



Grado en Ingeniería Informática
Grado en Matemáticas e Informática



Asignatura: PROGRAMACIÓN II

Programación Modular

Clara Benac Earle, Manuel Collado
Ángel Lucas González Martínez,
Jaime Ramírez, Guillermo Román

DLSIIS - E.T.S. de Ingenieros Informáticos
Universidad Politécnica de Madrid

Octubre 2013

LOS MÓDULOS EN JAVA

Módulos

- Es un mecanismo del que disponen prácticamente todos los lenguajes de programación
- Permite separar el código de un programa en unidades o compartimentos.
- Un módulo en java puede estar definido por:
 - ➔ Una **clase** (fichero .java)
 - ➔ Un **paquete** que puede contener varias clases (carpeta)

¿Por qué dividir en módulos?

- Es más fácil de entender el programa
- Es más fácil de encontrar algo
- Si la división se hace de una forma apropiada, se puede reducir mucho el esfuerzo de realizar ciertas modificaciones en el programa
- Facilita el reparto del trabajo
- Facilita la reutilización del código: un mismo módulo se puede utilizar en distintas aplicaciones.

Un poco de terminología

- Un módulo (clase o paquete) ya existente se dice que ofrece unos **servicios**.
 - ➔ El módulo que los usa se dice que es un **cliente del módulo servidor o proveedor de servicios**.
- Los **servicios** de una clase son sus métodos públicos.



Módulos: Paquetes

- Un paquete permite agrupar en una misma unidad varias clases e interfaces.
- Un paquete puede contener otros paquetes.
- Un paquete crea un espacio de nombres \Rightarrow
 - **Dos clases pertenecientes a distintos paquetes se pueden llamar igual**
- Si no se especifica nada la clase pertenece al paquete por defecto
- Si trabajamos en eclipse, cuando creamos un paquete **a** dentro de un proyecto, se crea una carpeta **a** en el sistema de ficheros en la que se incluyen todos los ficheros del paquete.

Módulos: Cómo crear un paquete

- Por convenio se usan minúsculas en el identificador del paquete
- Al comienzo de todos los ficheros pertenecientes a un mismo paquete 'a':

***package* a;**

- Si el paquete 'a' está dentro del paquete 'b', y éste a su vez está dentro de 'c', entonces:

***package* c.b.a;**

- Una forma habitual de nombrar los paquetes es:
 - ➔ Tipo de organización: org
 - ➔ Nombre Organización: upm
 - ➔ Proyecto: programacion2
 - ➔ Detalles proyecto: practica1
 - ➔ Otras divisiones ...

Módulos: Cómo usar un paquete

- Escribiendo la referencia completa al elemento (importación calificada):

`nombrePaquete.NombreElem`

Es necesario cuando hay ambigüedad

- Importando (importación no calificada):

→ el paquete entero (todas las clases):

import `nombrePaquete.*;`

→ o sólo el elemento a ser utilizado:

import `nombrePaquete.NombreElem;`

- La importación permite usar el conjunto de identificadores definidos. **No se trae el código**

Módulos: Cómo usar un paquete

```
import puntos.Punto2; //si se comenta se usa importación
                      //calificada Punto2 -> puntos.Punto2
import consola.Consola;
public class TestPuntos2 {
    static Punto2 leerPunto () {//leerPunto
        float x,y;
        .....
        return new Punto2 (x,y);
    }//leerPunto

    static void imprimirPunto (puntos.Punto2 p)
    {//imprimirPunto
        System.out.printf("La coordenada x del punto es %f\n",
                           p.getX());
        System.out.printf("La coordenada y del punto es %f\n",
                           p.getY());
    }//imprimirPunto
}
```

(importación no calificada)
Se importa la clase Consola

Calificada

Módulos: Cómo crear una librería

- Una librería es un fichero .jar o .zip que contiene una o más clases que pueden estar agrupadas en paquetes (y opcionalmente otros ficheros).
- Una librería puede ser utilizada en más de un programa (proyecto).
- Para crear una librería en eclipse debemos crear un fichero jar/zip a partir de un proyecto.
 1. Seleccionamos los paquetes del proyecto que queremos incluir
 2. Indicamos si solo vamos a incluir el código compilado (.class) o también el código fuente (.java)

Módulos: Cómo usar una librería

- Supongamos que queremos utilizar en un proyecto de eclipse alguna clase definida en una librería.
- Para usar una librería en un proyecto debemos añadirla al proyecto como “archivo externo”.
 - ➔ seleccionamos el fichero jar/zip en el sistema de ficheros.
 - ➔ La librería no se copia al proyecto, sino que se crea un vínculo o enlace a un fichero ya existente.

