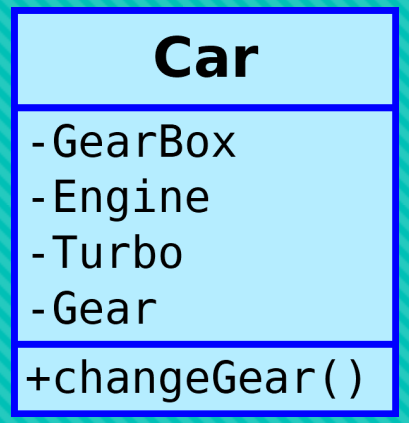


UA5. Elaboración de diagramas de clases

Entornos de Desarrollo – 1ºDAM

Luis del Moral Martínez

versión 21.03

Contenidos del tema

Contenidos del tema

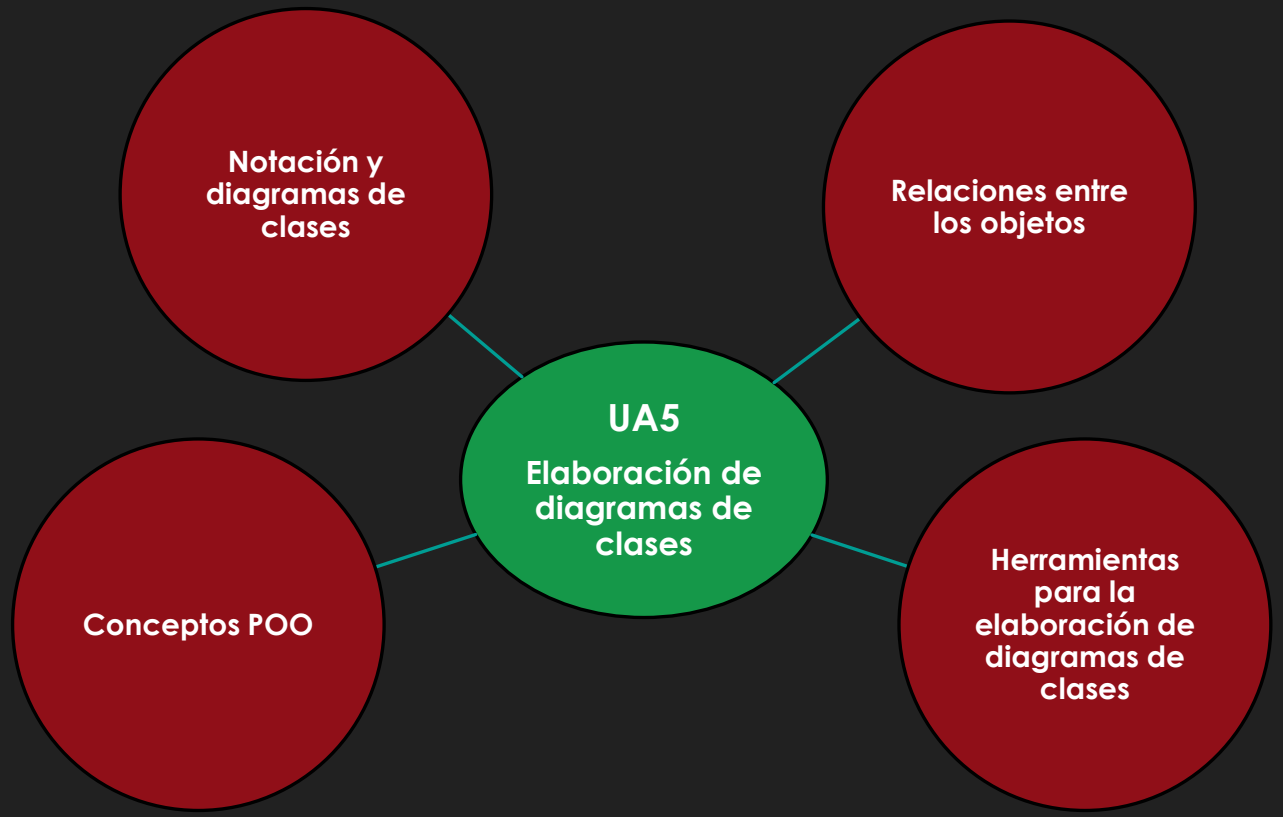
5. **Recomendaciones finales**
   - 5.1 Reutilización de código
   - 5.2 Extensibilidad
   - 5.3 ¡Encapsula!
   - 5.4 ¡Usa interfaces!
   - 5.5 Composición frente a herencia
6. **¿Y ahora qué?**
   - 6.1 Ejemplos y ejercicios

Contenidos de la sección

1. **Introducción**
   - 1.1 Mapa conceptual del tema
   - 1.2 ¿Para qué sirve la POO?
   - 1.3 Algunos conceptos importantes

1. Introducción

1.1 Mapa conceptual del tema



1. Introducción

1.2 ¿Para qué sirve la POO?

- La programación estructurada se centraba más en la **funcionalidad** que en los **datos**
- El programador se centraba en **"qué pasos debía seguir"** para resolver el problema
- La POO representa los elementos del problema que vamos a resolver
- De esta forma, se crean **objetos** que tienen **atributos** y **comportamiento**
- Los objetos interactúan entre si enviándose **mensajes** (paso de mensajes)

1. Introducción

1.2 ¿Para qué sirve la POO?

- **Ciclo de vida de un programa orientado a objetos**
  1. Creación de los **objetos** (a medida que son necesarios)
  2. El usuario **interactúa** con el programa y los objetos **intercambian mensajes**
     - Es posible que se **creen** nuevos objetos o se **destruyan** algunos existentes
  3. Los objetos que no son necesarios se **destruyen** (ya sea de forma programada o automatizada)
     - En Java existe un sistema encargado de la destrucción de **objetos no utilizados** (Garbage collector)

1. Introducción

1.3 Algunos conceptos importantes

- **Atributo**: característica de una clase concreta (con un rango propio). Las clases tienen 0+
- **CASE**: herramienta automatizada para el desarrollo y normalización de proyectos (desde 70s)
- **Clase**: tipo de "cosas". Los objetos de la misma clase tienen atributos y comportamiento similar
- **Objeto**: los objetos se crean con el molde de la clase (los prototipos)
- **Instancia anónima**: una instancia de una clase que carece de nombre
- **Jerarquía (de clases)**: grupo de clases que poseen una relación de parentesco

1. Introducción

1.3 Algunos conceptos importantes

- **Mensaje**: cuando un objeto ejecuta un método, en realidad "pasa un mensaje"
- **Modelo UML**: muestra cómo "es" el sistema, pero no "cómo se implementará"
- **OMG**: organismo que promueve y fomenta la POO y mantiene UML
- **OO**: orientación a objetos
- **POO**: programación orientada a objetos
- **UML**: lenguaje unificado de modelado

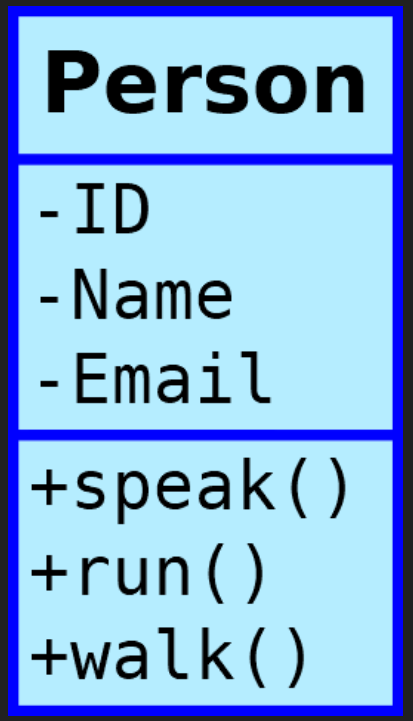
Contenidos de la sección

2. **Breve repaso de conceptos de la POO**
   - 2.1 El paradigma de POO
   - 2.2 Pilares de la POO

