombre;
        this->edad = edad;
        this->salario=salario;
        fechaNacimiento.anio=1979;
        fechaNacimiento.mes=9;
        fechaNacimiento.dia=12;
        this->departamento=departamento;
    }
    string dameInfoJefe(){
        stringstream info;
        info << "Nombre:" << nombre <<
            "\nEdad:" << edad <<
            "\nSalario:" << salario <<
            "\nFechaNacimiento:" <<
            fechaNacimiento.anio << "/" <<
            fechaNacimiento.mes << "/" <<
            fechaNacimiento.dia << "\n";
        return info.str()+ "Departamento:" +
            departamento+"\n";
    }
};

int main()
{
    Jefe luisAlberto("Luis Alberto", 32,
        30000, "Computacion");
    cout << "---\n" << luisAlberto.dameInfoJefe();
    return 0;
}
```

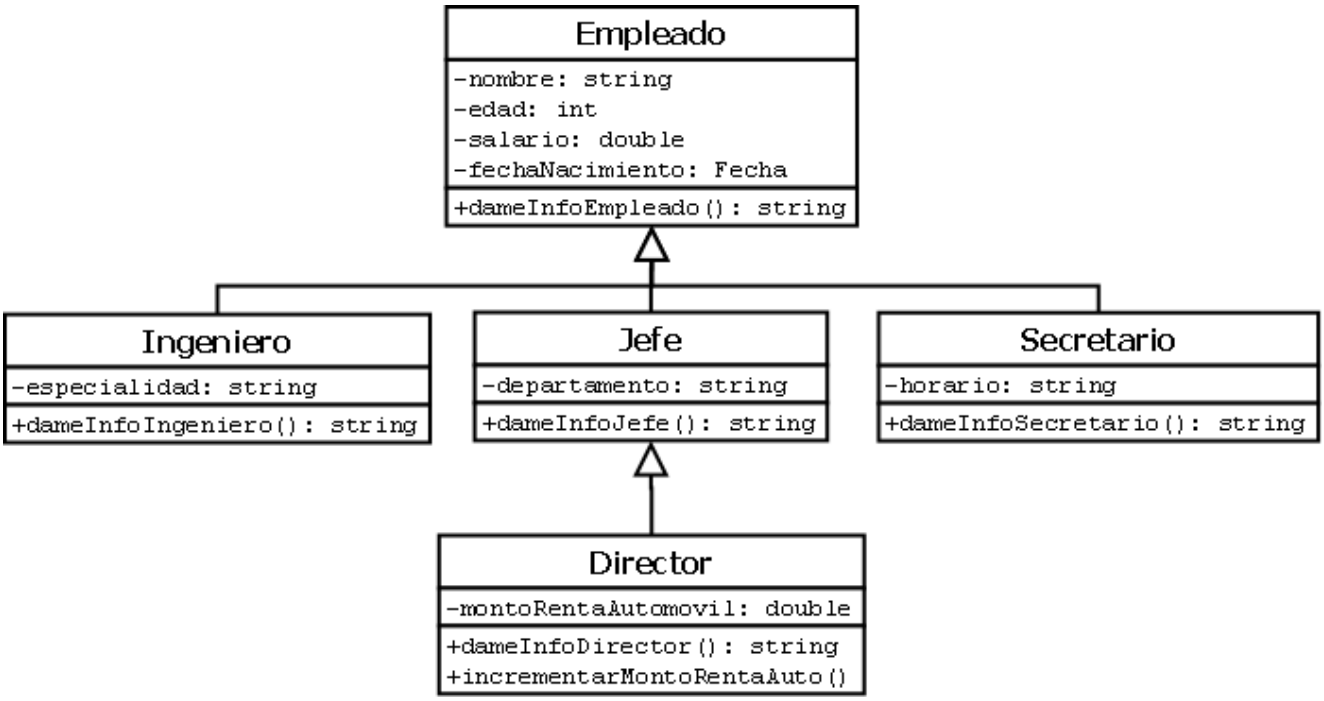
Solución:



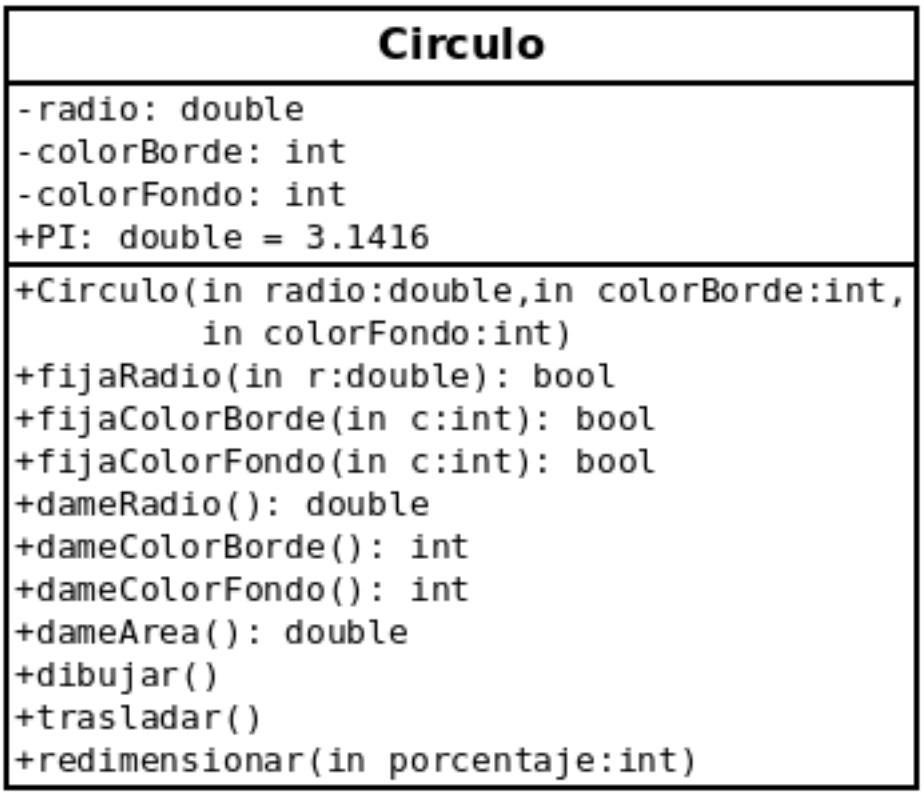

```
class Jefe : public Empleado{
    string departamento;
public:
    Jefe(string nombre, int edad,double salario,
        string departamento):
    Empleado(nombre, edad, salario){
        this->departamento=departamento;
    }
    string dameInfoJefe(){
        return dameInfoEmpleado() "Departamento:"
+
            departamento +"\n";
    }
};

int main()
{
    Empleado luis("Luis",31,18000);
    Jefe luisAlberto("Luis Alberto", 32, 30000,
        "Computacion");
    cout << "---\n" << luis.dameInfoEmpleado();
    cout << "---\n" << luisAlberto.dameInfoJefe();
    return 0;
}
```

La *herencia* es el mecanismo mediante el cual se aplica la reutilización de código en la programación orientada a objetos, esto nos permite abstraer en una clase *padre* todas las variables (*atributos*) y operaciones (*métodos*) que son comunes entre varias clases (*generalización*) y establecer en las clases hijas las demás variables y operaciones que no son comunes (*especialización*).

