V 1.3

Tema 1
Introducción a la POO

Programación Orientada a Objetos

José Francisco Vélez Serrano

Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Índice

- La complejidad del software y su tratamiento
- Características de la programación orientada a objetos
- Abstracción
- Encapsulación
- Jerarquías
- Modularización
- Otras características de la POO
  - Tipado
  - Concurrencia
  - Persistencia

La complejidad del software y su tratamiento

El software es complejo y los lenguajes de programación evolucionan para ayudar a tratar esa complejidad.

La complejidad del software

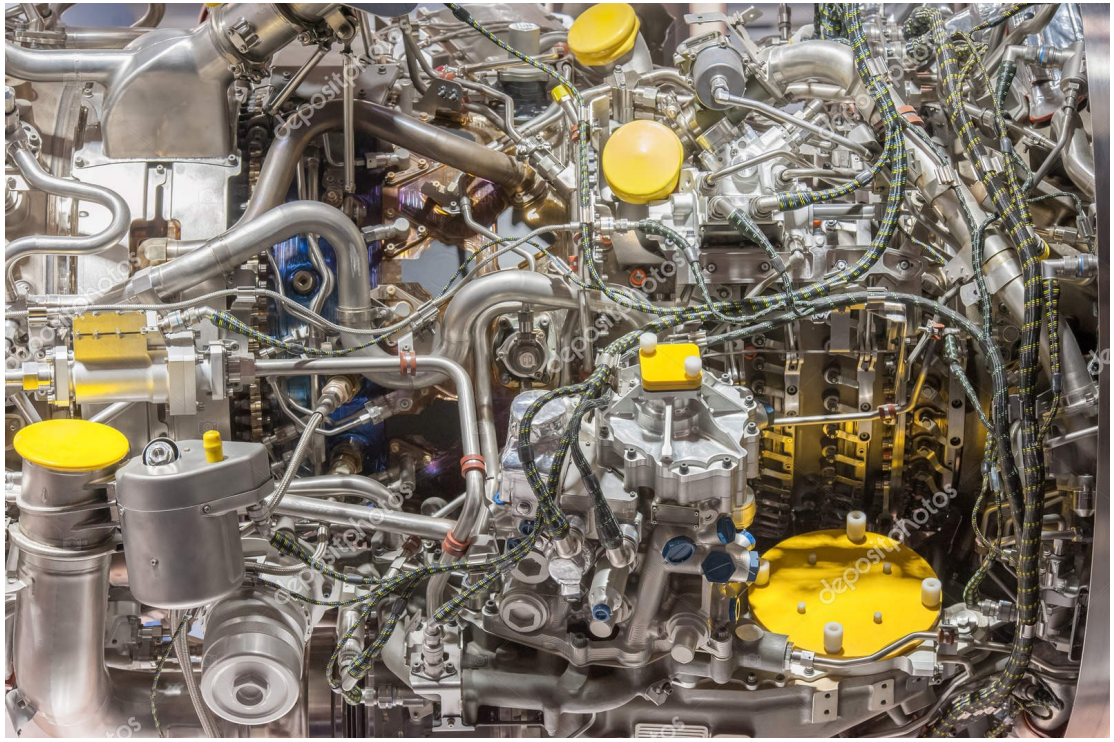
La complejidad es una propiedad inherente al software. No es un accidente debido a una mala gestión o un mal diseño.

La complejidad del software se deriva fundamentalmente de:

- La complejidad del dominio del problema
- La dificultad de gestionar el proceso de desarrollo
- La flexibilidad de las herramientas de software
- Comportamiento impredecible del software

La complejidad del dominio del problema

Si un problema es complejo, el software que resuleve el problema suele ser complejo también.



by nelsonart

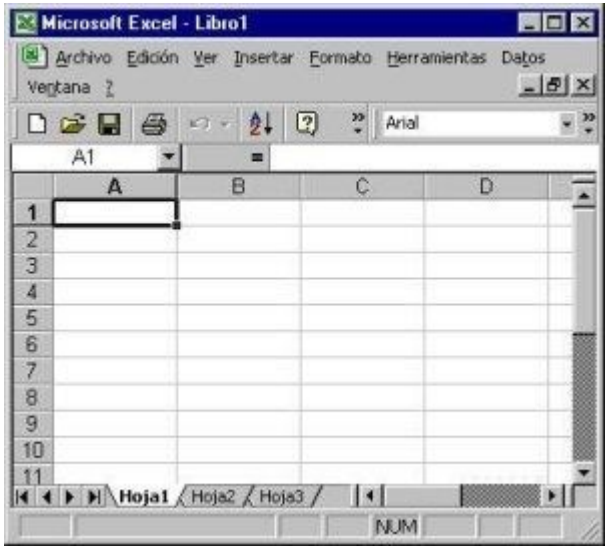
La gestión el proceso de desarrollo

La gestión del proceso de desarrollo del software añade complejidad.

- La comunicación entre las personas.
- La gestión de las personas y sus habilidades.

La flexibilidad de las herramientas de software

Las herramientas que proporcionan mucha flexibilidad se pueden utilizar de muchas formas equivocadas.



Comportamiento impredecible del software

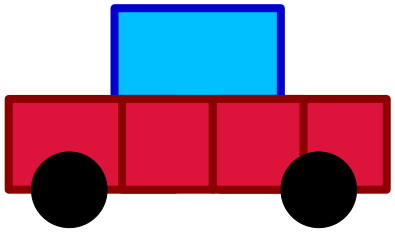
El software pude cambiar de un estado a otro completamente distinto con solo cambiar un bit.

```
Delete all = 0
```

Herramientas para tratar la complejidad

Los humanos, para enfrentarnos a la complejidad usamos principalmente 3 herramientas:

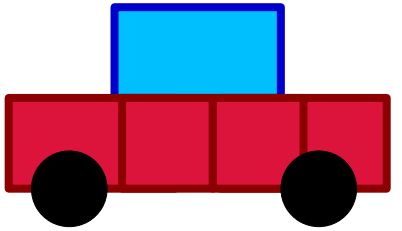
- Abstraer: Se ignoran los detalles del problema que parecen poco significativos y se concentra el esfuerzo en los que se consideran esenciales.



Herramientas para tratar la complejidad

Los humanos, para enfrentarnos a la complejidad usamos principalmente 3 herramientas:

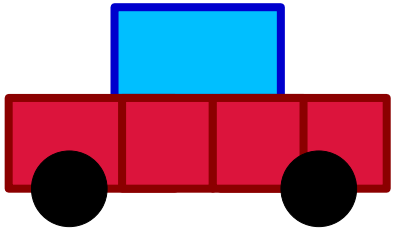
- Abstraer: Se ignoran los detalles del problema que parecen poco significativos y se concentra el esfuerzo en los que se consideran esenciales.
- Descomponer: El problema completo se descompone en problemas más simples que se resuelven por separado.

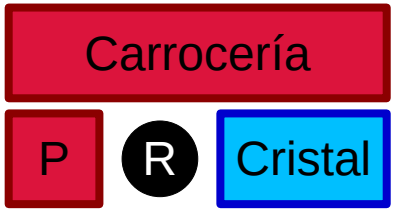


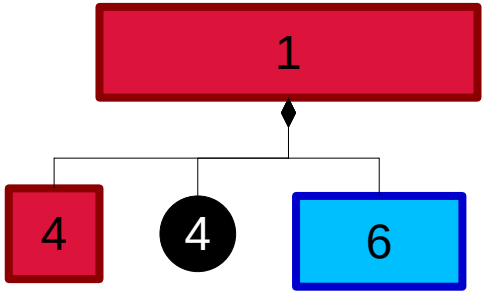
Herramientas para tratar la complejidad

Los humanos, para enfrentarnos a la complejidad usamos principalmente 3 herramientas:

- Abstraer: Se ignoran los detalles del problema que parecen poco significativos y se concentra el esfuerzo en los que se consideran esenciales.
- Descomponer: El problema completo se descompone en problemas más simples que se resuelven por separado.
- Jerarquizar: Se ordenan los elementos del problema, localizando semejanzas y diferencias.







Tipos de descomposiciones

Clásicamente se distinguen dos formas de descomponer un problema:

- Descomposición algorítmica: El problema se descompone en tareas más simples.
- Descomposición orientada a objetos: El problema se descompone en objetos de cuya interacción surge la solución.

Estas dos formas de descomponer un problema no son incompatibles entre sí.

Paradigmas de programación

Orientación a aspectos

Procedimental < Orientado a objetos

Programación concurrente

Programación funcional

Imperativo <> Declarativo

Programación lógica

Metaprogramación

Programación simbólica

Secuencial <> Dirigido por eventos

Programación diferenciable

Genericidad

La complejidad del software y su tratamiento

Evolución de lenguajes de programación

1951-1960	1961-1970	1971-1980	1981-1990	1991-2000	2001-2010	2011-2020
Ensamblador	APL	C	Ada	Python	C#	Dart
Fortran	Simula	Smalltalk	Matlab	Visual Basic	Scratch	Kotlin
Cobol	Basic	Prolog	Foxpro	Lua	Grovy	Swift
Lisp	Pascal	ML	Eiffel	R	Scala	
	B	SQL	Objetive C	Ruby	Clojure	
		C++	Erlang	Java	Go	
			Hashkell	Javascript	Rust	
			Perl	Delphi		
				Php		

Características de la POO

Llamamos lenguajes de Programación Orientados a Objetos a los que proporcionan ciertas características.

Definición de objeto

Un objeto es un componente software que:

- Puede recibir mensajes y responder a los mismos
- Tiene identidad
- Tiene un estado
- Tiene un comportamiento bien definido

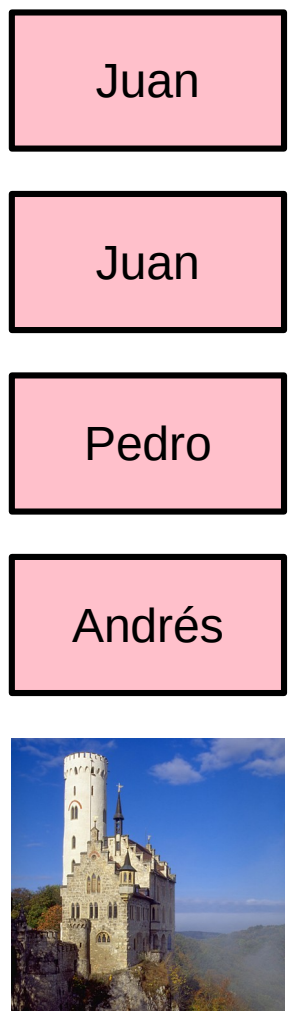
Idealmente, puedes tener varios objetos, cada uno con su identidad y con su estado interno. Le puedes enviar un mensaje a cualquiera de los objetos y esperar una respuesta.