2. Breve repaso de conceptos de la POO

2.1 El paradigma de POO

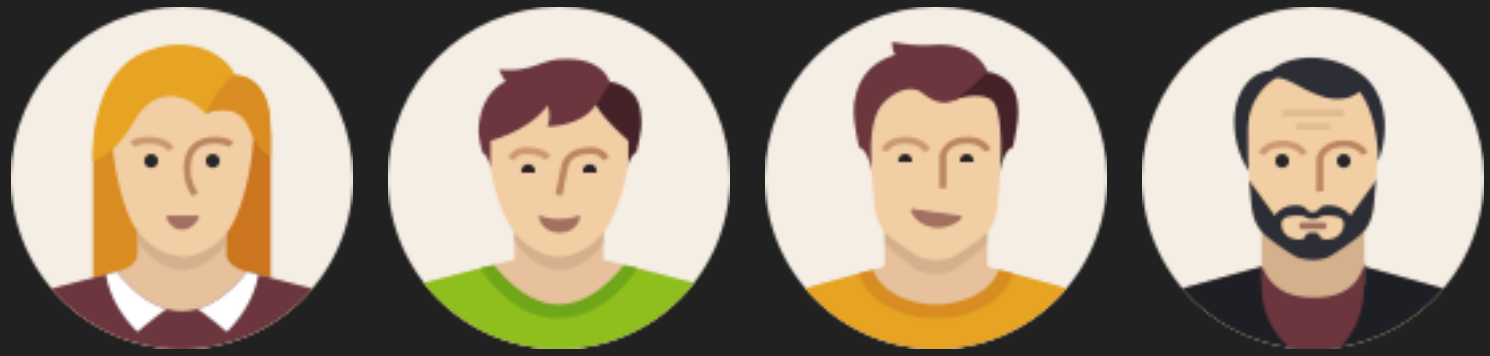
- Es un **paradigma** de programación
- Envuelve **información** y **comportamiento** en **objetos**
- Los objetos se construyen desde una plantilla llamada **clase**
- Una **clase** se compone de **información** y **comportamiento**

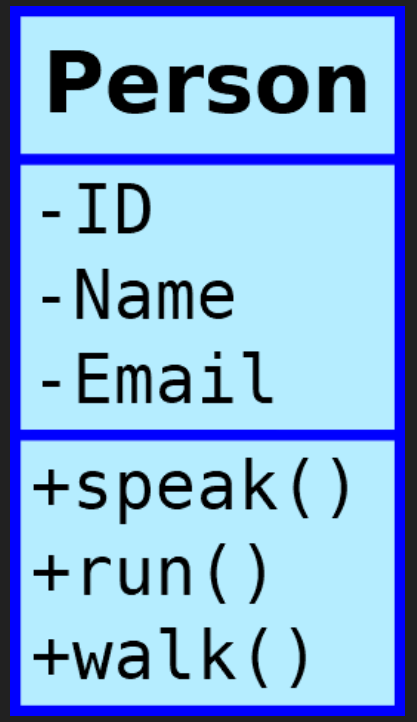


2. Breve repaso de conceptos de la POO

2.1 El paradigma de POO

- **Clases y Objetos**
  - La clase representa un **concepto**
  - Los objetos son **instancias** particulares de una clase
  - Todos los objetos de una clase **se comportan de forma similar**

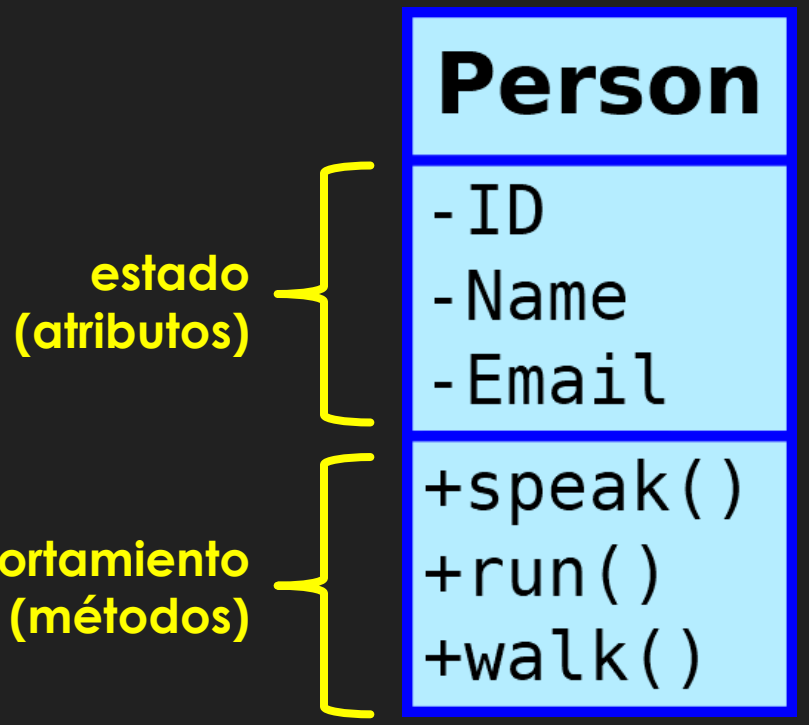




2. Breve repaso de conceptos de la POO

2.1 El paradigma de POO

- La **información** del objeto se denomina **estado**
- Los **métodos** (funciones) definen su **comportamiento**
- El **estado** de dos objetos de una clase es **diferente**



2. Breve repaso de conceptos de la POO

2.1 El paradigma de POO

- Clases y objetos

```
ID      =   1
Name    =   Elisa
Email   =   elisa@gmail.com
```

```
ID      =   2
Name    =   Pepe
Email   =   pepe@gmail.com
```

Person

```
-ID
-Name
-Email
```

```
+speak()
+run()
+walk()
```

2. Breve repaso de conceptos de la POO

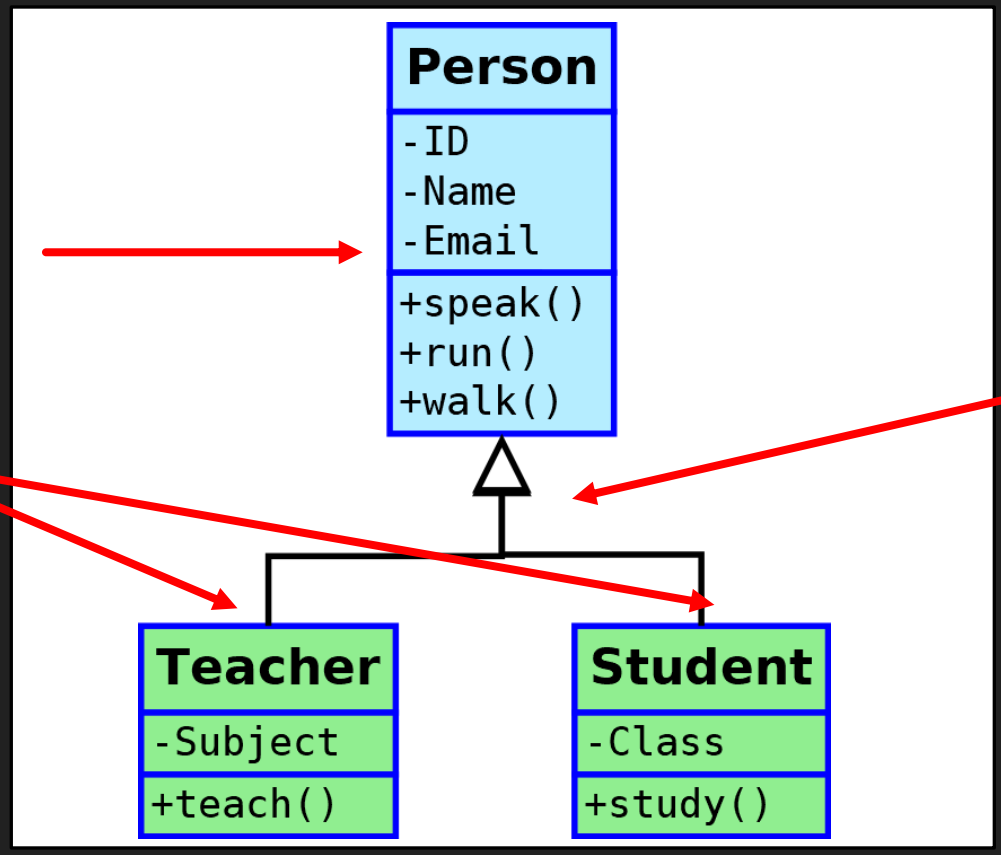
2.1 El paradigma de POO

- **Jerarquía de clases**
  - Las clases pueden organizarse según una **jerarquía**
  - La clase persona puede tener N **subclases**: Profesor, Jardinero, Policía…
  - Las subclases tienen ciertos **atributos comunes** (los de la clase base o padre)
  - Cada subclase puede tener, además, **atributos** o **métodos propios**