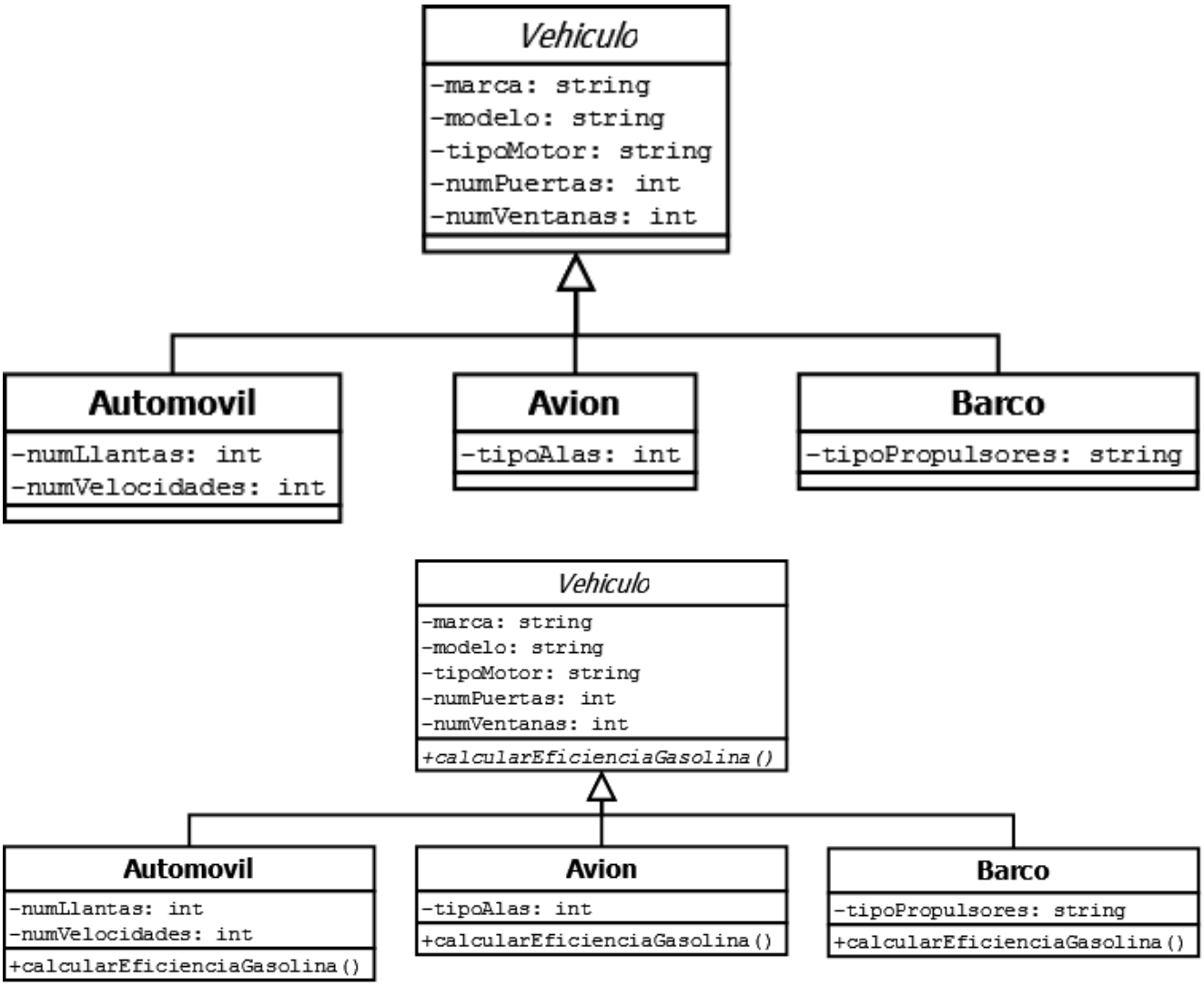
Reutilización de código mediante la herencia

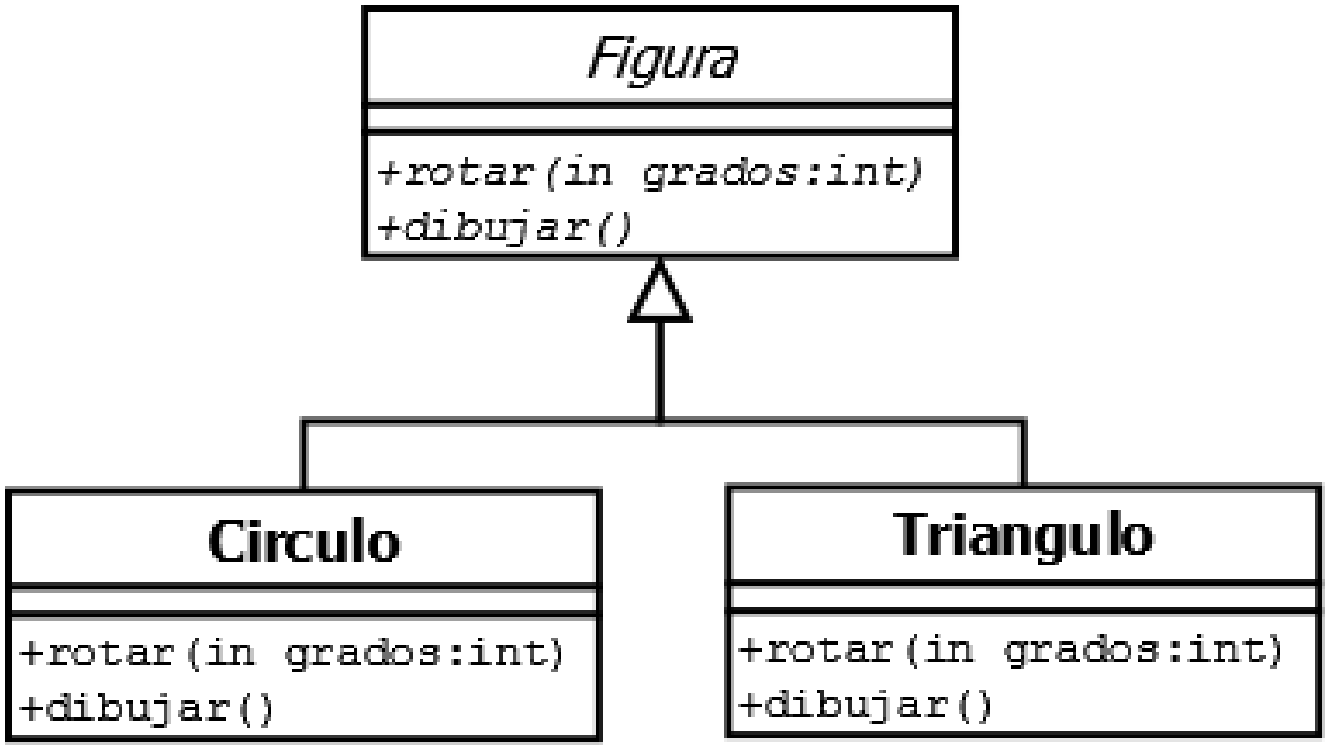


Métodos de Comportamiento



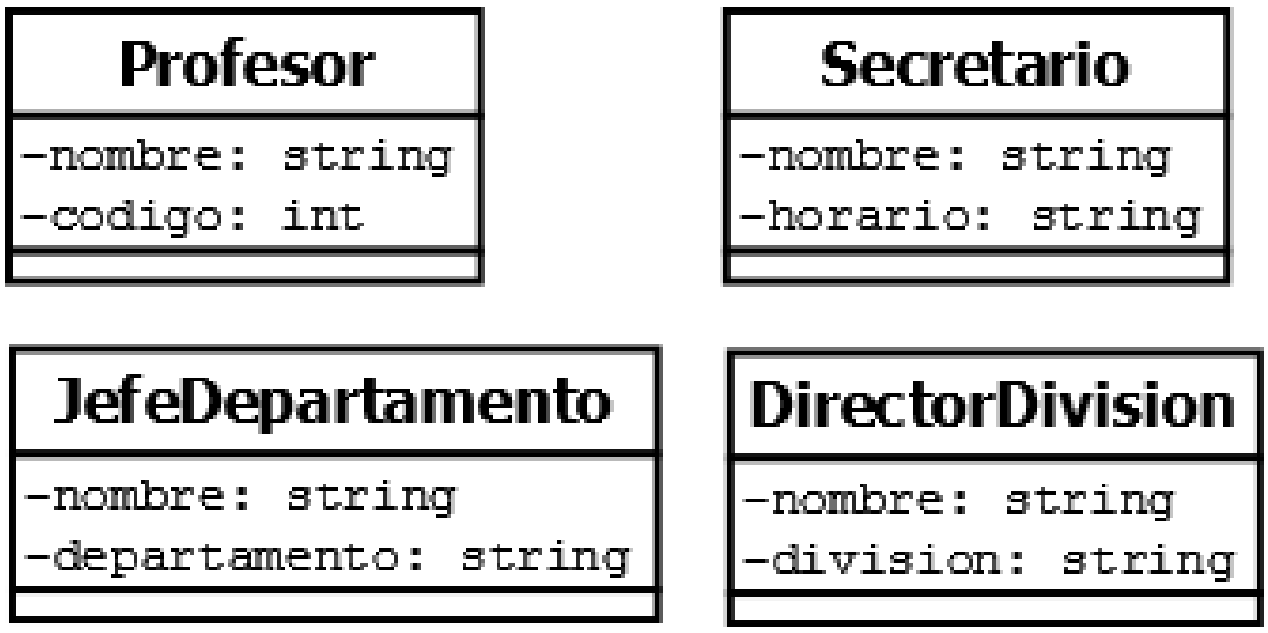
Clases Abstractas



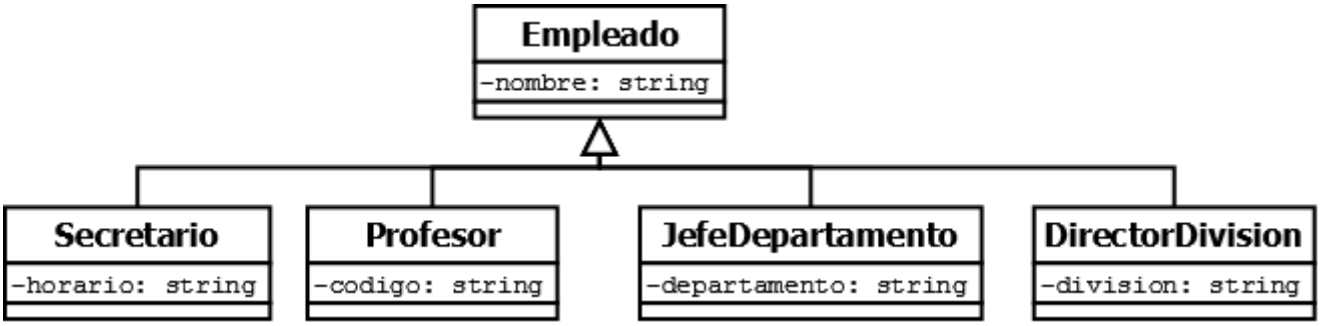


Generalización

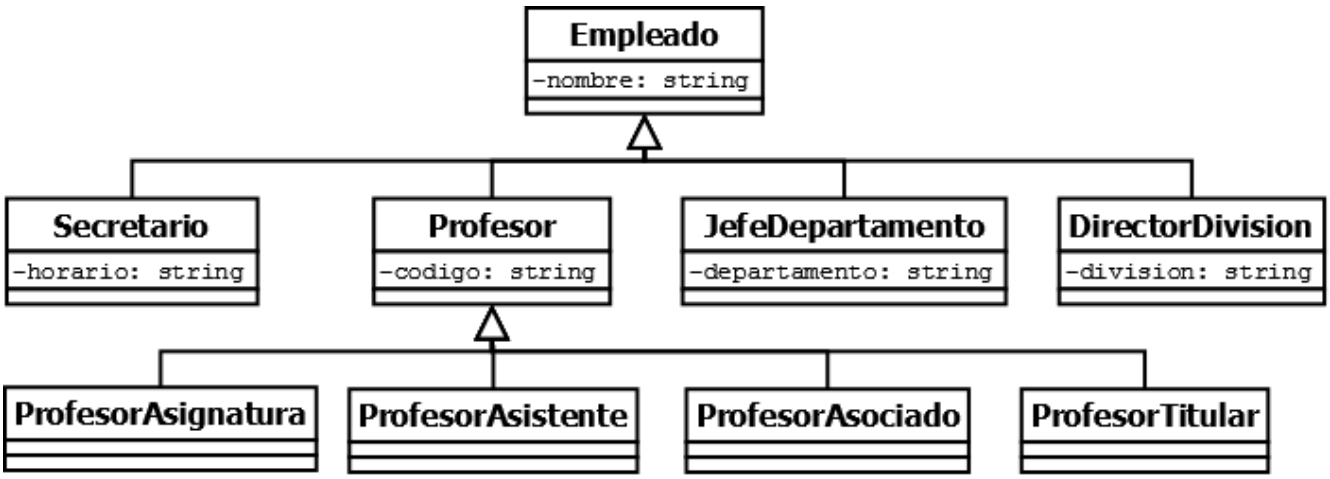