Ejemplo de objetos

Un programa tiene 4 objetos de tipo cadena de texto modificable y un objeto de tipo imagen.

Dos de los objetos de tipo cadena contienen, como estado interno, la palabra "Juan" y los otros dos tienen la palabra "Pedro" y "Andrés".

El objeto de tipo imagen contiene un mapa de 512x512 píxeles en color.

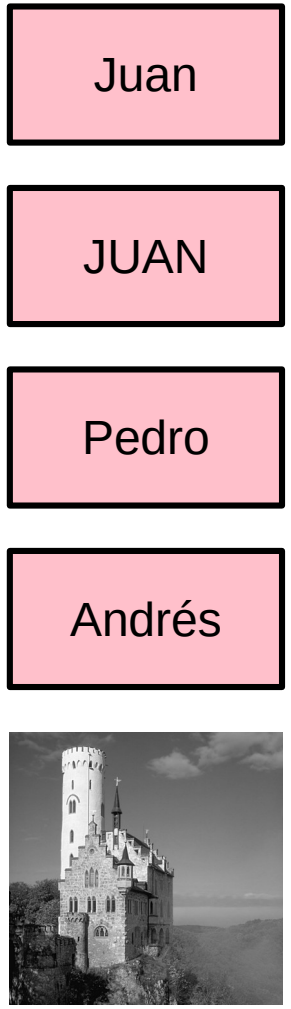


Ejemplo de mensajes a objetos

Los dos objetos con la palabra "Juan" tienen el mismo estado interno, pero son objetos con diferente identidad.

Si a uno de los objetos con estado "Juan" le pido que se pase a mayúsculas, su estado interno cambia a "JUAN", pero el del otro no cambia.

Si al objeto de tipo imagen le pido que cambie a niveles de gris el mapa de gris cambia eliminado el color.



Lenguaje Unificado de Modelado (UML)

El Lenguaje Unificado de Modelado (UML) se usará para representar gráficamente los ejemplos que se vayan proponiendo.

UML es un lenguaje gráfico para visualizar, especificar, construir y documentar los componentes de un sistema software orientado a objetos.

La introducción de la notación UML se hará de manera gradual según se vaya necesitando.


Booch


Runbaugh


Jacobson

Características de la POO

Lenguajes orientados a objetos

La POO se basa en el uso de las siguientes capacidades primarias:

- Abstraer
- Encapsular
- Modularizar
- Jerarquizar

También se pueden considerar estas capacidades secundarias:

- Tipo
- Concurrencia
- Persisistencia

Los lenguajes de programación orientados a objetos se caracterizan porque proporcionan mecanismos que dan soporte a estas capacidades.

Abstracción

La posibilidad de un lenguaje para definir abstracciones.

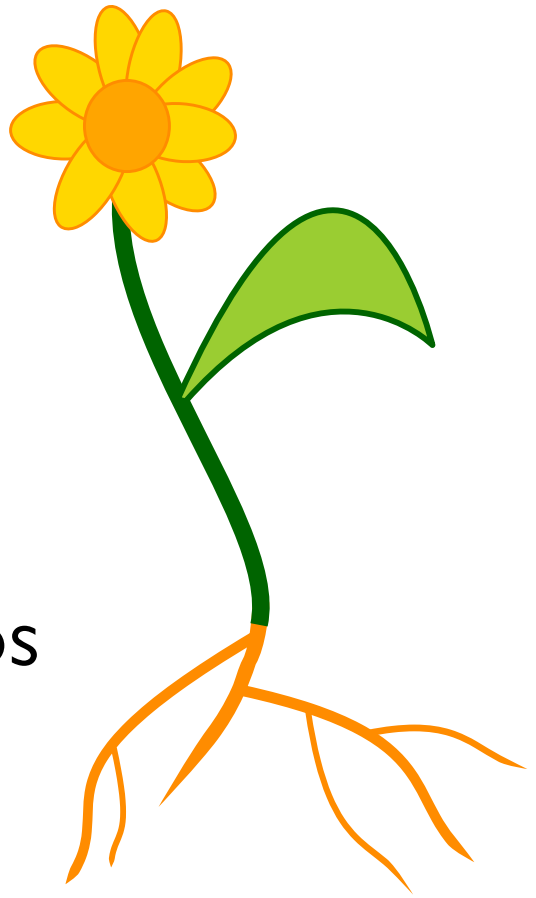
Abstracción

Definición

Los humanos hemos desarrollado la capacidad de abstraer para tratar la complejidad.

Abstraer es la capacidad que permite distinguir aquellas características fundamentales de un objeto que lo hacen diferente del resto, y que proporcionan límites conceptuales bien definidos relativos a la perspectiva del que lo visualiza.

Al estudiar algo ignoramos los detalles, y tratamos con ideas generales de un modelo simplificado de ese algo.

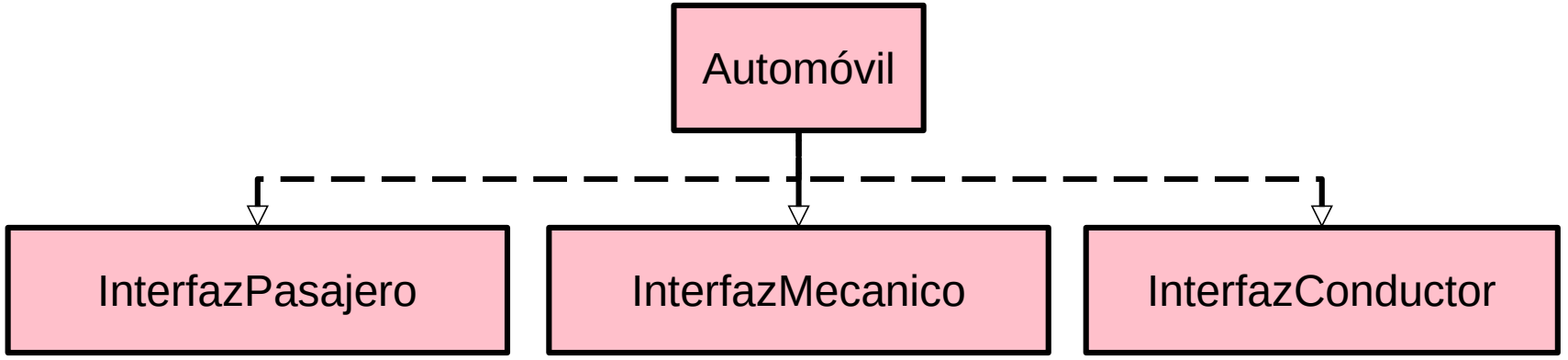


Soporte a la abstracción en los lenguajes

Los lenguajes de POO facilitan abstraer gracias a que permiten definir interfaces para comunicarse con clases de objetos.

Estas interfaces están compuestas por métodos, que son funciones que pueden aplicarse sobre el objeto y que pueden verse como los mensajes que es posible enviar al objeto.

Normalmente un objeto puede tener varias interfaces.



Abstracción

Clases o modelos como abstracciones

Algunos lenguajes de POO (como C++ o Java) disponen el concepto de clase de objetos, o simplemente clase, que une las interfaces definidas en el proceso de abstracción con la implementación del comportamiento deseado.

Otros lenguajes (como SmallTalk o JavaScript) no permiten definir clases de objetos, siendo preciso definir el comportamiento sobre un objeto particular y luego usarlo como prototipo o modelo.

La implementación se basa en la definición de propiedades (variables del objeto) y en el código interno de los métodos.

Las abstracciones como cliente/servidor

Entre los objetos de un programa siempre se crean comportamientos de tipo cliente/servidor.

El objeto cliente conoce el comportamiento del objeto servidor y le pide cosas. Entre ellos hay un contrato que establece las responsabilidades del servidor.

El orden en que se aplican las operaciones que el cliente puede realizar se conoce como protocolo.

Este protocolo suele implicar precondiciones y poscondiciones que deben ser satisfechas por los métodos de la abstracción.

Cuando un cliente envia un mensaje a un objeto servidor y este no puede cumplir las poscondiciones se produce una excepción.

Ejemplo de protocolo

El contrato de los objetos pila de enteros establece que puede almacenar y devolver enteros.

El protocolo de los objetos pila exige que se inserte algo antes de poder recuperarlo.

La siguiente secuencia sería un caso real:

1. El objeto calculadora crea un objeto de tipo pila.
2. El objeto calculadora inserta un entero con valor 7 en el objeto de tipo pila.
3. El objeto calculadora pide un entero al objeto y el objeto pila devuelve un 7.
4. El objeto calculadora pide otro entero al objeto pila cuando ya quedan más elementos en la pila.
5. El objeto pila lanza una excepción al no poder devolver nada.

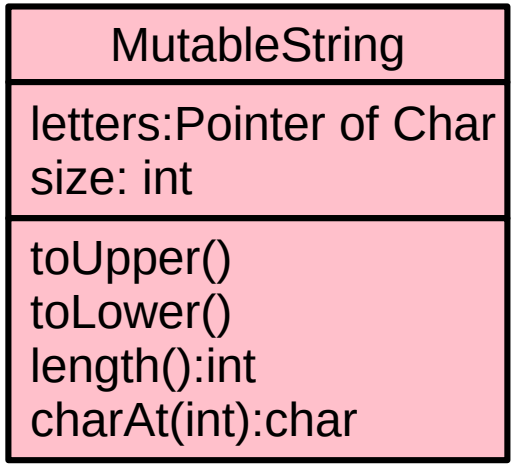
Medir la calidad de una abstracción

Para ayudar a crear abstracciones se puede medir su calidad revisando los siguientes aspectos:

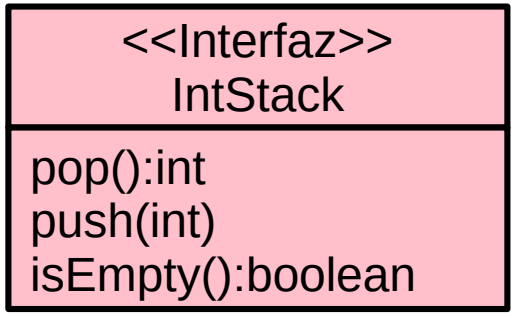
- Acoplamiento.- Minimizar el grado de asociación entre diferentes abstracciones.
- Cohesión.- Maximizar el grado de asociación dentro de una abstracción.
- Suficiencia y completitud.- Que tenga las características precisas para permitir un funcionamiento eficiente y completo.
- Primitividad.- Las operaciones de una abstracción deben ser lo más básicas posibles.

La abstracción en UML

En UML las clases se representan en los diagramas estáticos de clases mediante rectángulos. En el interior de cada rectángulo se indican, separados por líneas horizontales, el nombre de la clase, las propiedades y los métodos.