2. Breve repaso de conceptos de la POO

2.1 El paradigma de POO

- Jerarquía de clases



superclase

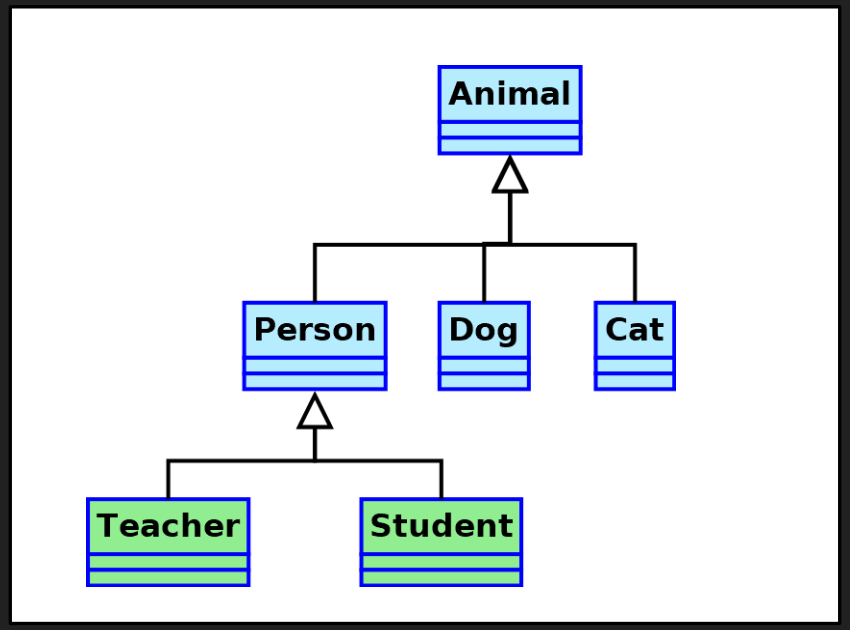
subclases

este símbolo representa una relación de herencia

2. Breve repaso de conceptos de la POO

2.1 El paradigma de POO

- **Jerarquía de clases**
  - La jerarquía de clases se puede complicar todo lo que sea preciso…



2. Breve repaso de conceptos de la POO

2.2 Pilares de la POO

- La POO se basa en cuatro **fundamentos**:
  - Abstracción
  - Encapsulación
  - Herencia
  - Polimorfismo

2. Breve repaso de conceptos de la POO

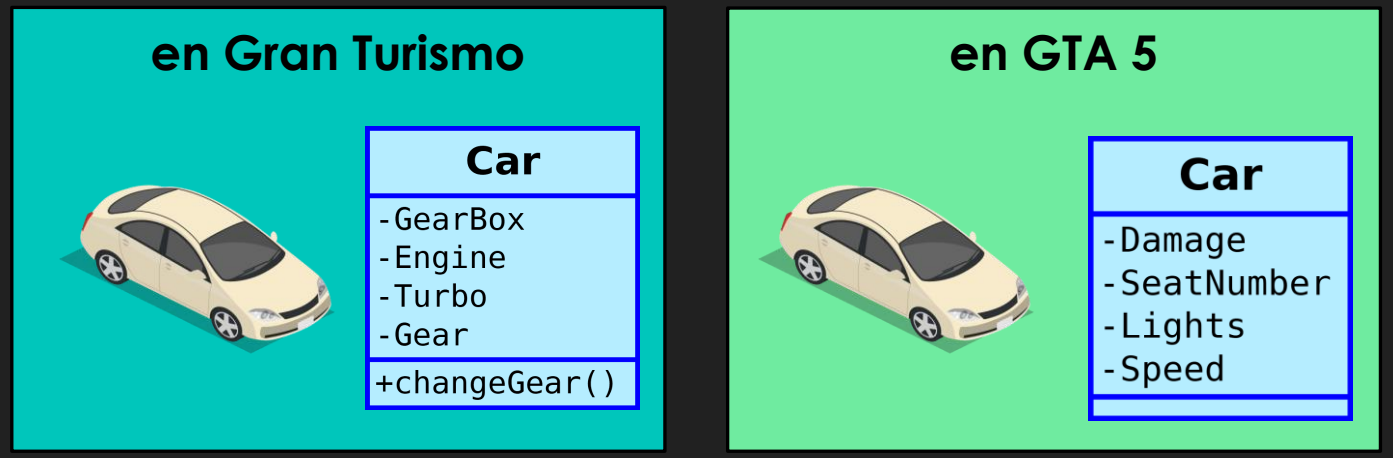
2.2 Pilares de la POO

- **Abstracción**
  - Consiste en **modelar** un objeto del mundo real
  - Se copian ciertas características del objeto
  - No obstante, no copiamos todas, sino que nos limitamos al contexto del programa
  - Dentro del contexto de nuestro programa, el objeto queda definido con gran precisión

2. Breve repaso de conceptos de la POO

2.2 Pilares de la POO

- **Abstracción**
  - En otro contexto, puede ser que el mismo tipo de objeto tenga otros atributos



2. Breve repaso de conceptos de la POO

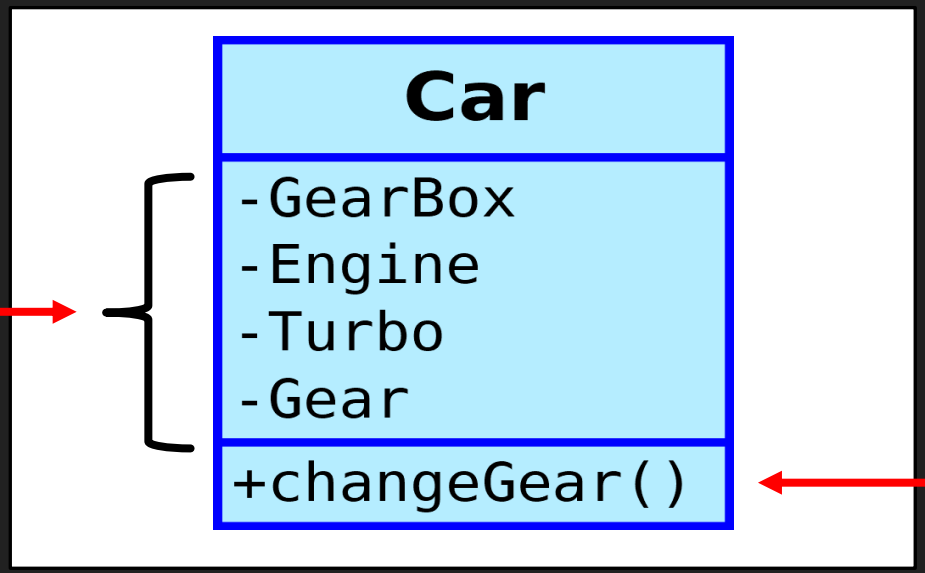
2.2 Pilares de la POO

- **Encapsulación**
  - Es la capacidad que tiene un objeto para **ocultar** parte de su estado o comportamiento
  - La parte que queda oculta no puede ser conocida por otros objetos
  - Encapsular significa hacer **privado**
  - **Ejemplo**:
    - No tienes porqué saber cómo funciona un motor para conducir
    - Tan sólo giras la llave y el coche arranca

2. Breve repaso de conceptos de la POO

2.2 Pilares de la POO

- **Encapsulación**
  - Para **cambiar** el estado del objeto se deben usar los métodos públicos



Oops!

¡El estado del objeto siempre debe ser privado!