Sin Herencia:



Con Herencia:



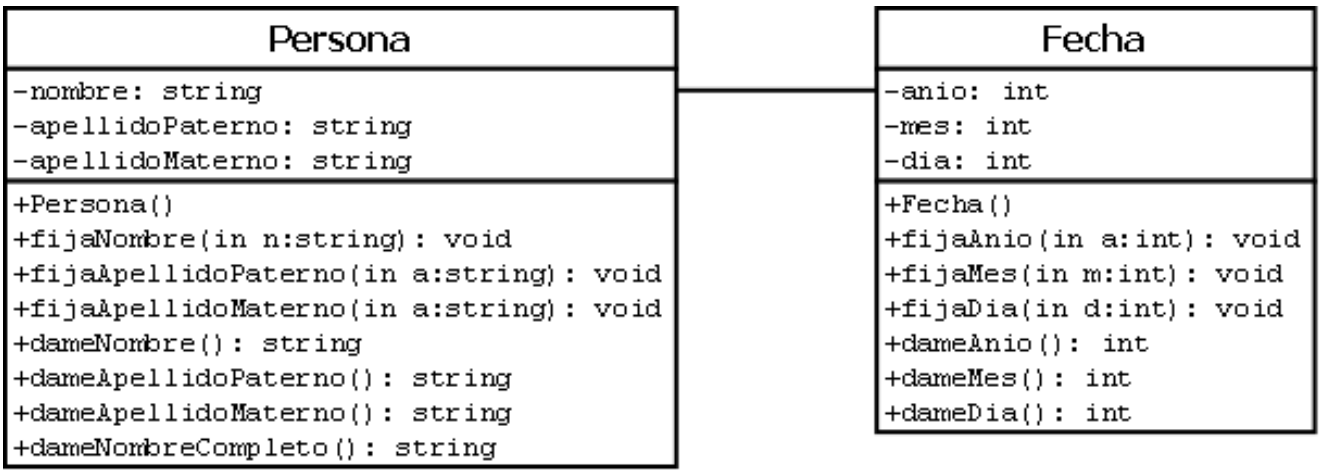
Especialización



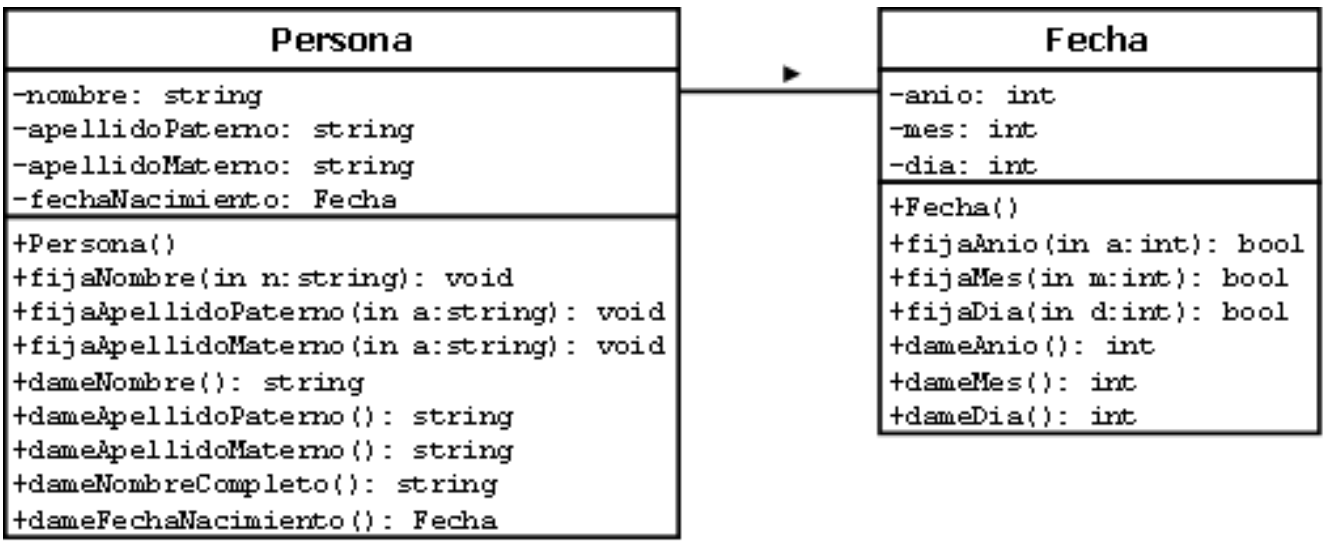
Relaciones entre clases

Asociación de clases

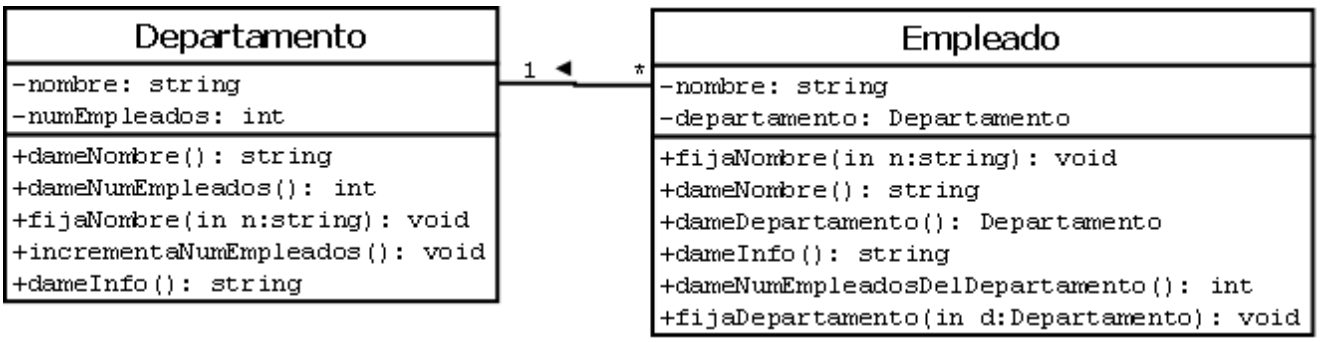
En el siguiente ejemplo dos clases están asociadas pero no es claro cual tiene la **referencia** hacia la otra o si las dos se **apuntan** mutuamente.



En el siguiente ejemplo, es claro quién tiene la **referencia** del otro, y esto se logra ya sea colocando el atributo de tipo Fecha, o bien, usando el triángulo sobre la asociación para indicar quién hace referencia a cuál. La asociación es una relación débil, incluso aplicable si un método cualquiera y no un constructor, recibe un parámetro tipo clase, por lo tanto la asociación no precisamente obliga a que se tenga un atributo del tipo de la otra clase.



En el siguiente ejemplo se ilustra una relación de un Departamento hacia muchos empleados.