Variable CLASSPATH

- Es una variable de entorno del Sistema Operativo.
- Debemos preocuparnos de ella si vamos a ejecutar un programa en java FUERA de Eclipse.
- Supongamos que el programa a ejecutar utiliza clases del paquete **a**
- Todos los ficheros del paquete **a** deben:
 - ➔ Estar localizados en un directorio llamado **a** cuyo path esté incluido en la variable CLASSPATH (por ejemplo: *c/b/a*) o
 - ➔ estar incluidos en una carpeta **a** dentro de un fichero de librería .jar o .zip, y el path del fichero debe estar incluido en la variable CLASSPATH (por ejemplo: */misclases/libreria.jar*)

OCULTACIÓN DE INFORMACIÓN

Necesidad de Ocultación

- Al implementar un módulo, es posible ocultar ciertos elementos del módulo a los otros módulos.
 - ➔ Un módulo va a tener una parte pública y una parte privada
- La parte pública recibe el nombre de **interfaz**
- La parte privada está constituida por los **detalles de implementación**
- ¿Por qué se introduce esta separación?
 - ➔ reducir el esfuerzo de realizar cambios en el código
 - ➔ facilitar el reparto del trabajo
 - ➔ facilitar la reutilización de librerías

Partes pública y privada de una clase

Parte Pública

- Cabeceras de los métodos públicos
- Semántica de los métodos: explica qué hacen

Normalmente, esta información es la que se incluye en la documentación javadoc

Parte Privada

- Importación de paquetes
- Atributos
- Implementación de los métodos públicos: se especifica un cuerpo {...}
- Métodos auxiliares

Ocultación: facilitar cambios

- Supongamos que tenemos una clase Fecha con atributos públicos:

```
public class Fecha{  
    public int dia;  
    public int mes;  
    public int anio;  
    public FechaPublico(int dia, int mes, int anio) {  
        .....  
    }  
    public int darDia() {  
        .....  
    }  
}
```

Posibles Puntos de acoplamiento

The diagram highlights three specific points in the code as potential coupling points: the public attributes (dia, mes, anio), the constructor method (FechaPublico), and the public method (darDia). Each point is enclosed in a green box, and a line from the 'Posibles Puntos de acoplamiento' box points to each of these three locations.

- ¿Qué puede hacer un programa cliente con esta clase?

Ocultación: facilitar cambios

```
public class TestFecha {  
  
    public static void main(String[] args) { //MAIN  
        fecha.Fecha fecha1;  
        fecha1 = new fecha.Fecha(30, 5, 2005);  
        System.out.println("La primera fecha es " + fecha1);  
        fecha1.mes = 2; //se crea una fecha incorrecta!!  
        System.out.println("La primera fecha es " + fecha1);  
  
        fecha1 = new fecha.Fecha(30, 2, 2005);  
    } //MAIN  
} //TestFecha
```

Puntos de acoplamiento

Ocultación: facilitar cambios

- Supongamos que modificamos la clase Fecha para que el atributo mes pase a ser un enumerado (Enero, Febrero,...) ⇒
 - ➔ Aparece un error de compilación en la línea `fecha1.mes = 2`
- Supongamos también que la clase Fecha forma parte de una librería que se utiliza en cientos de programas clientes que incluyen líneas como la anterior ⇒
 - ➔ Tendremos cientos de errores de compilación como el anterior en estos programas clientes
 - ➔ Al final cambiar el atributo mes en la clase Fecha va a dar mucho trabajo!!

Ocultación: facilitar cambios

- ¿Cómo podíamos haber reducido el esfuerzo de cambiar el atributo mes?
- SOLUCION: Se deberían haber ocultado los atributos a los programas clientes.
 - ➔ Los atributos de la clase Fecha deberían haber sido privados desde el principio
 - ➔ Así solo habríamos tenido que modificar la clase Fecha
- CONCLUSION: declarar los atributos como privados es una buena práctica de programación que nos ahorrará trabajo en el futuro.