2. Breve repaso de conceptos de la POO

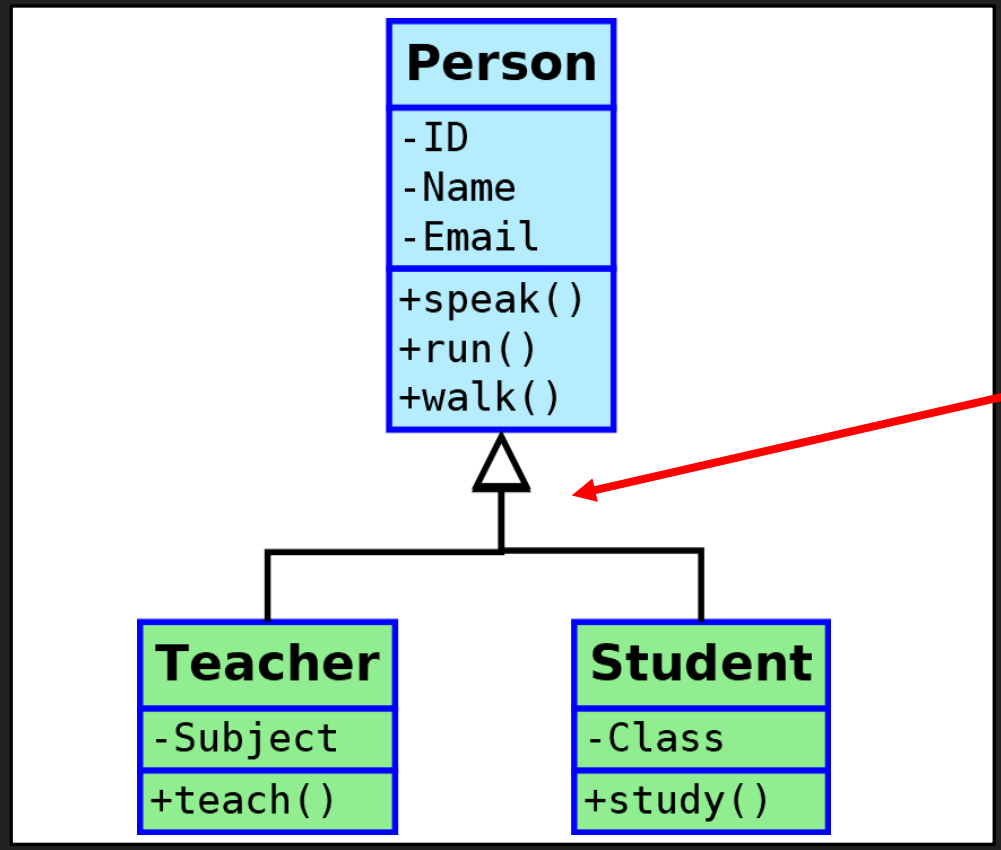
2.2 Pilares de la POO

- **Herencia**
  - Consiste en crear nuevas clases **extendiendo** clases existentes
  - Permite la **reutilización** de código
  - Las subclases pueden **modificar** su comportamiento frente a la clase padre
  - Hay que tener en cuenta que **las subclases tienen acceso a la interfaz de la clase padre**
  - Modificador **protected**: permite exponer atributos y métodos a las subclases

2. Breve repaso de conceptos de la POO

2.2 Pilares de la POO

- Herencia



este símbolo representa una relación de herencia

2. Breve repaso de conceptos de la POO

2.2 Pilares de la POO

- **Polimorfismo**
  - Es la capacidad que tiene un programa para **detectar** la clase correcta de un objeto
  - Una vez detectada la clase, se puede utilizar su implementación (datos + métodos)
  - **Ejemplo**:
    - Has creado una jerarquía de clases de personas (diapositiva anterior)
    - El programa, cuando recibe un objeto de tipo persona, no sabe si es Profesor o Estudiante
    - Una vez rastreado el objeto, el programa puede usar los métodos y atributos correctos

Contenidos de la sección

3. **Notación de diagramas de clases**
   - 3.1 Información previa
   - 3.2 Clases
   - 3.3 Atributos
   - 3.4 Notas
   - 3.5 Métodos
   - 3.6 Instancias
   - 3.7 Relaciones y visibilidad

3. Notación de diagramas de clases

3.1 Información previa

- Las clases y objetos se representan en **diagramas** usando UML
- UML es un lenguaje que permite **modelar**, **documentar** y **desarrollar** una aplicación OO
- UML fue creado por Grady Booch, Jim Rumbaugh e Ivar Jacobson (los padres de la OO)
- El **OMG** (Object Management Group) se encarga de mantener UML
- UML puede usarse en proyectos de cualquier grado de complejidad o envergadura
- **Consejo**: "cuando un código es complejo, crea más clases" (Grady Booch)

3. Notación de diagramas de clases

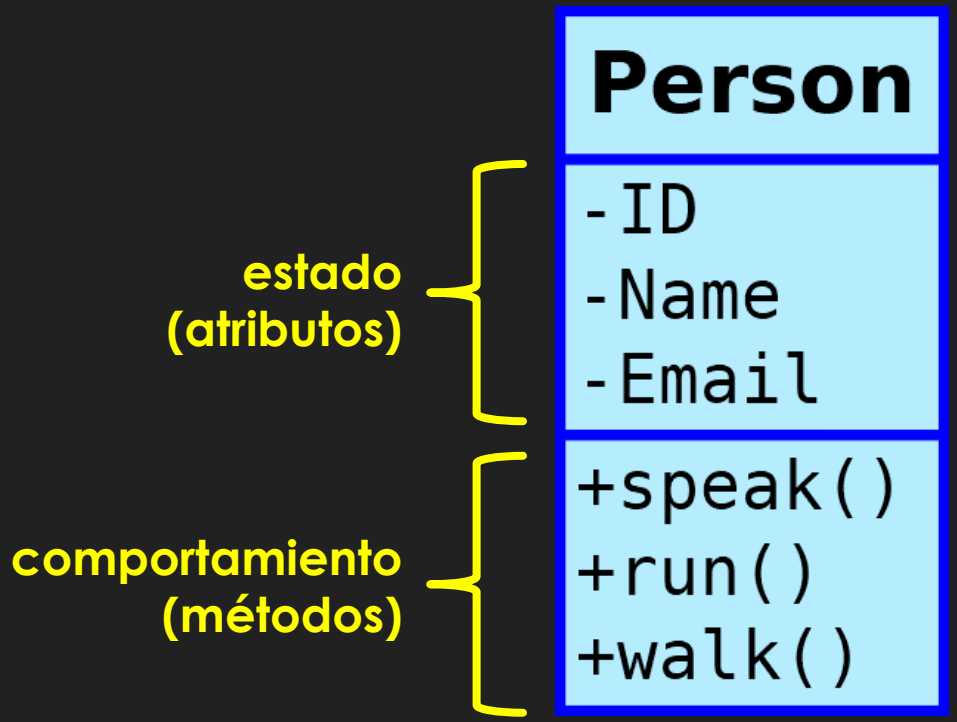
3.2 Clases

- Las clases son categorías o tipos de "cosas"
- Las clases se componen de un **nombre**, una serie de **atributos** y un conjunto de **métodos**
- A continuación analizaremos la simbología utilizada en UML para representar una clase
- Se deben nombrar con la notación **CamelCase** en mayúsculas
- En Java, el nombre de la clase debe ser el mismo que el nombre del fichero

3. Notación de diagramas de clases

3.2 Clases

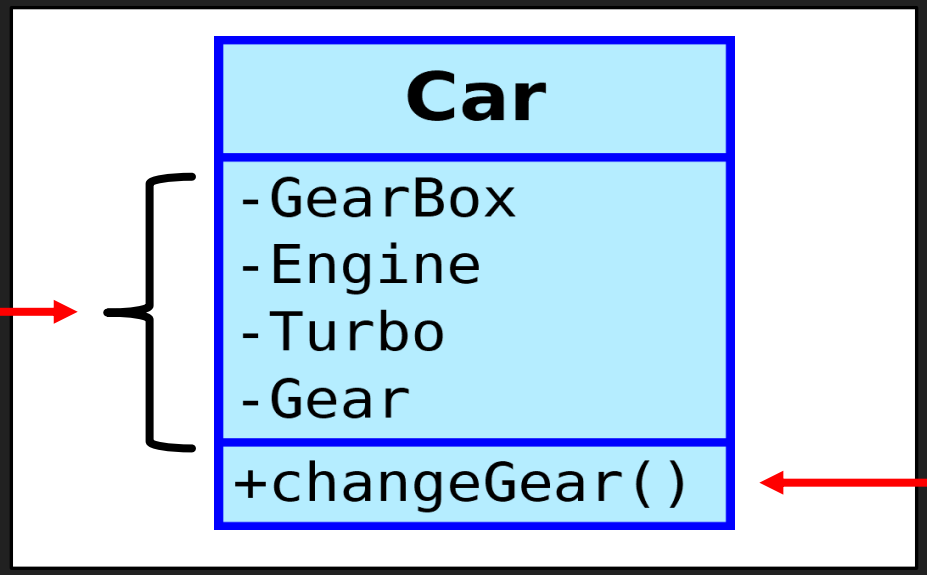
- La clase se compone de **atributos** (estado) y **comportamiento** (métodos)



3. Notación de diagramas de clases

3.2 Clases

- Los **atributos** generalmente son **privados**, y los **métodos** permiten modificarlos (son **públicos**)