Las interfaces se representan mediante los mismos rectángulos anteponiendo la palabra interfaz. Además, las interfaces no tienen propiedades.



Diferencia entre un LOO y funcional

En un LOO la abstracción incluye los métodos primitivos

```
p = Pila()
p.insertar(25)
```

En un lenguaje funcional no:

```
p = Pila()
insertar(p,25)
```

Encapsulación

La posibilidad de un lenguaje para ocultar partes de su implementación.

Encapsulación

Definición

Encapsular es la capacidad que permite mantener oculta la implementación de una abstracción para los usuarios de la misma.

El objetivo de encapsular es el de evitar que un sistema dependa de cómo se ha implementado otro.

Facilita que los clientes no noten los cambios internos en la implementación de otra abstracción.

Encapsulación

La encapsulación en los lenguajes

Algunos lenguajes (Java, C++...) permiten el etiquetado de partes públicas o privadas en el código.

Otros lenguajes ocultan el código en bibliotecas y solo permiten acceder mediante ciertas interfaces.

La encapsulación en UML

UML añade los siguientes símbolos:

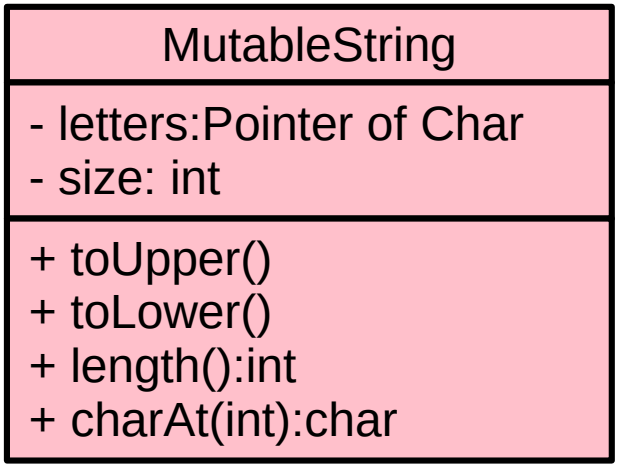
+ Para las partes públicas

- Para las partes privadas

Para las partes protegidas

Habitualmente las propiedades son privadas y los métodos públicos (salvo los auxiliares).

En general, en UML la ausencia de un adorno no implica que lo que representa no esté luego en la implementación.



Jerarquía

La posibilidad de un lenguaje para definir jerarquías de abstracciones.

Definición

Jerarquizar es una capacidad que permite ordenar abstracciones.
La organización de las abstracciones en jerarquías permite detectar estructuras y comportamientos comunes, simplificando el desarrollo.

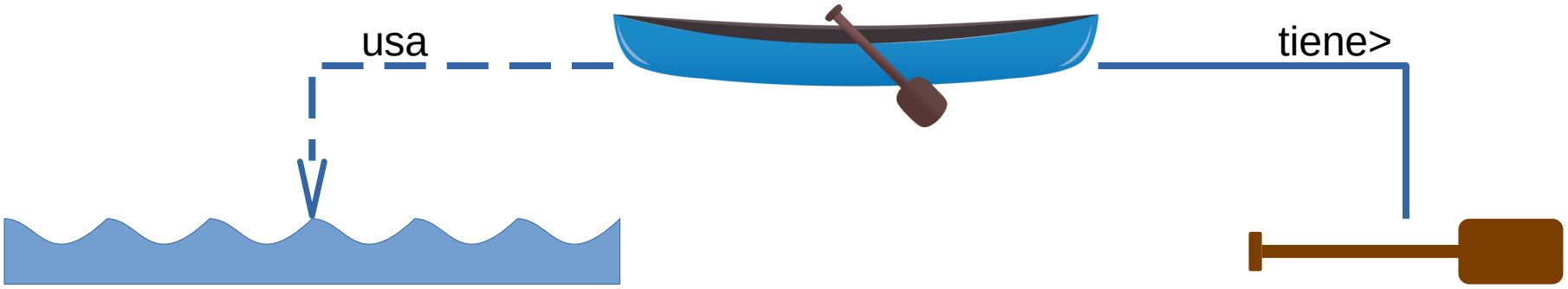
En el esquema de programación orientada a objetos se definen dos formas básicas de jerarquías:

- Jerarquías entre objetos.
- Jerarquías entre clases e interfaces.

Jerarquías de objetos

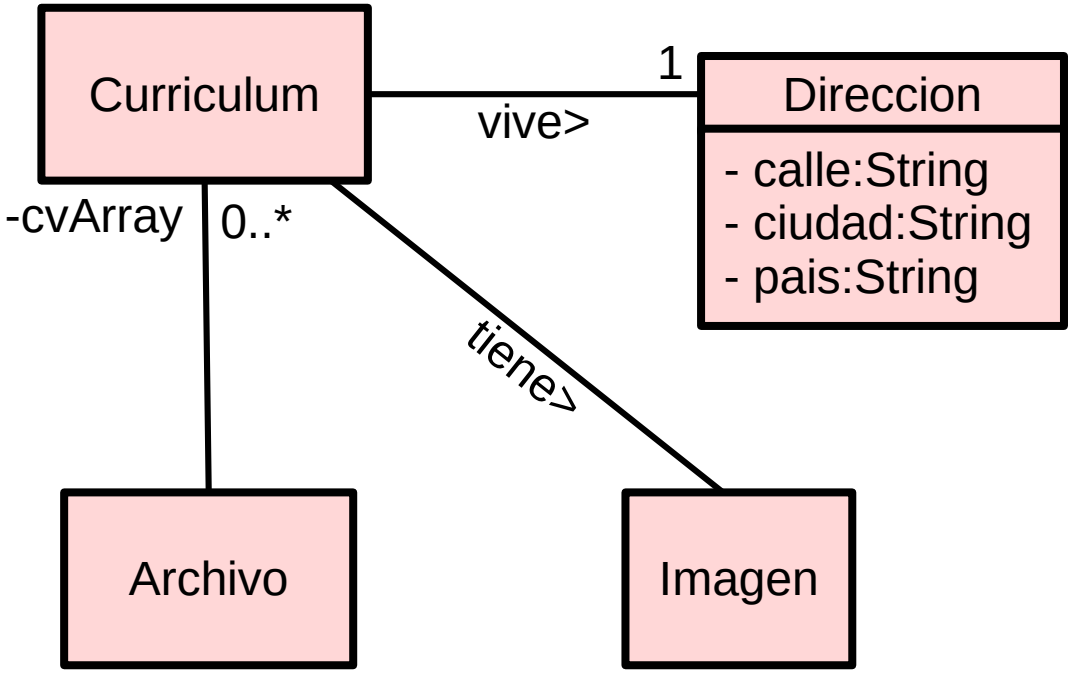
Las jerarquías entre objetos se pueden clasificar en 2 tipos de relaciones:

- Relaciones de asociación: establecen relaciones del tipo "tal objeto contiene a tal otro objeto".
- Relaciones de dependencia o uso: dan lugar a relaciones del tipo "tal objeto usa tal otro objeto"



Jerarquías de objetos: Asociación en UML

La asociación se representa mediante una línea o mediante una propiedad y se puede adornar con elementos diversos: nombre, multiplicidad, marcas de agregación, composición y navegación, nombre de propiedades, visibilidad…

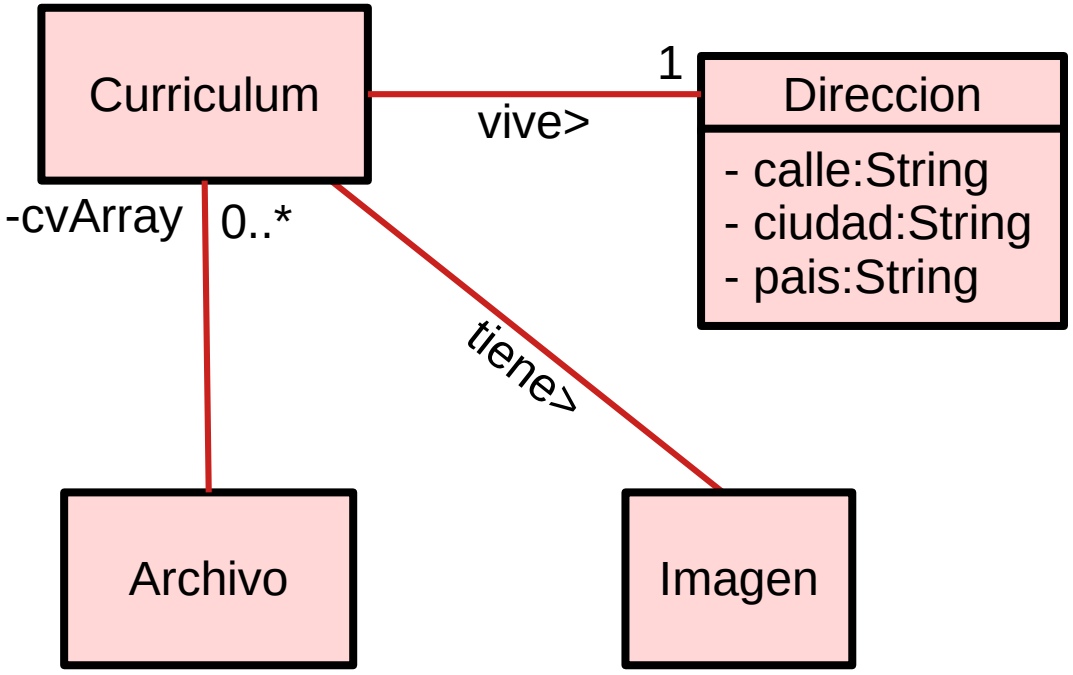


Este diagrama dice que:

- La clase Curriculum está asociada con las clases Direccion, Imagen y Archivo.
- La clase Curriculum contiene 1 objeto de la clase Direccion, que es donde vive.
- La clase Archivo guarda de 0 a n objetos de la clase Curriculum.
- La clase Direccion está asociada con String pues contiene 3 objetos de la clase String.
- Los objetos Curriculum se guardan en la variable privada cvArray de Archivo.