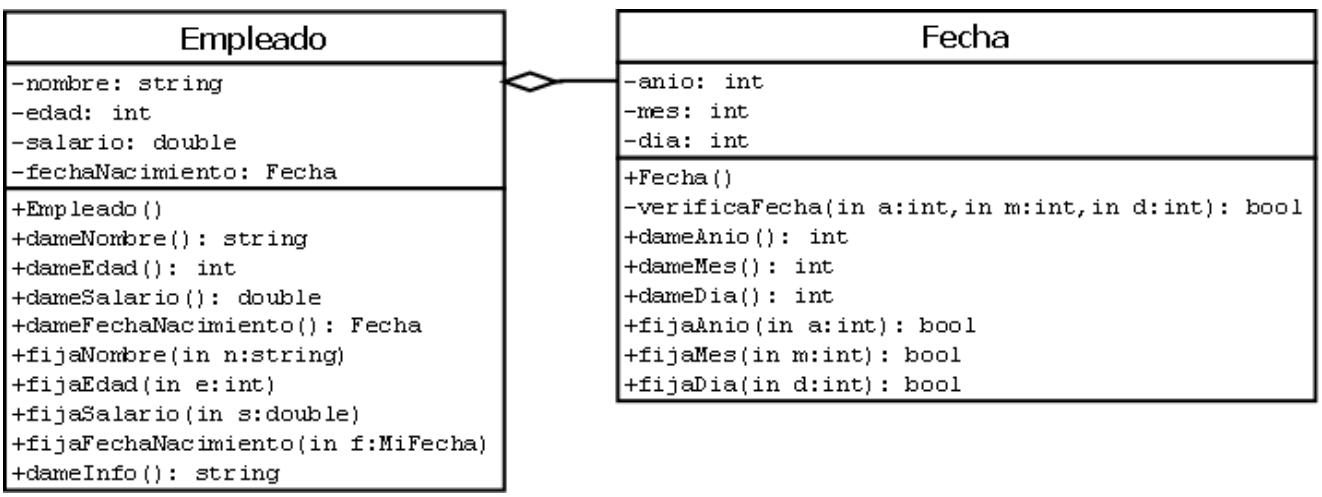
Cardinalidad

Podemos usar cardinalidad para especificar cuántos objetos de otra clase se relacionan con uno; dependiendo del mundo en abstracción se decide la cuantificación.

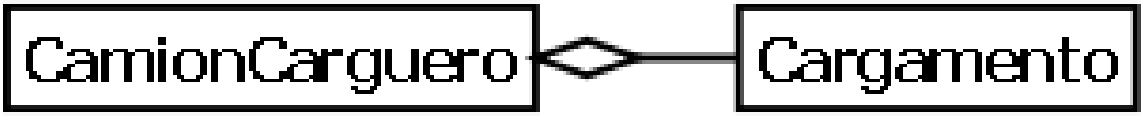
Para cardinalidad se pueden usar los siguientes en cualquiera de los extremos de la relación: 1, 2, 3, etc., M, N, donde estos son constantes, y si la cantidad es infinita se usa un *. La cardinalidad aplica para las relaciones de asociación, agregación y composición.

Agregación de clases

La agregación indica que una clase (la del lado del rombo), necesita de otra como parte de su descripción, o sea un atributo de la clase, pero la inicialización de dicho atributo no es indispensable para que el objeto (p. ej. Empleado) exista, permitiendo que mediante un método tipo fija() se asigne o agregue el valor del atributo, tiempo después de haber instanciado el objeto (Empleado).



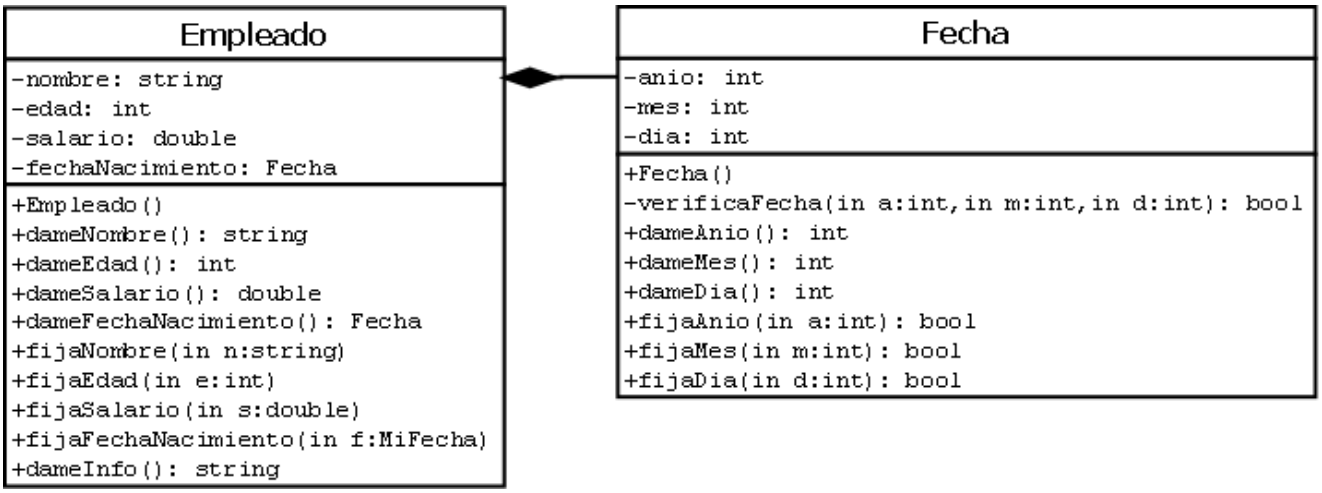
Otros ejemplos:



…el CamionCarguero puede existir y rodar sin cargamento.

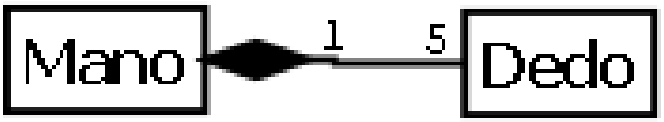
Composición

La composición es una relación de tipo contenedor-a-componente, obliga en diseño a que una instancia de una clase requiere de la inicialización de un atributo, ya sea que reciba la referencia al componente como parámetro en su constructor o bien que inicialice su componente con valores por defecto durante la ejecución del constructor del contenedor.