3. Notación de diagramas de clases

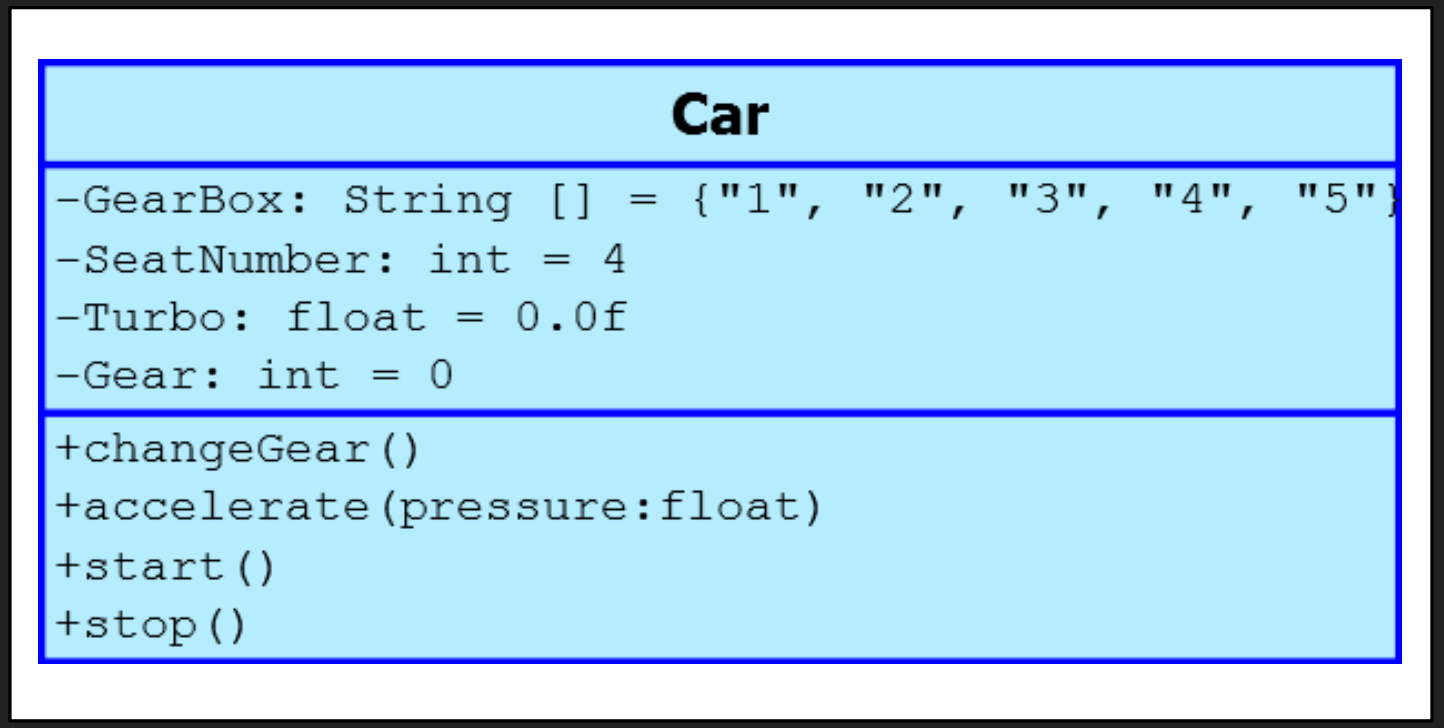
3.3 Atributos

- Los **atributos** de una clase pueden tener un tipo determinado
- En UML se puede especificar el tipo (igual que en un lenguaje de programación)
- Asimismo, los atributos pueden tener un valor por defecto
- Se deben nombrar con la notación **camelCase**

3. Notación de diagramas de clases

3.3 Atributos

- Esta clase posee **tipos de datos** y **valores por defecto**



3. Notación de diagramas de clases

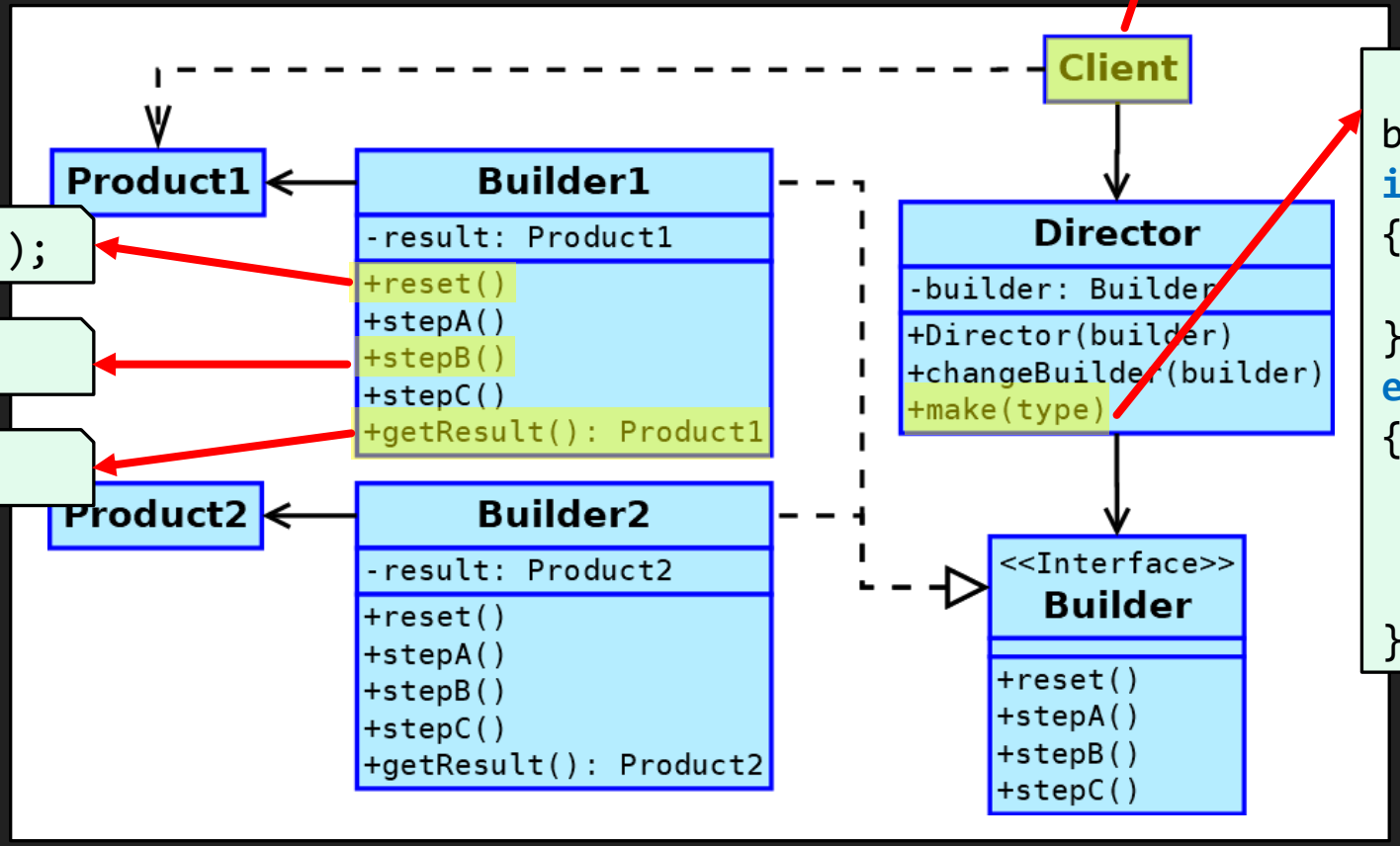
3.4 Notas adjuntas

- En ocasiones puede ser necesario incluir información para aclarar el esquema
- Estas aclaraciones figuran en una **nota adjunta** en el modelo

3. Notación de diagramas de clases

3.4 Notas adjuntas

```
Director d = new Director(new Builder1());
d.make();
Product1 p = d.getResult();
```




```
result = new Product1();
```

```
result.setFeatureB();
```

```
return this.result;
```

```
builder.reset();
if(type=="simple")
{
    builder.stepA();
}
else
{
    builder.stepA();
    builder.stepB();
    builder.stepC();
}
```

3. Notación de diagramas de clases

3.5 Métodos

- Los métodos de una clase constituyen la parte **dinámica** de la clase
- Se deben nombrar con la notación **camelCase**
- Los métodos pueden aceptar 0 o más parámetros
- Los métodos pueden devolver 1 valor o ninguno (**void**)

```
+accelerate(pressure:float)
```