Ocultación: facilitar el reparto del trabajo

- Cuando se comienza a desarrollar en equipo una aplicación de tamaño medio o grande, se suele hacer un primer diseño:
 - ➔ descomposición en módulos interdependientes
 - ➔ se definen las interfaces o “anclajes” entre los módulos
 - ➔ No se entra en los detalles de implementación de cada módulo, solo se especifica qué servicios debe ofrecer a los otros módulos.
 - ➔ Analogía con la fabricación de un coche
 - ★ un coche se puede dividir en módulos: motor, carrocería, luces, etc.
 - ★ es necesario definir cómo van a estar unidos estos módulos

Ocultación: facilitar el reparto del trabajo

- La creación de los módulos se puede repartir entre diferentes personas
 - ➔ pueden trabajar en paralelo en varios módulos
- Para que un módulo cliente funcione bien, necesita que sus módulos servidores implementen sus servicios de acuerdo a lo estipulado en sus interfaces.
 - ➔ Analogía del coche: El aire acondicionado requiere que la batería le proporcione una cierta corriente eléctrica para funcionar.
 - ➔ Si algún módulo no se implementa de acuerdo a la interfaz que se especificó para él, no se podrá integrar bien con los otros módulos.

Ocultación: facilitar la reutilización de librerías

- Si el programador de un módulo cliente necesita utilizar uno de los módulos de una librería:
 - ➔ Solo necesita conocer la interfaz de ese módulo (documentación de lo que ofrece).
 - ➔ No necesita conocer para nada los detalles de implementación.
- Si no existiera la separación entre interfaz e implementación, el programador tendría que hacer un mayor esfuerzo para poder utilizar la librería.
 - ➔ tendría que estudiar el código de la librería para averiguar cómo la puede aprovechar.

Separación física de la parte pública y privada en java

- La parte pública y la privada están mezcladas en el código fuente (ficheros .java).
- Para no tener que ver la parte privada cuando se quieren usar los servicios de una clase, se utiliza la **documentación javadoc en ficheros html**.
- En la documentación javadoc se encuentra la descripción de la parte pública, **que incluye todo lo que necesitamos saber para usar los servicios de una clase**.
- Para poder generar esta documentación, tenemos que incluir los comentarios apropiados en el código fuente (**/** ... */**) para cada método público.