Jerarquías de objetos: Asociación en UML

La asociación se representa mediante una línea o mediante una propiedad y se puede adornar con elementos diversos: nombre, multiplicidad, marcas de agregación, composición y navegación, nombre de propiedades, visibilidad…

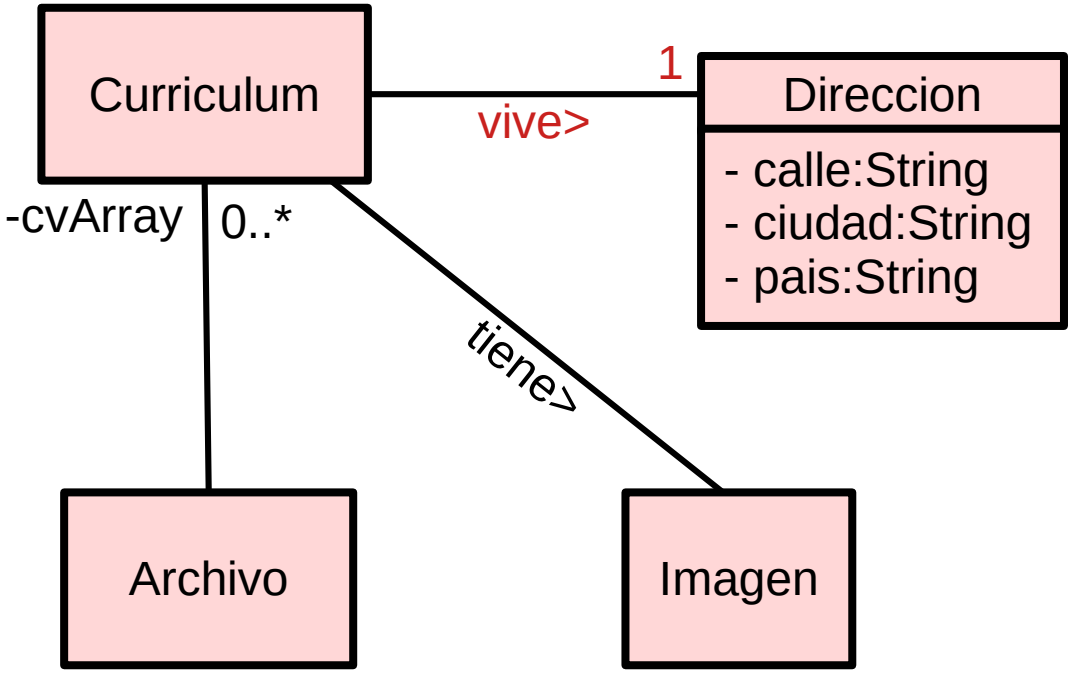


Este diagrama dice que:

- La clase Curriculum está asociada con las clases Direccion, Imagen y Archivo.
- La clase Curriculum contiene 1 objeto de la clase Direccion, que es donde vive.
- La clase Archivo guarda de 0 a n objetos de la clase Curriculum.
- La clase Direccion está asociada con String pues contiene 3 objetos de la clase String.
- Los objetos Curriculum se guardan en la variable privada cvArray de Archivo.

Jerarquías de objetos: Asociación en UML

La asociación se representa mediante una línea o mediante una propiedad y se puede adornar con elementos diversos: nombre, multiplicidad, marcas de agregación, composición y navegación, nombre de propiedades, visibilidad…

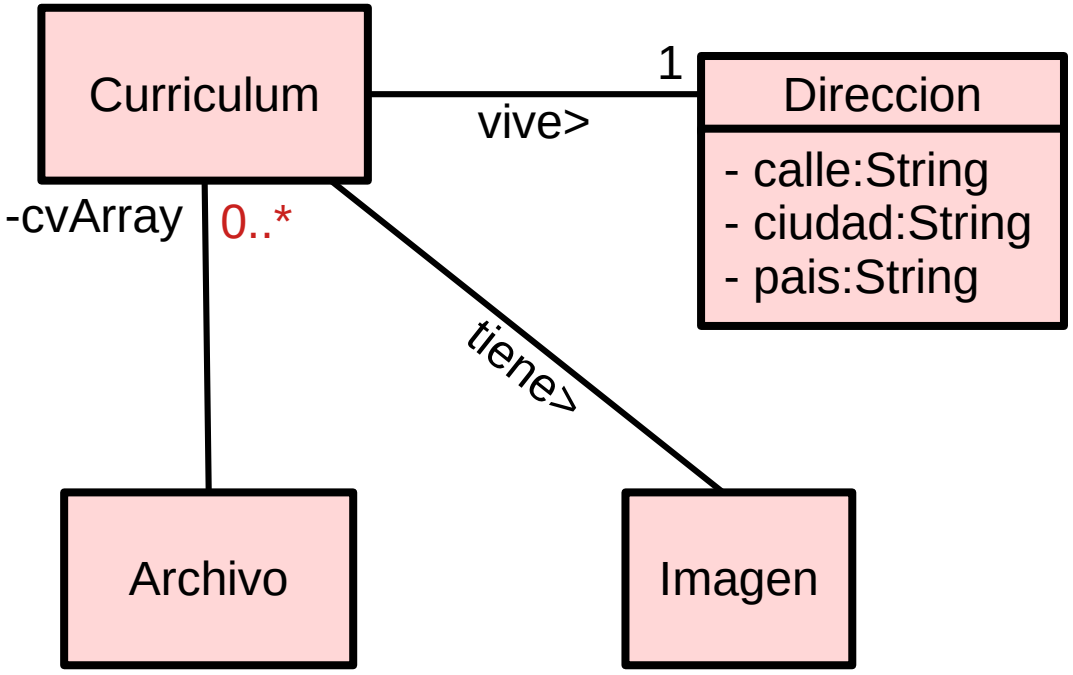


Este diagrama dice que:

- La clase Curriculum está asociada con las clases Direccion, Imagen y Archivo.
- La clase Curriculum contiene 1 objeto de la clase Direccion, que es donde vive.
- La clase Archivo guarda de 0 a n objetos de la clase Curriculum.
- La clase Direccion está asociada con String pues contiene 3 objetos de la clase String.
- Los objetos Curriculum se guardan en la variable privada cvArray de Archivo.

Jerarquías de objetos: Asociación en UML

La asociación se representa mediante una línea o mediante una propiedad y se puede adornar con elementos diversos: nombre, multiplicidad, marcas de agregación, composición y navegación, nombre de propiedades, visibilidad…

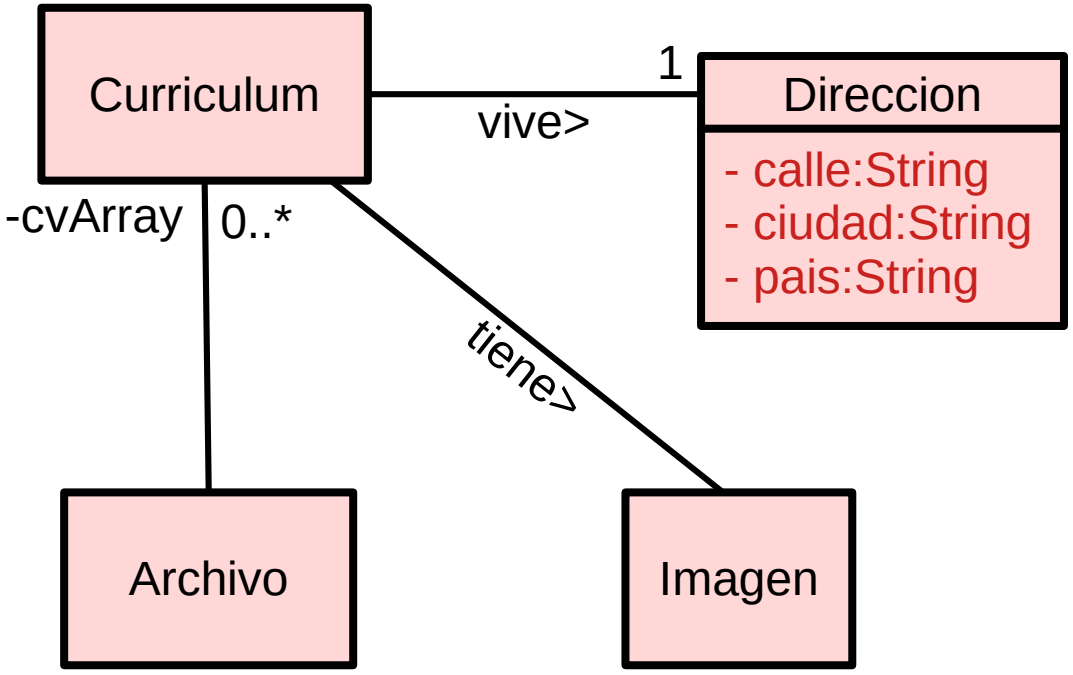


Este diagrama dice que:

- La clase Curriculum está asociada con las clases Direccion, Imagen y Archivo.
- La clase Curriculum contiene 1 objeto de la clase Direccion, que es donde vive.
- La clase Archivo guarda de 0 a n objetos de la clase Curriculum.
- La clase Direccion está asociada con String pues contiene 3 objetos de la clase String.
- Los objetos Curriculum se guardan en la variable privada cvArray de Archivo.

Jerarquías de objetos: Asociación en UML

La asociación se representa mediante una línea o mediante una propiedad y se puede adornar con elementos diversos: nombre, multiplicidad, marcas de agregación, composición y navegación, nombre de propiedades, visibilidad…

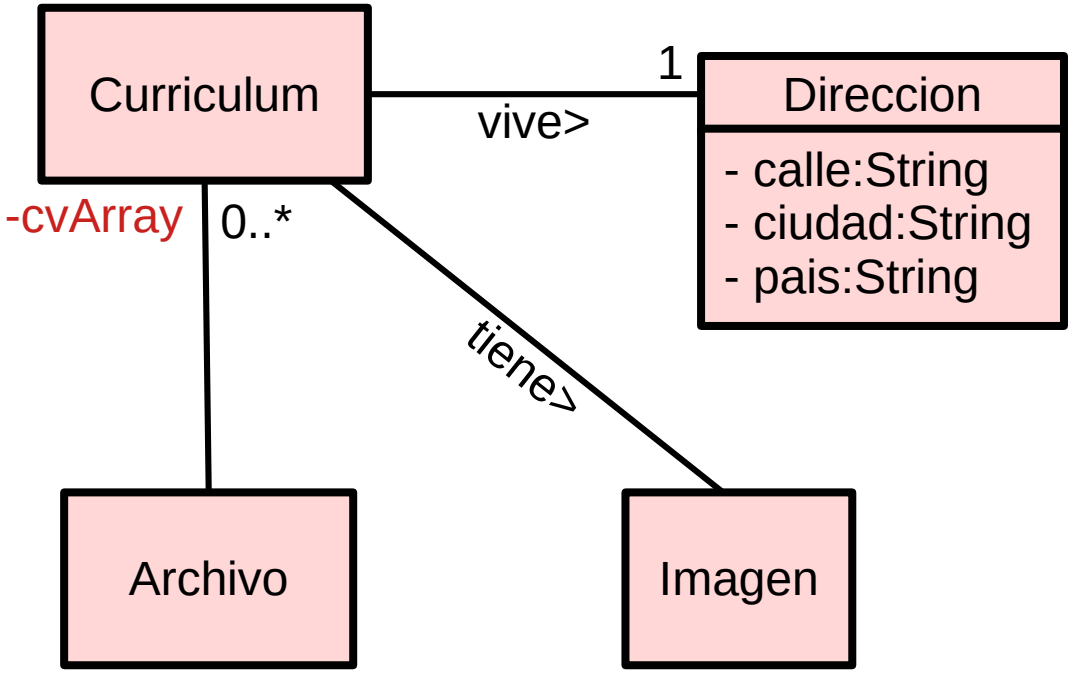


Este diagrama dice que:

- La clase Curriculum está asociada con las clases Direccion, Imagen y Archivo.
- La clase Curriculum contiene 1 objeto de la clase Direccion, que es donde vive.
- La clase Archivo guarda de 0 a n objetos de la clase Curriculum.
- La clase Direccion está asociada con String pues contiene 3 objetos de la clase String.
- Los objetos Curriculum se guardan en la variable privada cvArray de Archivo.