3. Notación de diagramas de clases

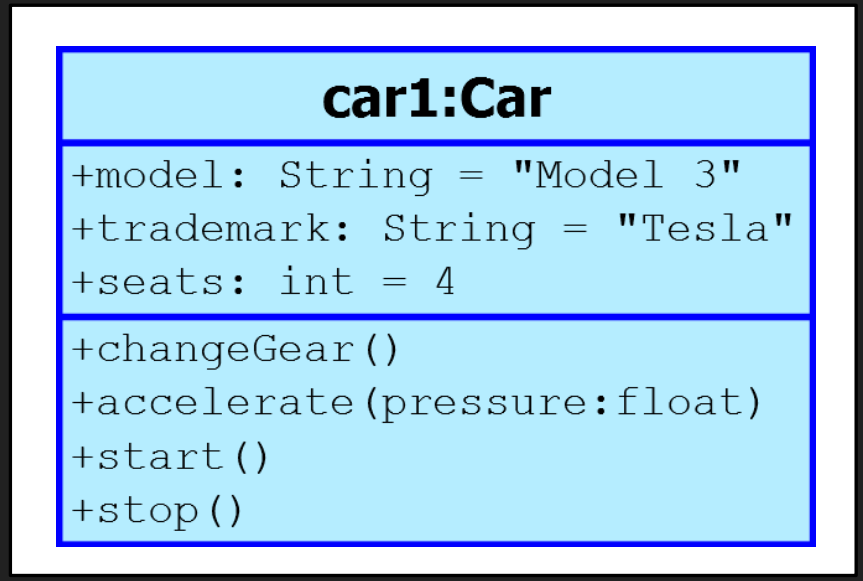
3.5 Métodos

- **Ejemplo práctico**: comprendiendo al cliente
  - **"El frigorífico muestra en el panel la temperatura del refrigerador, así como la del congelador"**
  - **Clases**: Frigorifico, Panel
  - **Atributos**: temperaturaRefrigerador, temperaturaCongelador
  - Los atributos en realidad pertenecen al frigorífico, no al panel
  - **Métodos**: mostrarTemperaturas(), es un método del panel para mostrar las temperaturas

3. Notación de diagramas de clases

3.6 Instancias

- Un objeto es una **instancia** de una clase
- La instancia tiene unos **valores específicos** en los atributos



3. Notación de diagramas de clases

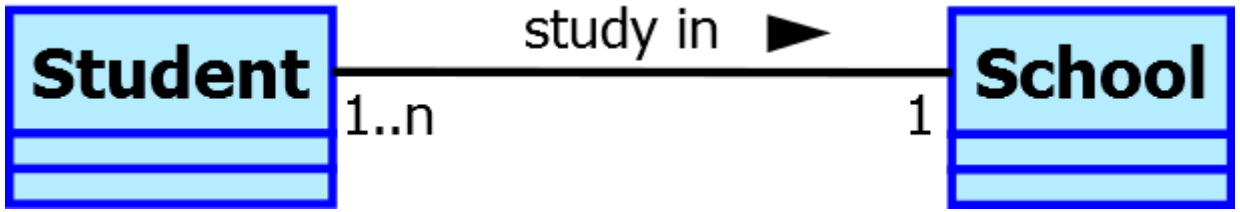
3.7 Relaciones y visibilidad

- Las clases se **conectan** unas con otras de forma conceptual
- Es posible escribir los **roles** y las **multiplicidades** en cada relación (como en el modelo E-R)
- Al igual que en el modelo E-R, pueden existir relaciones **exclusivas** o **reflexivas**
- **Multiplicidades**: 0..1, 1..*, 0..*, 12,24, 12..24
- Veamos algunos ejemplos y pasemos a analizar cada tipo de relación por separado