Otro ejemplo:

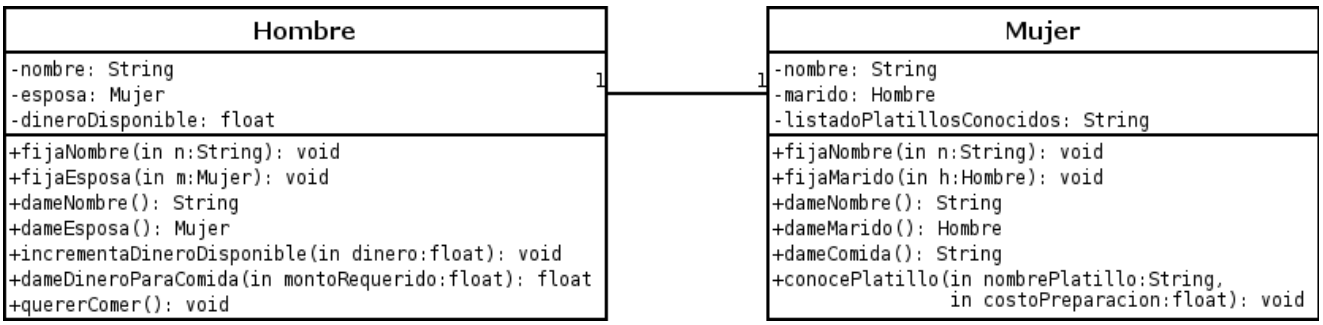
…el CamionDePasajeros puede existir sin pasajeros, pero no tiene mucho sentido su existencia si no transporta a alguien, al menos el Chofer.



…una mano se compone de 5 dedos, sería inapropiado instanciar una mano sin sus dedos.

Asociación bidireccional

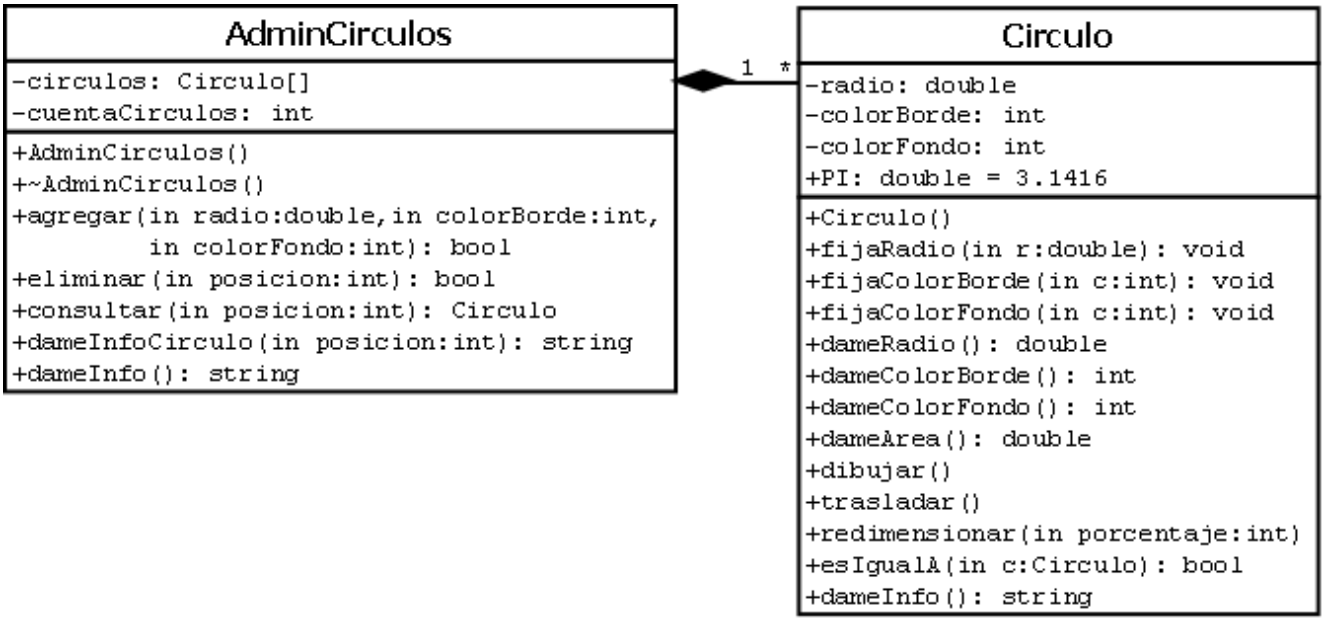
En una asociación podemos tener a dos objetos que pueden existir por separado, pero que en algún momento necesitan comunicación el uno con el otro para la ejecución satisfactoria de sus métodos; aquí no hay una relación de contenedor-componente por lo que no se coloca el rombo de agregación si el de composición; nótese que podemos usar cardinalidad para especificar cuántos objetos de otra clase se relacionan con uno; dependiendo del mundo en abstracción se decide la cuantificación.