Jerarquías de objetos: Asociación en UML

La asociación se representa mediante una línea o mediante una propiedad y se puede adornar con elementos diversos: nombre, multiplicidad, marcas de agregación, composición y navegación, nombre de propiedades, visibilidad…



Este diagrama dice que:

- La clase Curriculum está asociada con las clases Direccion, Imagen y Archivo.
- La clase Curriculum contiene 1 objeto de la clase Direccion, que es donde vive.
- La clase Archivo guarda de 0 a n objetos de la clase Curriculum.
- La clase Direccion está asociada con String pues contiene 3 objetos de la clase String.
- Los objetos Curriculum se guardan en la variable privada cvArray de Archivo.

Nota sobre UML

Si en un diagrama UML aparece un adorno, cuando se codifica hay que implementar lo que indica el adorno.

Por ejemplo, si señalamos que una propiedad es privada con el signo "-", en la implementación deberá aparecer como privada. Pero, si no ponemos si una propiedad es privada o pública, en la implementación podría aparecer como privada o como pública.

UML tiene muchos adornos, y estos tienen un significado muy concreto.

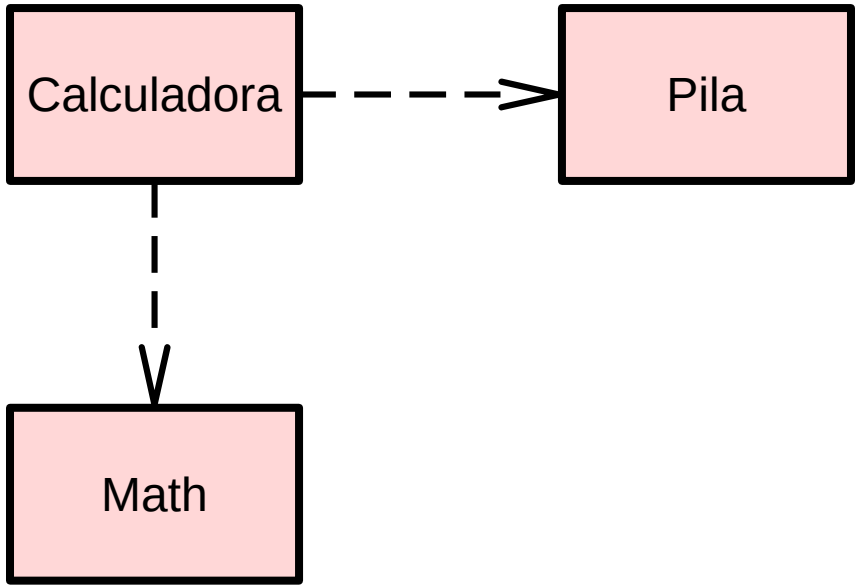
Por tanto, si no sabemos si un adorno es correcto o no es mejor no poner el adorno que ponerlo mal.

La ausencia de adornos denota un diseño pobre, pero al menos no da instrucciones equivocadas.

Jerarquías de objetos: Uso en UML

El uso o dependencia se representa en UML mediante una línea punteada terminada en una flecha.

Un objeto A usa a otro B cuando B no aparece como propiedad en A, pero es necesario B para que funcione A.



Este diagrama dice que:

- La calculadora depende del objeto Math, porque por ejemplo lo utiliza dentro de uno de sus métodos para hacer un cálculo.
- La calculadora depende de la Pila, por ejemplo porque a uno de los métodos le pasan un objeto de Pila como parámetro.

Jerarquías de clases

Las jerarquía de clases se denominan relaciones de herencia.

Definen relaciones del tipo:

La abstracción B es un A

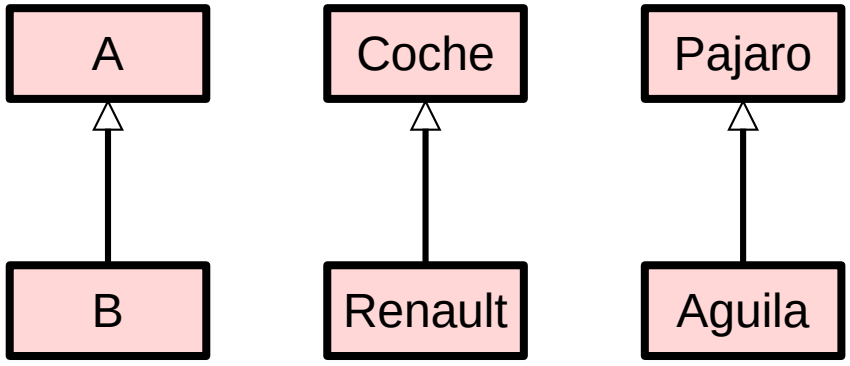
En UML se representan mediante una flecha de punta hueca.

Se cumple que:

- Los elementos de A son más generales
- Los que elementos de B están más especializados

Por ejemplo:

- Un Renault es un Coche
- Un águila es un pajaro



Jerarquías de clases: Tipos de herencia

Se puede hablar de:

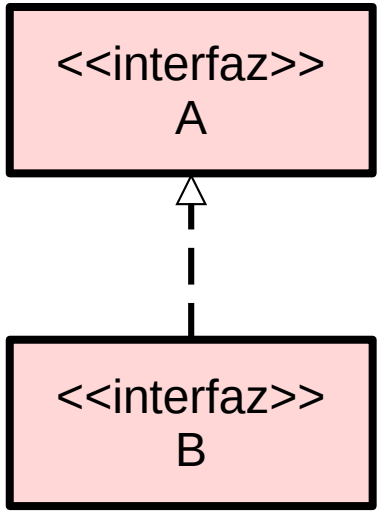
- Herencia entre interfaces
- Herencia entre clase e interfaz
- Herencia entre clases

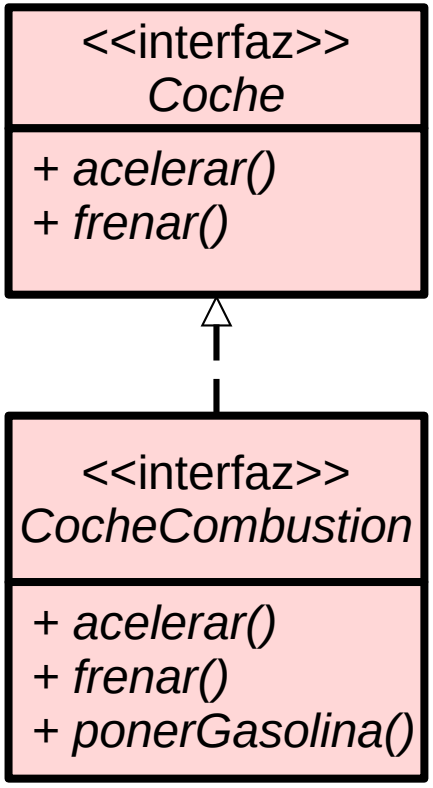
Jerarquías de clases: herencia de interfaz

Las interfaces solo contienen métodos. Cuando una interfaz B hereda de otra A, B solo obtiene la declaración de los métodos de A.

Cuando un método no tiene implementación en UML se pone en cursiva.

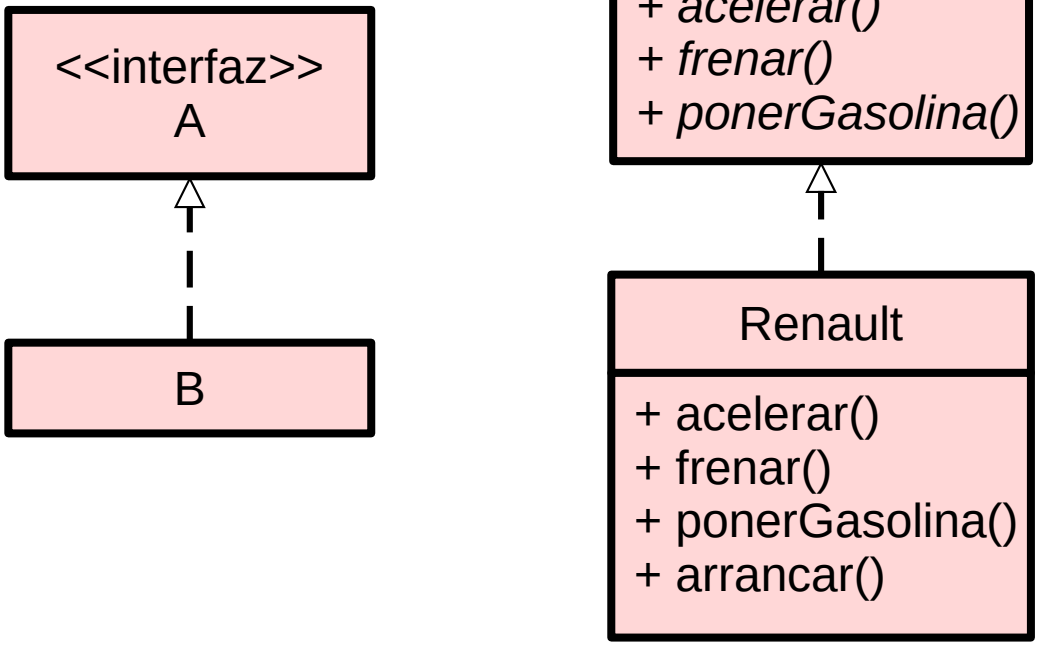
Cuando la relación de herencia involucra una interfaz la línea se dibuja punteada.





Jerarquías de clases: clase-interfaz

Cuando la clase B hereda de la interfaz A se dice que B implementa o cumple A. Solo se hereda la definición de métodos. La clase que hereda implementa el comportamiento.