3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

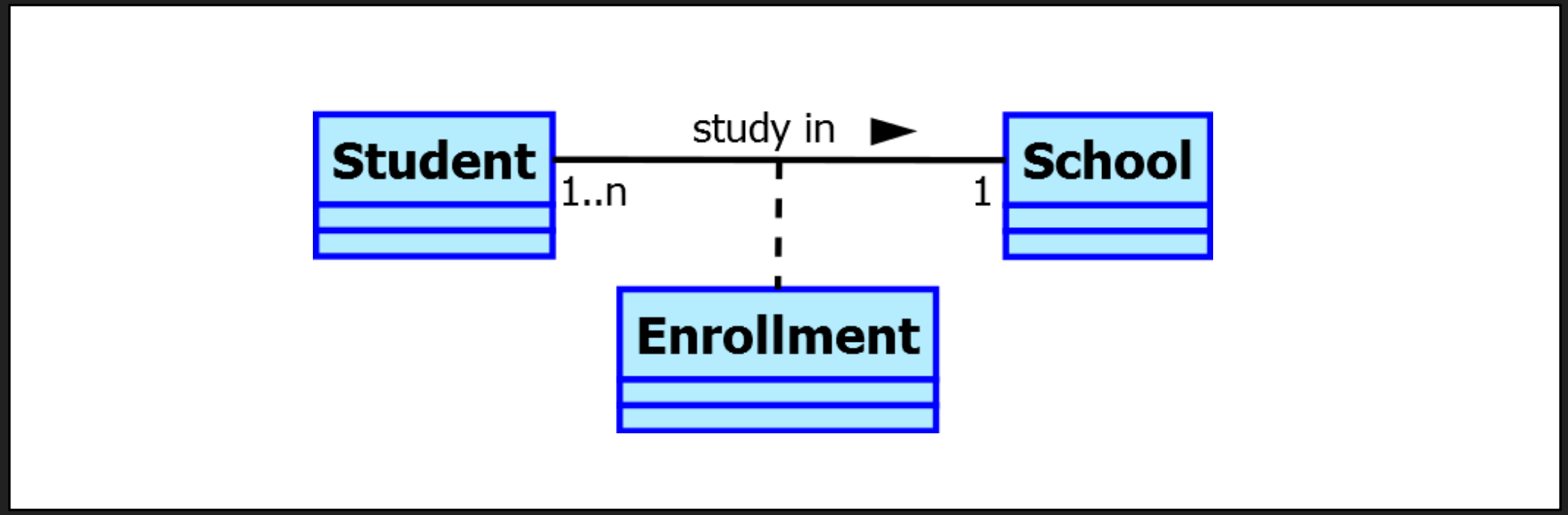
- **Ejemplos**



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

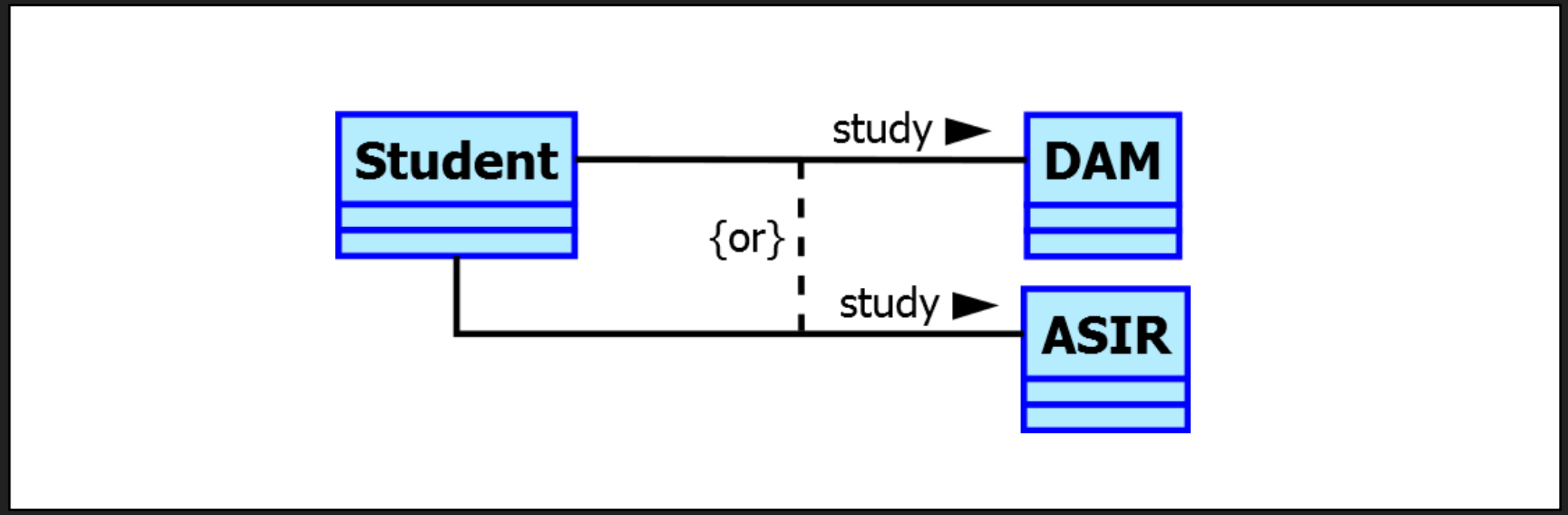
- **Ejemplos**



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

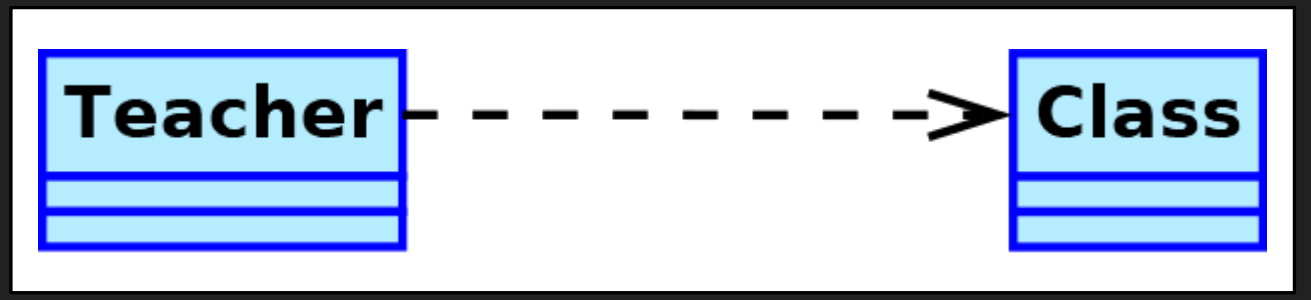
- Ejemplos



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

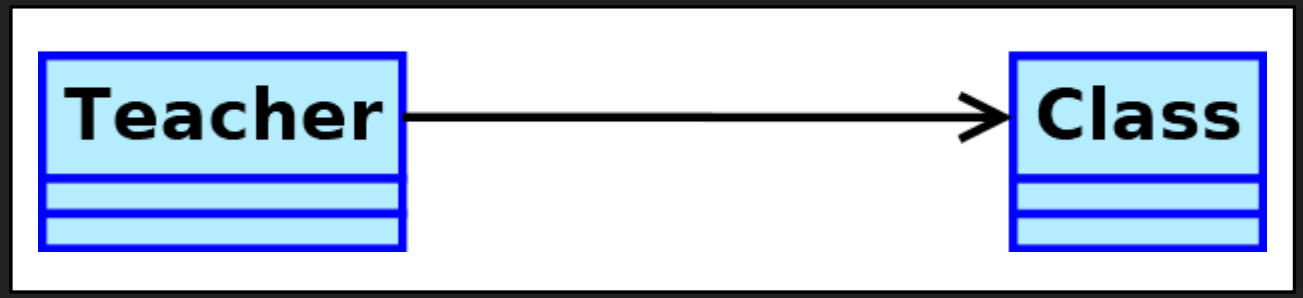
- **Dependencia**
  - Es la relación más básica que puede existir entre dos clases (y también la más débil)
  - Significa que una clase **depende** de la implementación de la otra



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

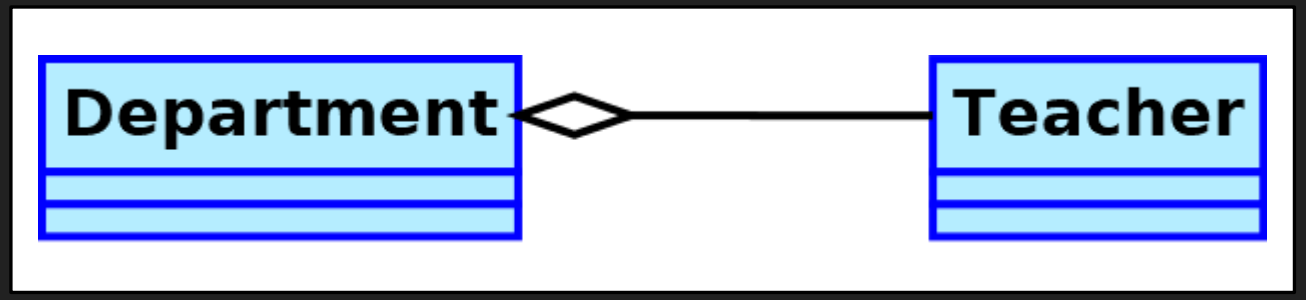
- **Asociación**
  - Significa que un objeto se **comunica** con otro (puede ser bidireccional)
  - Sería un tipo especializado de dependencia (el vínculo es permanente)
  - El objeto tiene acceso a los métodos de los objetos con los que interactúa



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

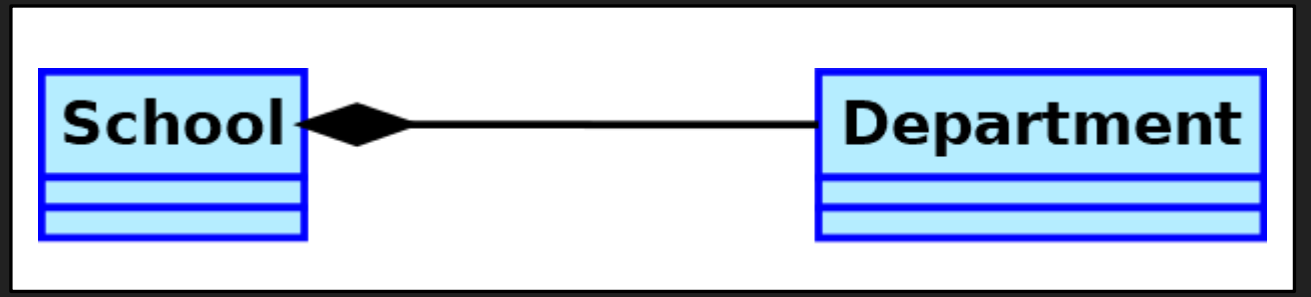
- **Agregación**
  - Es un tipo de asociación especial, que representa una relación **1:N** o **N:N**
  - Uno de los objetos mantiene una **serie** o **colección** de objetos y sirve como **contenedor**
  - El componente puede existir sin el contenedor y puede estar vinculado a más contenedores



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

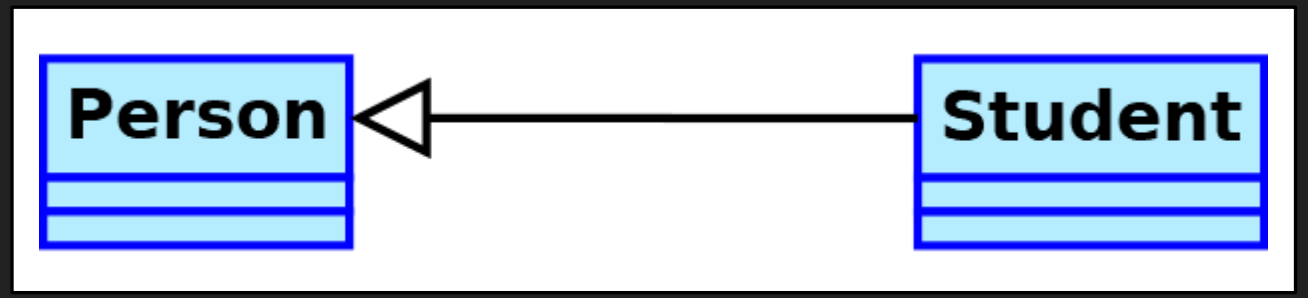
- **Composición**
  - Es un tipo de **agregación** que implica que un objeto **se compone** de otros
  - El componente sólo puede existir como parte del contenedor



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

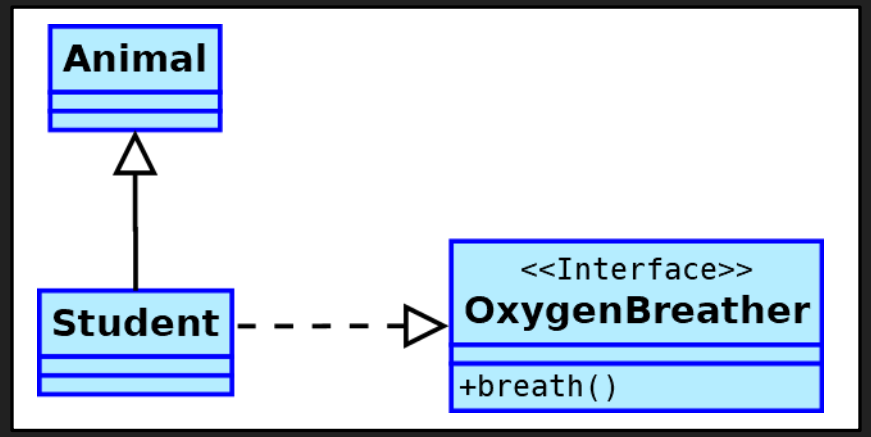
- **Herencia**
  - Permite **generalizar** una **clase base** en una o más **subclases**
  - Hay **Lenguajes de programación** que no soportan la **herencia múltiple** (ejemplo: Java)
  - La subclase **hereda** la **interfaz** y la **implementación** de la clase padre (y puede extenderla)
  - Un objeto de la subclase puede tratarse como un objeto de la clase padre