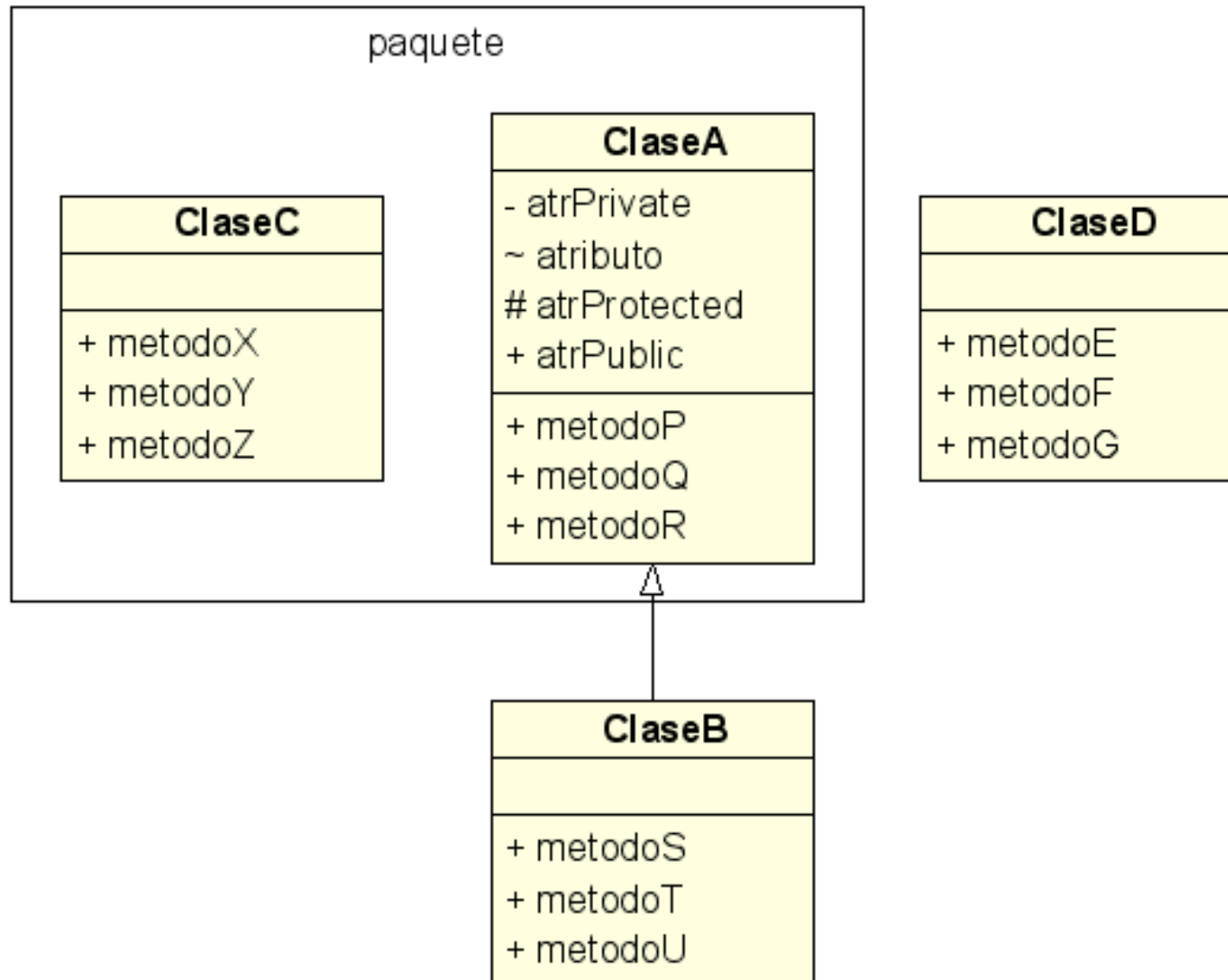
Ocultación entre módulos

- En la parte pública de una clase se pueden distinguir distintos niveles de visibilidad: *public* o *friendly*.
- Los tres niveles de visibilidad o