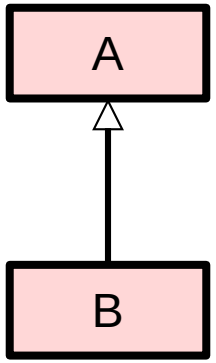
Jerarquías de clases: herencia de clases

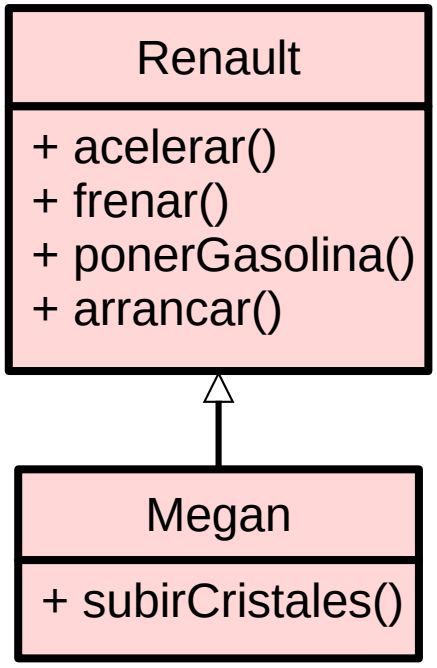
Cuando una clase B hereda tanto la interfaz como el comportamiento de otra clase A.

La clase B tiene todos los métodos y propiedades de A.

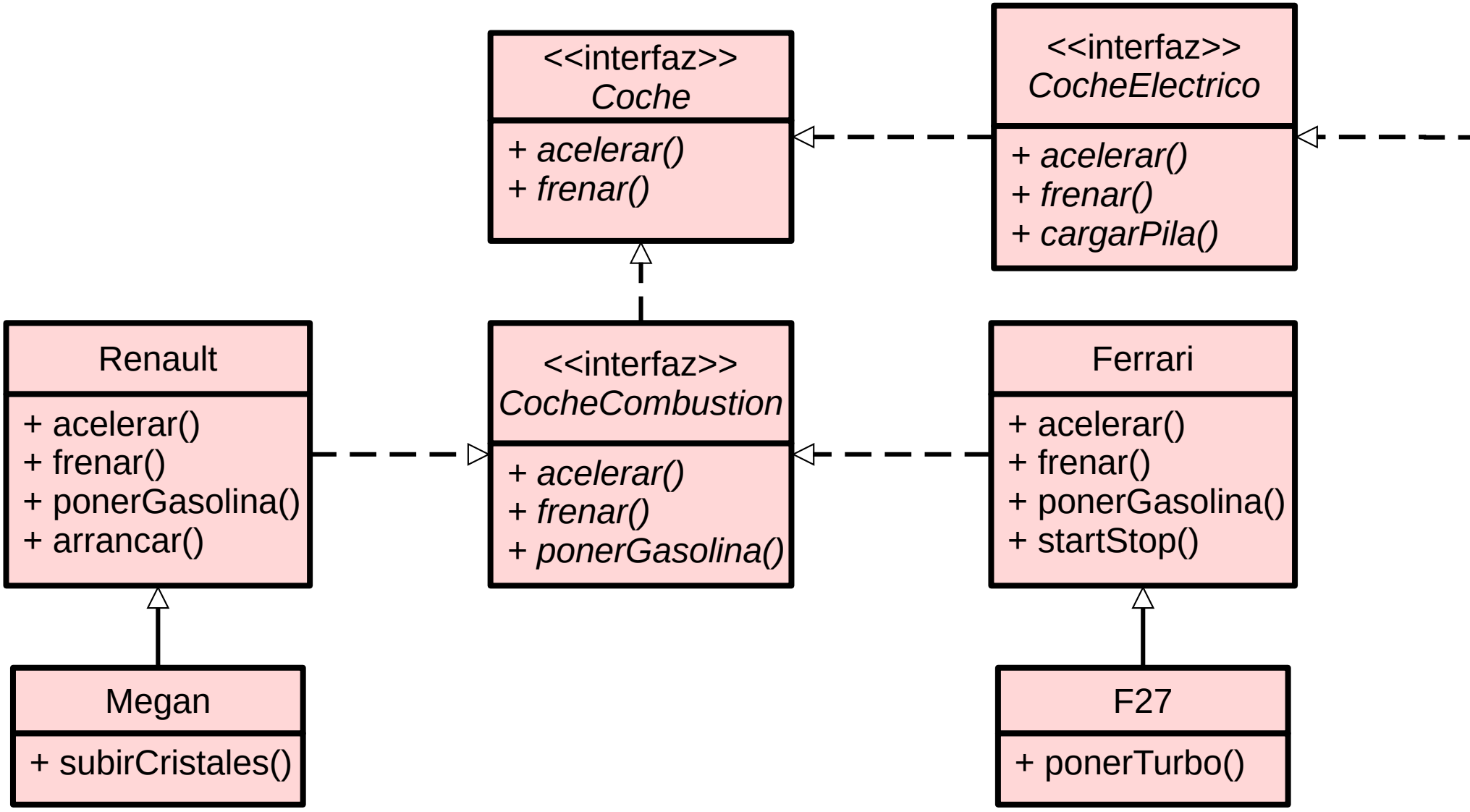
Cualquier llamada a un método de la clase B se comportará como lo haría en A, salvo que se modifique su comportamiento en B.

Cuando la relación de herencia no involucra una interfaz la línea se dibuja continua.





Jerarquías de clases: un ejemplo



Jerarquías de clases

Principales ventajas de crear jerarquías de clases:

- Especializar abstracciones
- Utilizar el polimorfismo

Jerarquías de clases: Especializar abstracciones

Tras heredar siempre se pueden añadir nuevos métodos para especializar la nueva abstracción en algún aspecto.

Si lo que hereda es una clase se puede:

- Añadir nuevas propiedades
- Añadir nuevos métodos
- Implementar algunos o todos los métodos

Si la clase resultante no tiene algún método implementado se dice que es una clase abstracta.

Jerarquías de clases: Polimorfismo

El polimorfismo consiste en permitir utilizar una misma variable para designar objetos de clases diferentes pero que cumplan la misma interfaz.

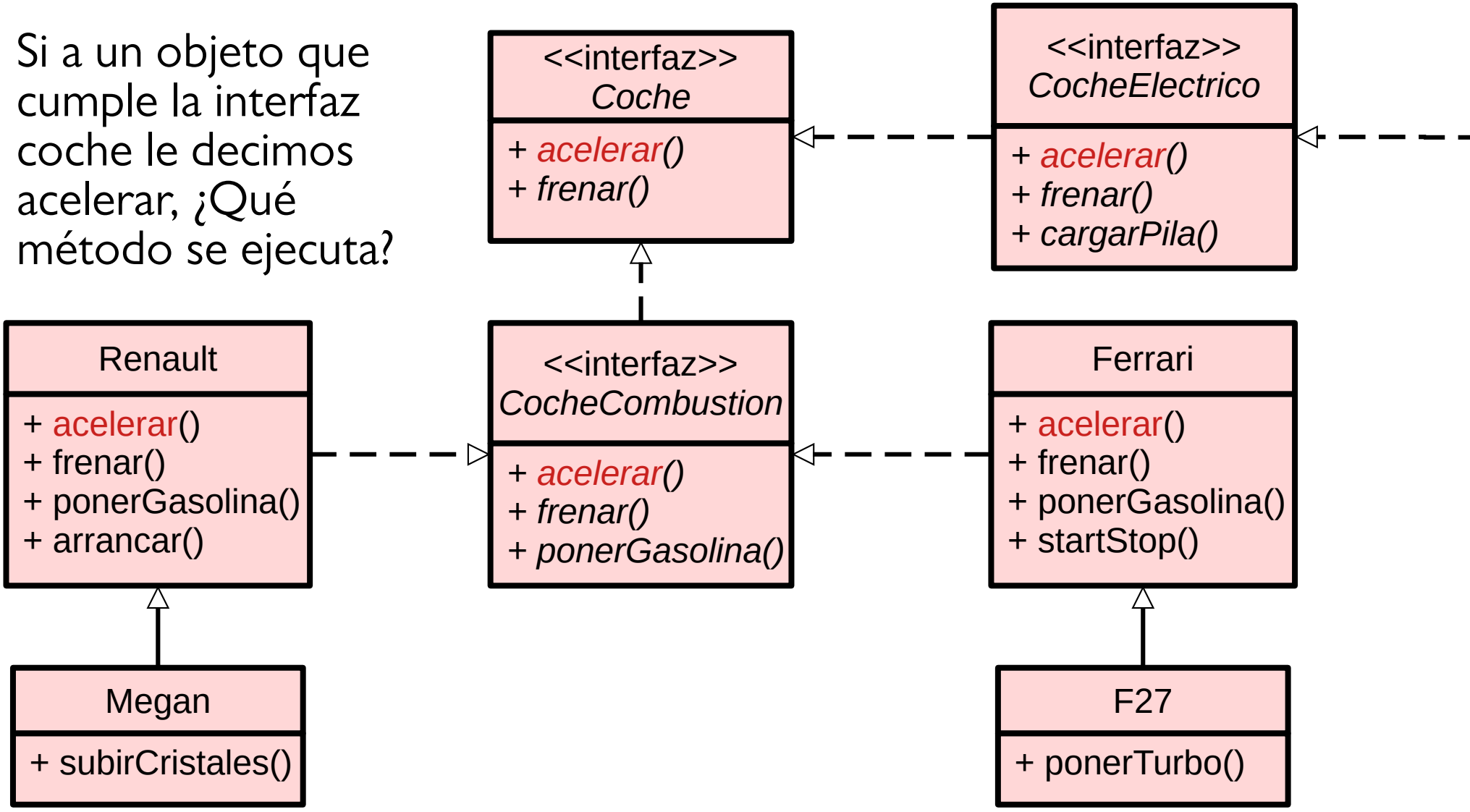
El polimorfismo puede impedir que se sepa en compilación a qué método de qué clase se debe llamar.

Cuando en compilación se puede determinar la dirección del método se habla de polimorfismo estático, y cuando solo es posible determinarla en ejecución se habla de polimorfismo dinámico.

El polimorfismo dinámico también se conoce como transferencia de control indirecto.

Polimorfismo: un ejemplo

Si a un objeto que cumple la interfaz coche le decimos acelerar, ¿Qué método se ejecuta?



Modularidad

La posibilidad de un lenguaje para dividir los programas en módulos.

Modularidad

Definición

La modularidad es la capacidad que permite dividir un programa en agrupaciones lógicas de sentencias llamadas módulos.

En C++ y en Java el concepto de módulo encuentra soporte a varios niveles.

- Al menor nivel cada módulo se corresponde a un fichero. Así, los ficheros se pueden escribir y compilar de manera separada.
- Las bibliotecas aportan un segundo nivel de modularidad a C++. Mientras, en lenguajes como Java, se ha creado el concepto de paquete que permite un número ilimitado de niveles de modularidad aprovechando el concepto de directorio.
- También suele utilizarse el concepto de componente y de programa como módulos que tienen una funcionalidad completa e independiente.