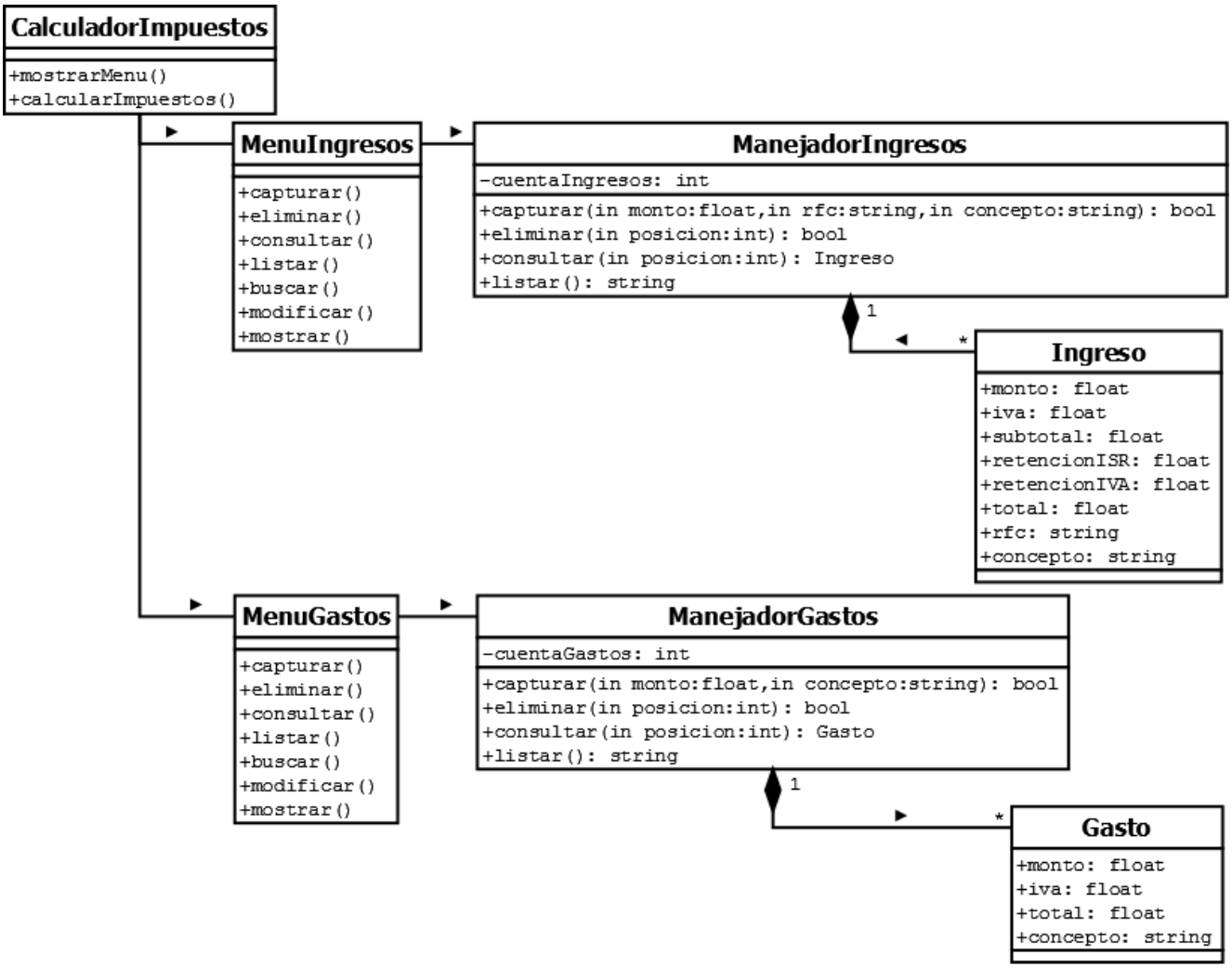
Delegación

Es cuando un objeto le encarga a uno o más objetos la realización de un trabajo que le han encomendado; p. ej. “Jefe” provee un método “elaboraReporte(..)” y delega la elaboración de dicho reporte a otro objeto digamos “Secretario”, el cual a su vez provee un método “elaboraReporte(…) que devuelve el reporte.

Clase Administradora



Diseño de Aplicaciones Orientado Objetos



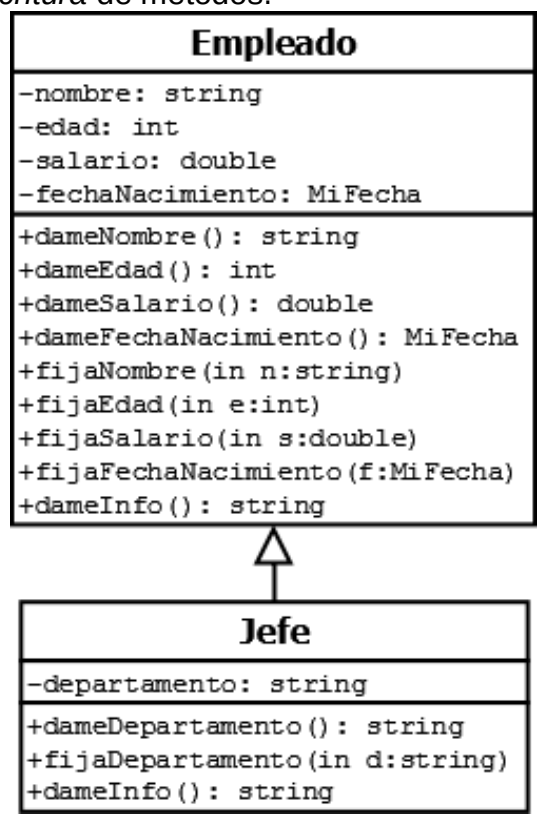
- NOTA IMPORTANTE. Los objetos Ingreso y Gasto debieran tener sus atributos privados y sus métodos dameAtributo() (getters) y fijaAtributo() (setters que validen la información modelada)

Polimorfismo

- *Polimorfismo* es la habilidad de tener muchas formas diferentes; por ejemplo la clase Jefe tiene acceso a los métodos de la clase Empleado
- Un objeto tiene únicamente una forma, si se instanció como Empleado es un Empleado, si se instanció como Jefe, es un Jefe
- El *polimorfismo* permite que un objeto se comporte acorde a la naturaleza de la información mediante la *sobrecarga* de métodos.

- Una variable de instancia puede referenciar a objetos de diferentes formas
- Si una variable de instancia es de tipo Empleado, solo le son válidos de invocar los atributos y métodos (públicos) de la clase Empleado, aunque el objeto instanciado fuese por ejemplo de tipo Jefe

- El *polimorfismo* permite que objetos de distintas clases se comporten acorde a la naturaleza propia (como clase) cuando contiene atributos y métodos heredados mediante la *sobreescritura* de métodos.



Herencia Múltiple - Interfaces