modificadores se definen respecto a cuatro ámbitos distintos:

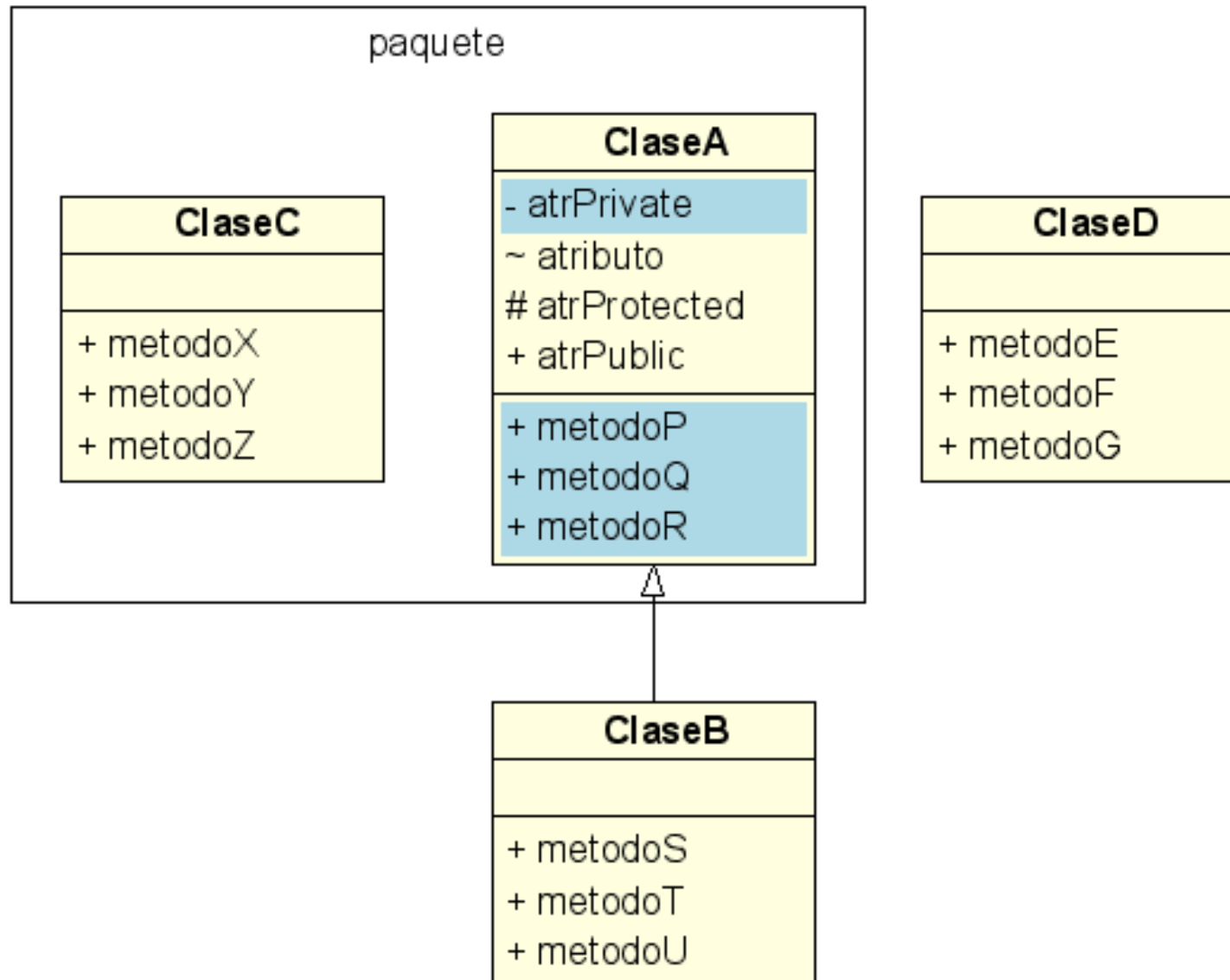
Modificador	Clase	Paquete	Subclase	Universo
public	Si	Si	Si	Si
Nada (friendly)	Si	Si	No	No
private	Si	No	No	No