Ventajas de la modularidad

Las ventajas que ofrece la modularidad son:

- Facilidad de mantenimiento, diseño y revisión. Al dividir el programa se facilita que varias personas puedan desarrollar de manera simultánea e independiente conjuntos disjuntos de módulos.
- Aumento de la velocidad de compilación. Los compiladores suelen compilar por módulos. Esto significa que el cambio de un módulo solo implica la recompilación del módulo y de los que dependan de el, pero no la del total de módulos.
- Mejora en la organización y en la reusabilidad, ya que es más fácil localizar las abstracciones similares si se encuentran agrupadas de una manera lógica.

Cómo diseñar los módulos

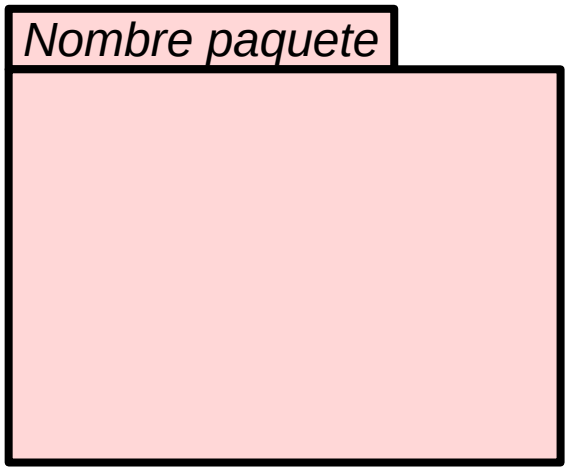
A la hora de diseñar los módulos debe tenerse en cuenta:

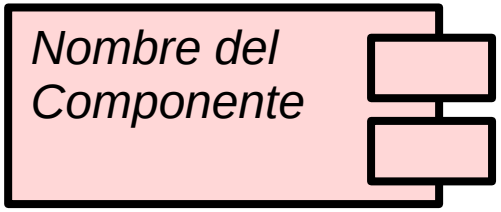
- Maximizar la coherencia. Agrupar en un mismo módulo las abstracciones relacionadas lógicamente.
- Minimizar las dependencias entre módulos. Que para compilar un módulo no se necesite compilar muchos otros.
- Controlar el tamaño de los módulos. Módulos pequeños aumentan la desorganización, módulos muy grandes aumentan los tiempos de compilación y reducen su manejabilidad.

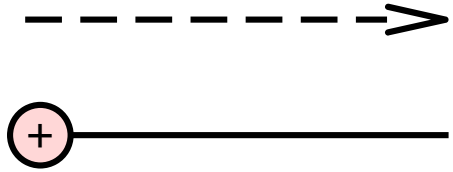
Representación en UML

En UML los paquetes se representan en los Diagramas de Paquetes mediante unos rectángulos que se asemejan a carpetas. Estas carpetas se etiquetan con el nombre del paquete.

Los componentes y los programas se representan utilizando unas cajas decoradas con dos cajas en su interior. Entre paquetes se pueden dar relaciones de asociación y dependencia.

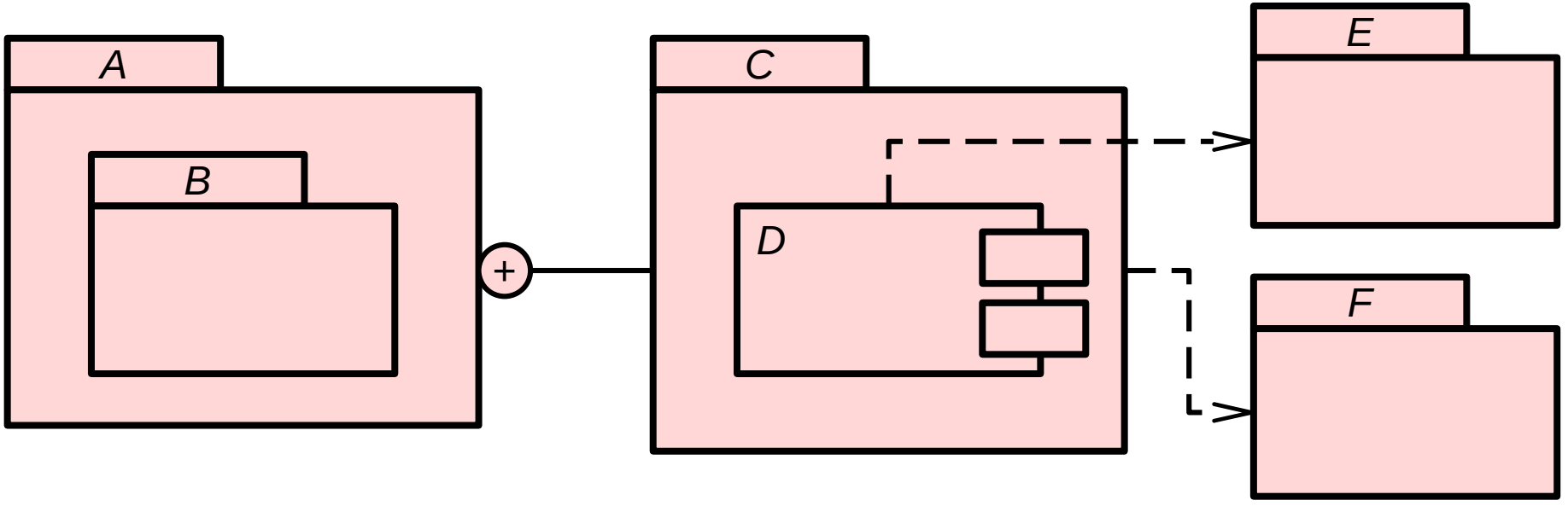






Representación en UML

- El paquete A contiene los paquetes B y C.
- D es un componente del paquete C
- El paquete C usa o depende del paquete F
- El componente D usa o depende del paquete E



Otras características habituales en POO

El tipado, la concurrencia o la persistencia son otras características que habitualmente encontramos en los lenguajes de POO.

Tipado

Definición

Un tipo es una caracterización precisa asociada a un conjunto de datos. La asociación del tipo a un dato se conoce como tipado.

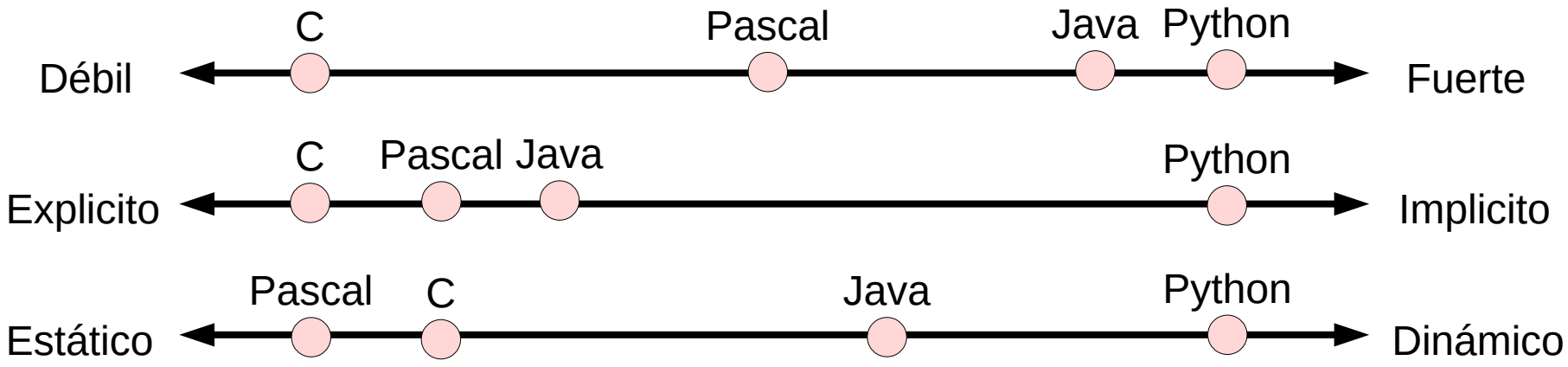
El tipado refuerza las decisiones de diseño, impidiendo que se confundan abstracciones diferentes y dificultando que puedan utilizarse abstracciones de maneras no previstas.

En lenguajes como Java, cada abstracción define un tipo.

Clasificación de tipados

El tipado de un lenguaje se puede clasificar como:

- Débil o fuerte.- Si se puede, o no, cambiar el tipo de un dato.
- Explícito o implícito.- Si hay que declarar el tipo de las variables o no.
- Estático o dinámico.- Si conociendo el tipo de una variable se puede deducir qué código se ejecutará o no.



Tipado explicito, débil y dinámico en C

```c
#include <stdio.h>
#include <stdlib.h>

void fun1(int c) {
    printf("Resultado = %d",c);
}
void fun2(int c) {
    printf("Result = %d",c);
}

main(){
    float a = 25.0;
    void *b = &a;
    int *c = (int*) b;
    void (*fun_ptr)(int) = rand()>0.5?&fun1:&fun2;
    (*fun_ptr)(*c);
}
```

Débil, porque el dato apuntado por a (de tipo float) lo puedo apuntar desde c (de tipo entero).

Explicito, porque hay que declarar el tipo de cada variable.

Dinámico porque en tiempo de compilación no sabemos que función se llamará el puntero a función.

> Resultado = 1103626240

Tipado implicito, fuerte y dinámico en Python

```
>>> a = 24.0
>>> type(a)
<type 'float'>
>>> b = len(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'float' has no len()
>>> a*"Jose"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
>>> d = random.choice([2,3.5])
>>> type(d)
```

Fuerte, porque no hay forma de cambiar el tipo de un dato.

Implícito, porque no hay que declarar el tipo de las variables.

Dinámico porque en tiempo de compilación no sabemos que tipo tendrá d.

Tipado: casting, coerción y conversión

Se suele hablar de casting cuando se asigna un dato de un tipo a una variable de otro tipo.