- Existe un cuarto modificador (*protected*) que se estudiará cuando se vea la herencia de clases.

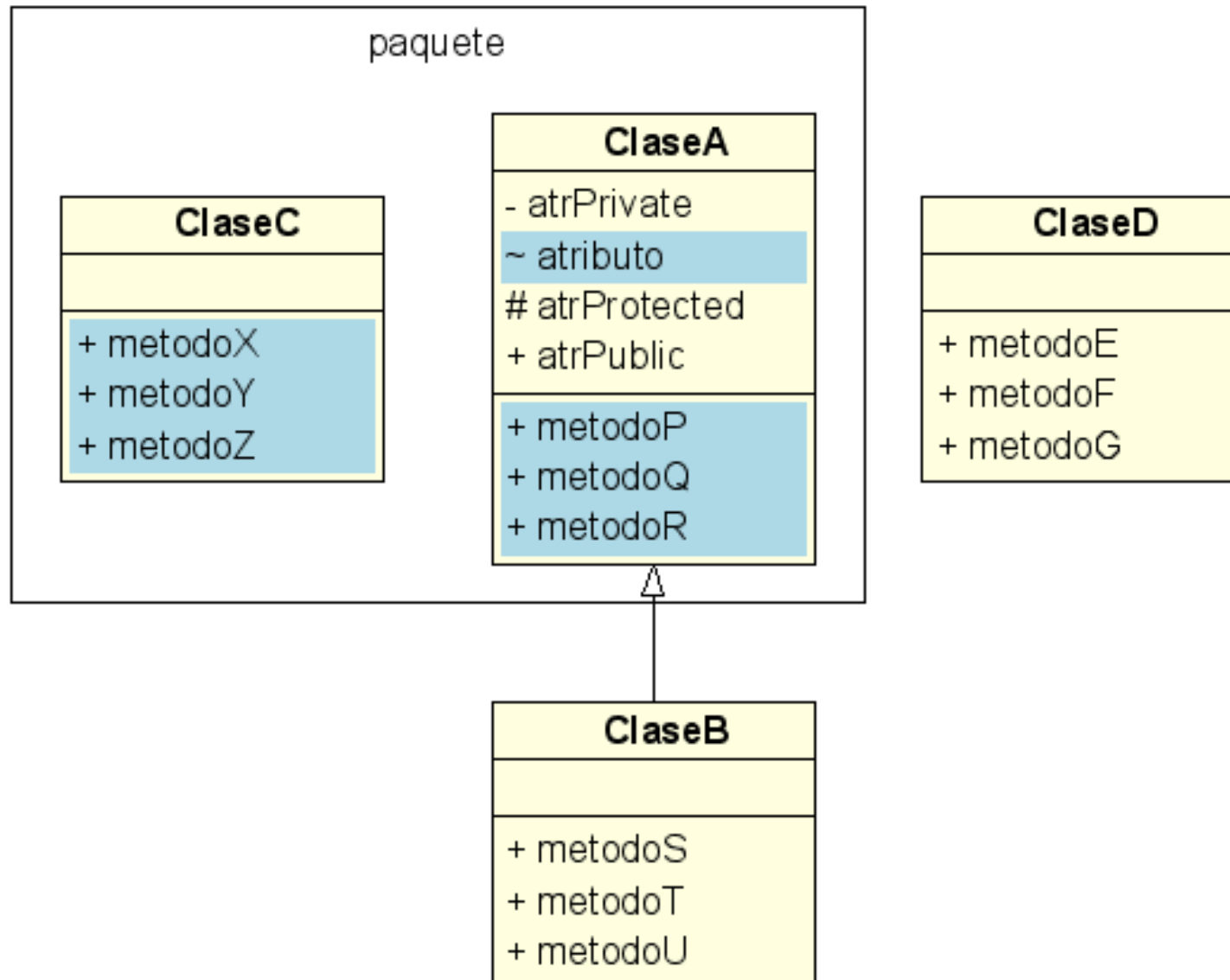
Ocultación entre módulos



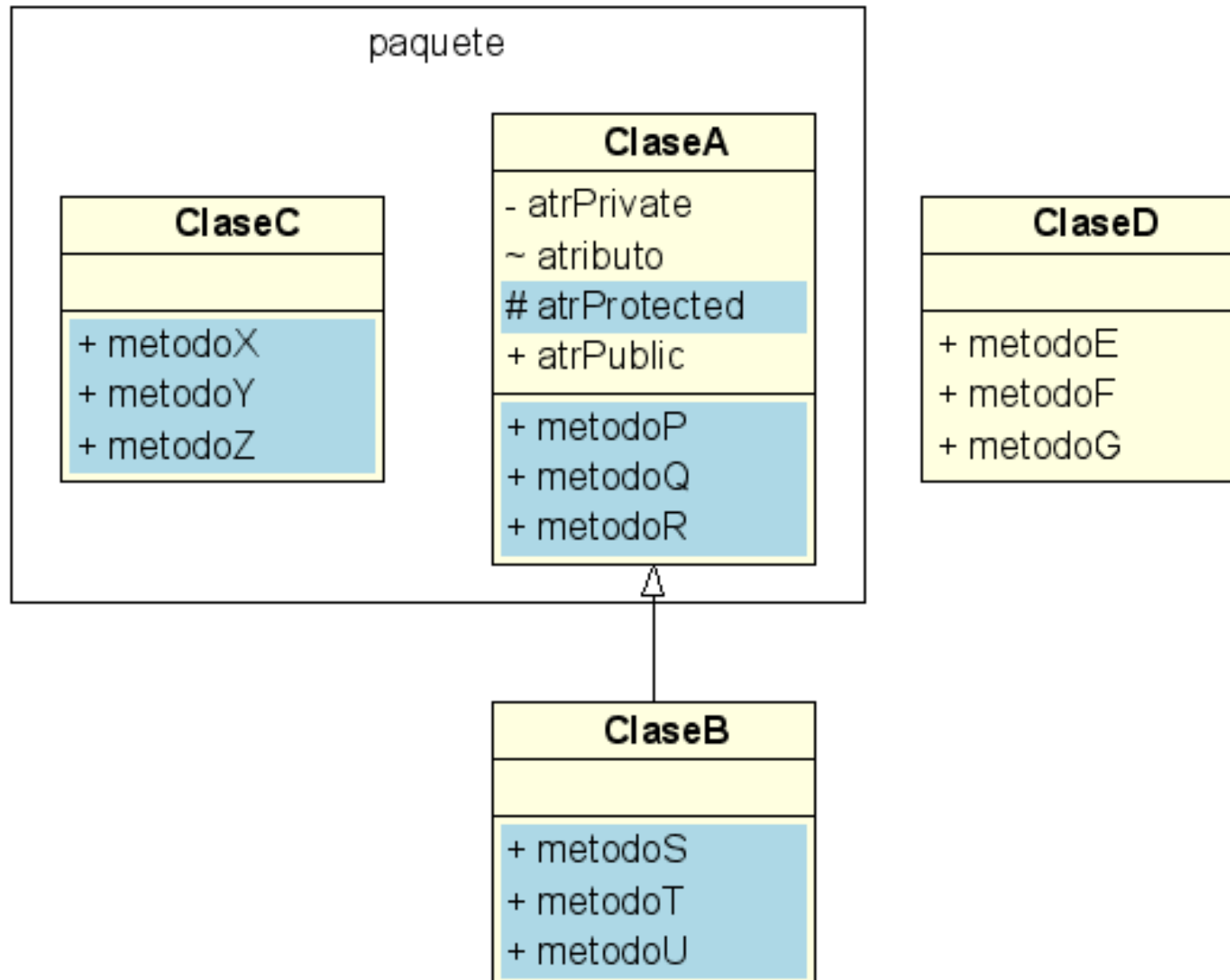
Ocultación entre módulos



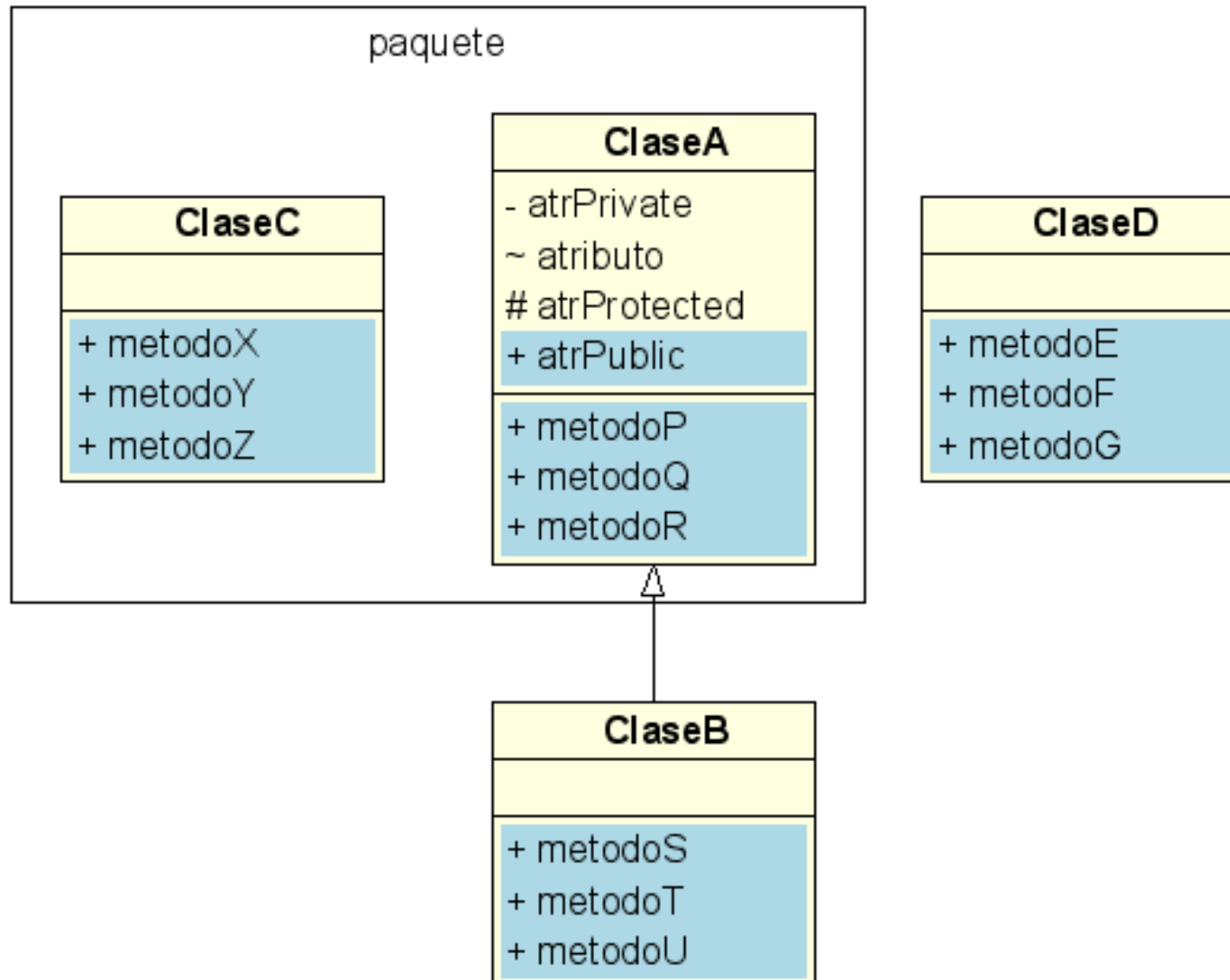
Ocultación entre módulos



Ocultación entre módulos



Ocultación entre módulos



Recomendaciones sobre control de acceso

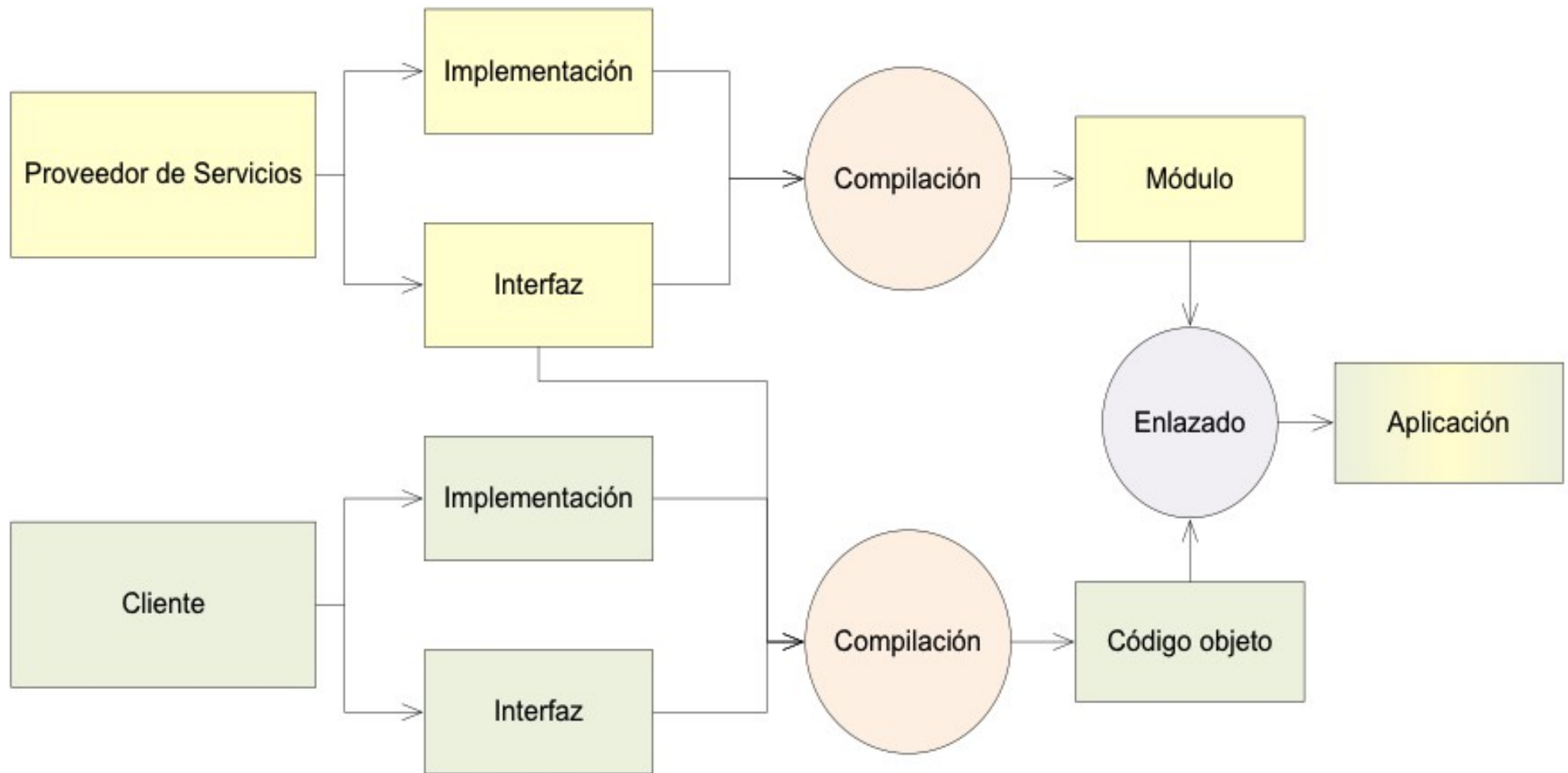
- Los **atributos de instancia** deben ser **siempre privados**.
- Los **atributos de clase** que representen constantes pueden ser públicos, si se consideran de propósito general (e.g. Math.PI).
- Los **métodos** deben ser:
 - ➔ **private**: si son de apoyo para el desarrollo de la clase
 - ➔ **“friendly”**: si se desea que sólo sean utilizados dentro del paquete.
 - ➔ **public**: si se desea permitir el acceso a ellos desde cualquier parte del programa.

Cohesión y Acoplamiento

- La **cohesión** y el **acoplamiento** son dos conceptos esenciales para la correcta organización del software
 - ➔ sirven de guía al realizar la descomposición modular
- La **cohesión** habla del grado de relación (similitud) que existe entre los elementos de un mismo módulo
- El **acoplamiento** habla del grado de relación (dependencia) que existe entre módulos diferentes
- Los objetivos a perseguir son:
 - Maximizar la cohesión
 - Minimizar el acoplamiento
- La separación entre interfaz e implementación reduce los puntos de acoplamiento entre dos módulos ⇒
 - ➔ ayuda a minimizar el acoplamiento

COMPILACIÓN DE MÓDULOS

Compilación separada



Compilación separada en Java