En realidad casting se puede referir a dos operaciones diferentes:

- Coerción.- Se interpreta el valor de un dato de acuerdo a un tipo sin realizar ninguna otra operación.
- Conversión.- Se ejecuta una función que transforma un dato de un tipo a otro de manera coherente (algunas veces de forma implícita).

```
main(){
    float a = 25.0;
    void *b = (void*) &a;
    int *c = (int*) b;

    printf("Pos a: %p\n",(void*)&a);
    printf("Pos b: %p\n",(void*)b);
    printf("Pos c: %p\n",(void*)c);

    printf("Val a: %f\n",a);
    printf("Coerción c: %d\n",*c);
    printf("Conversion a: %d\n",(int)a);
}
```

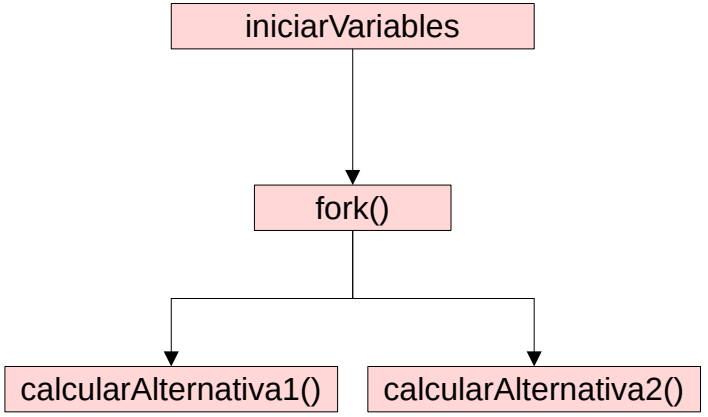
```
Pos a: 0x7ffd87032a6c
Pos b: 0x7ffd87032a6c
Pos c: 0x7ffd87032a6c
Val a: 25.000000
Coerción c: 1103626240
Conversion a: 25
```

Definición

La concurrencia es la capacidad que permite la ejecución simultánea de varias secuencias de instrucciones. La concurrencia permite que un programa tenga varios puntos de ejecución simultáneamente.

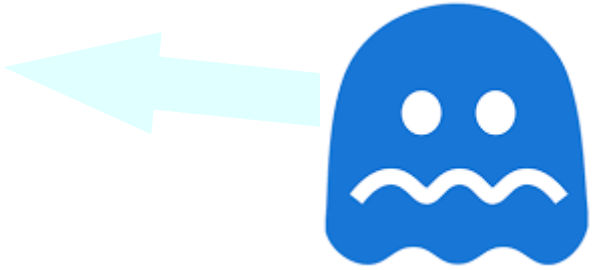
Clásicamente, los lenguajes de programación no daban soporte a la concurrencia sino que era proporcionada por los sistemas operativos.

En Unix la concurrencia se consigue con la invocación de una función del sistema operativo llamada fork que divide la línea de ejecución, creando múltiples líneas de ejecución (también conocidas como hilos o threads).

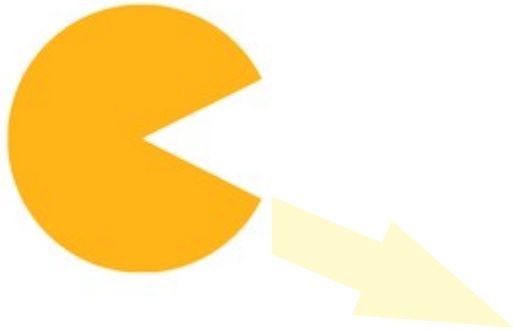


La concurrencia en los lenguajes de POO

Los lenguajes de POO pueden dar soporte a la concurrencia de una manera natural haciendo que un objeto se pueda ejecutar en un thread separado. A tales objetos se les llama objetos activos, frente a los pasivos que no se ejecutan en threads separados.



Java da soporte a la concurrencia creando hilos al crear ciertos objetos, aunque en el caso de Java el hilo puede ejecutar luego código que esté en otros objetos.



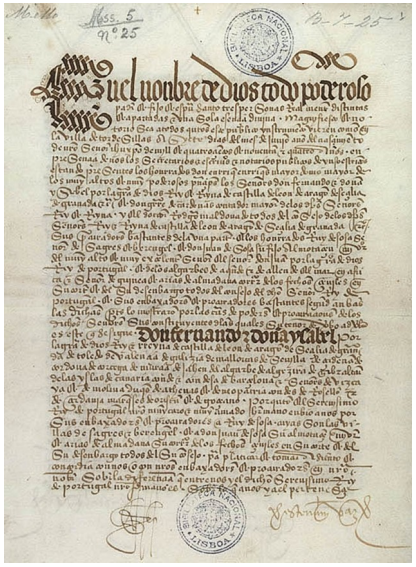
Persistencia

Definición

La persistencia es la capacidad que permite que la existencia de los datos trascienda en el tiempo y en el espacio.

En relación con su tiempo de vida, los datos se pueden catalogar en:

- Expresiones, con una vida inferior al de una línea de código.
- Variables locales, cuya vida se circunscribe a una función.
- Variables globales, que viven mientras se ejecuta un programa.
- Datos que persisten de una ejecución a otra.
- Datos que sobreviven a una versión de un programa.
- Datos que sobreviven cuando ya no existen los programas, los sistemas operativos e incluso los ordenadores en los que fueron creados.



Los lenguajes de POO suelen dar soporte a todos usando ficheros y bases de datos.

Ejemplo

```
const capacity = 13;

type Stack = record
    top: 0 .. capacity;
    content: array [1 .. capacity] of string[7]
end;

procedure Init (var s: Stack);
begin
    s.top := 0
end;

function Size (s: Stack): integer;
begin
    Size := s.top
end;

function Empty (s: Stack): boolean;
begin
    Empty := (s.top=0)
end;

procedure Push (var s: Stack; x: string[7]);
begin
    ASSERT(s.top < capacity);
    s.top := s.top + 1;
    s.content[s.top] := x
end;

procedure Pop (var s: Stack; var x: string[7]);
begin
    ASSERT( s.top>0 );
    x := s.content[s.top];
    s.top := s.top - 1
end;
```

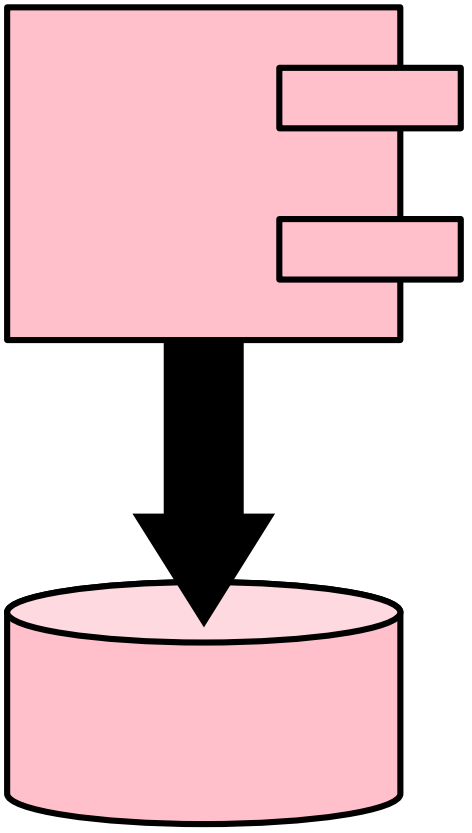
```
procedure storeStack(var s: Stack);
begin
    ??????????????
end;

function loadStack (fileName: string): Stack
begin
    ??????????????
end;
```

La persistencia en los lenguajes de POO

Un lenguaje de POO con soporte para la persistencia debe permitir grabar los objetos, así como la definición de sus clases, de manera que puedan cargarse más tarde sin ambigüedad, incluso en otro programa distinto al que lo ha creado.

Java da cierto nivel de soporte a la persistencia de una clase si esta cumple la interfaz Serializable.



El ejercicio de la báscula

Un ejercicio para practicar los conceptos presentados.

El ejercicio de la báscula

Planteamiento

Se desea desarrollar un programa para gestionar el uso de una balanza de supermercado como la de la figura por parte de los clientes del supermercado.

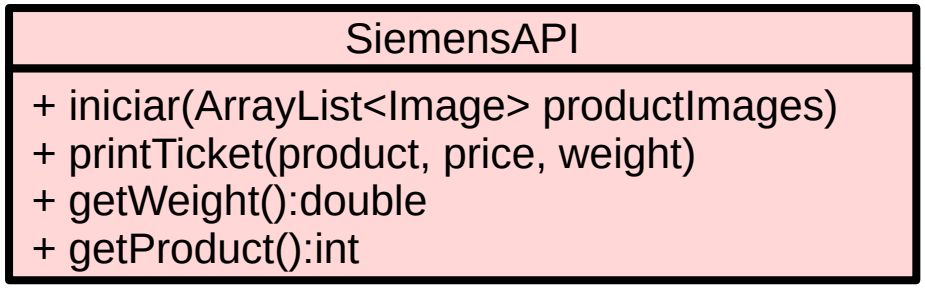
- Básicamente, la balanza muestra una pantalla con los diferentes elementos que se podrían pesar.
- El cliente primero sitúa el producto sobre la balanza y luego selecciona uno de los productos que se muestran en pantalla.
- Finalmente, la balanza imprimiría un ticket indicando el tipo de producto, el peso y el precio.

El ejercicio de la báscula

Control del hardware

El fabricante del hardware de la báscula proporciona una clase llamada SiemensAPI que dispone de los siguientes métodos:

- Iniciar: método para iniciar la báscula que recibe una lista de imágenes y con ella crea una tabla que muestra dichas imágenes.
- PrintTicket: método que imprime un ticket con los datos que se le pasan.
- getWeight: método que devuelve el peso del objeto que está situado sobre la balanza mediante un número flotante de precisión doble en kilogramos.
- getProduct(): método que bloquea la ejecución hasta que alguien pulsa la pantalla. En ese momento devuelve un entero con la posición del producto pulsado respeco a la lista que se pasó en el método de iniciar.



El ejercicio de la báscula

Preguntas

a) El sistema debe almacenar en alguna estructura de datos los nombres, precios y pesos de todos los productos. ¿Qué clases propones para ello?

b) Siempre que la balanza esté en marcha el programa estará esperando a que alguien pulse la pantalla (bloqueado en una llamada al método getProduct). Luego, cuando alguien pulse una tecla se buscará el producto pulsado en la estructura de datos para saber su precio, se revisará el peso del artículo sobre la balanza, se calculará el precio y se imprimirá el ticket. ¿Qué objeto podría realizará esta tarea?

c) ¿Se podría crear un diseño que permita ampliar el listado de productos sin más que añadir imágenes a un directorio y el producto, nombre y precio por kilo, a un fichero de texto?

Referencias

- UML Gota a gota, Martin Fowler, Pearson,1999 (libro)
- Object-Oriented Analysis and Design, G. Booch, Benjamin Cummings, 1994 (libro)
- El lenguaje Unificado de Modelado, G. Booch, I.. Jacobson y J. Rumbaugh, Addison Wesley 1999 (libro)
- Paradigmas de programación (wikipedia)
- Comparación de los paradigmas de programación (wikipedia)
- Sobre la semántica de la navegabilidad en las asociaciones (PDF)
- OMG - Unified Modeling Language (metamodelo) v 2.5.1 (PDF)