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

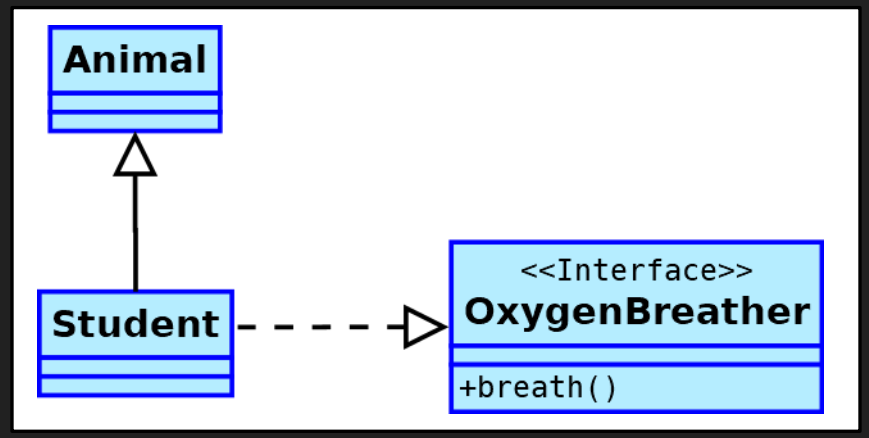
- **Implementación**
  - Consiste en **implementar** los métodos definidos en una **interfaz** (funciona como un contrato)
  - Las interfaces sólo se interesan por el **comportamiento** de los objetos (no tienen atributos)
  - Las interfaces posibilitan la **herencia múltiple** (ejemplo: Java)



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

- **Clases abstractas**
  - Nunca existirá una instancia de esta clase (son genéricas)
  - Sí pueden existir subclases de esta clase genérica (**Java**: modificador **abstract**)



3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

- **Visibilidad**
  - Existen **tres niveles** de visibilidad para atributos y métodos:
    1. **Nivel público (public, +)**
       - Las subclases pueden usar los atributos y métodos públicos de la clase base
       - Las subclases heredan los atributos y métodos públicos

3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

- **Visibilidad**
  - Existen **tres niveles** de visibilidad para atributos y métodos:
    2. **Nivel privado (private, -)**
       - Las subclases no pueden acceder a los atributos y métodos de la clase base

3. Notación de diagramas de clases

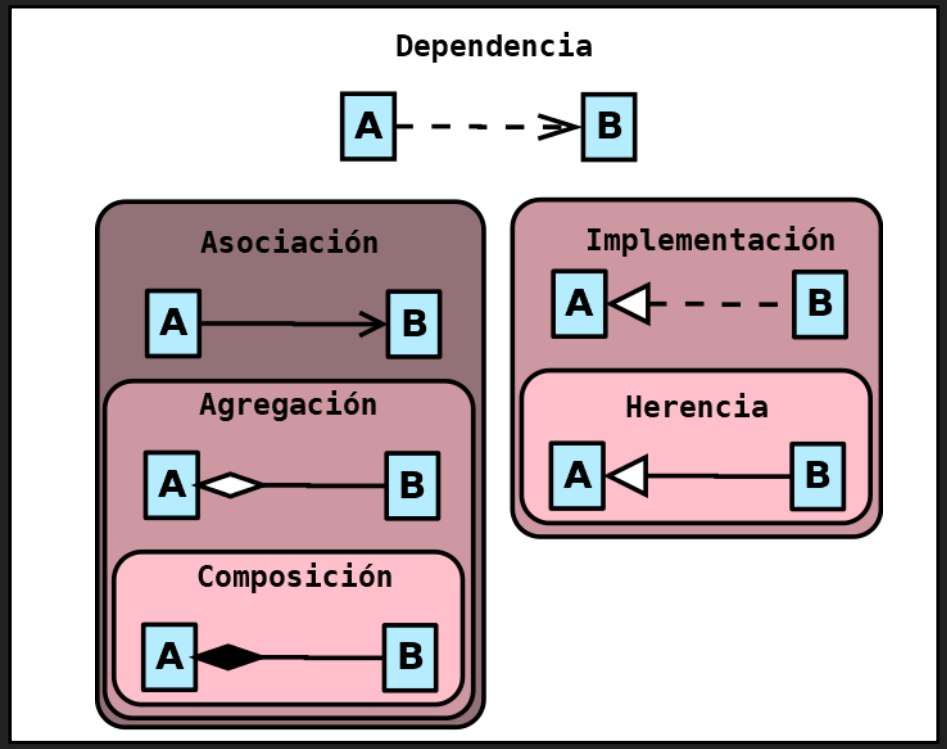
3.7 Relaciones y visibilidad

- **Visibilidad**
  - Existen **tres niveles** de visibilidad para atributos y métodos:
    3. **Nivel protegido (protected, #)**
       - Las subclases pueden usar los atributos y métodos protegidos de la clase base
       - Las subclases de éstas no los heredarán

3. Notación de diagramas de clases

3.7 Relaciones y visibilidad

- Resumen de relaciones (de la más débil a la más fuerte)



Contenidos de la sección

4. **Herramientas para generar diagramas de clases**
   - 4.1 Instalación de DIA Diagram Editor
   - 4.2 Uso de DIA Diagram Editor

4. Herramientas para generar diagramas de clases

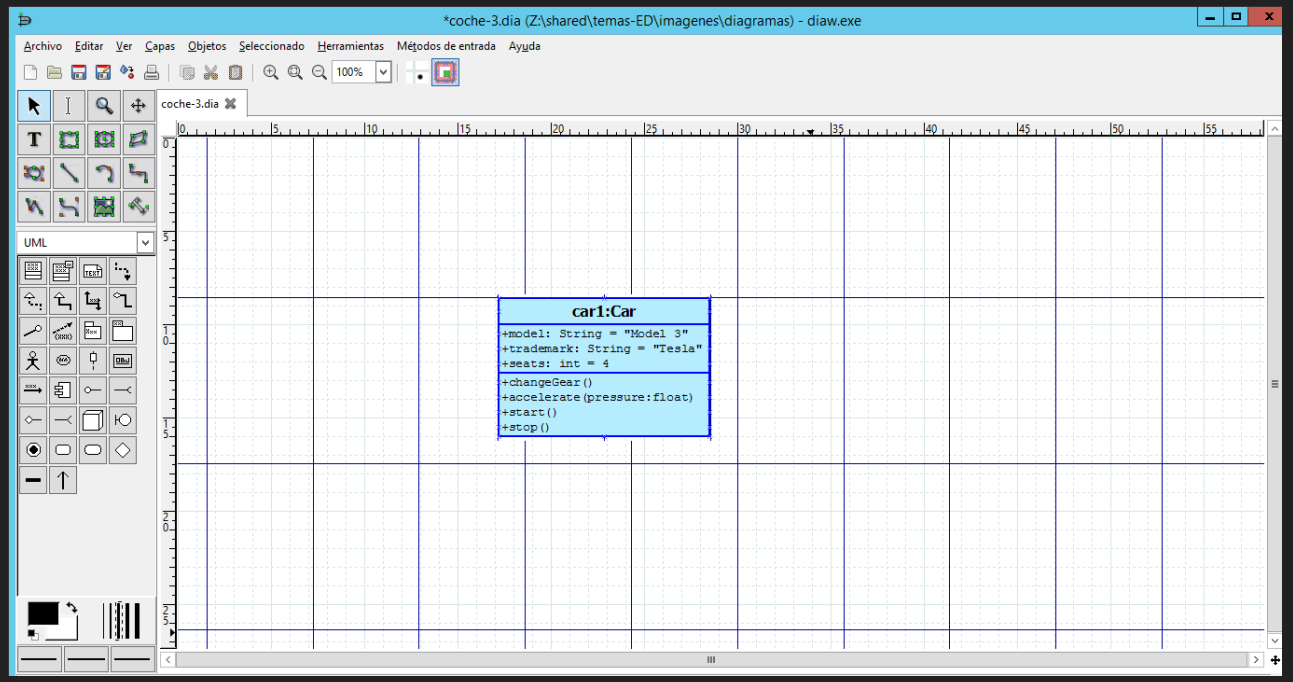
4.1 Instalación de DIA Diagram Editor

- Ahora realizaremos la instalación de **Dia Diagram Editor**
- Esta aplicación permite generar diagramas UML
- La aplicación es compatible con Windows, Linux y Mac
- Puedes obtenerlo en este **enlace**

4. Herramientas para generar diagramas de clases

4.2 Uso de DIA Diagram Editor

- A continuación, explicaremos, sobre el programa, cómo usar la paleta y dibujar en UML



Contenidos de la sección

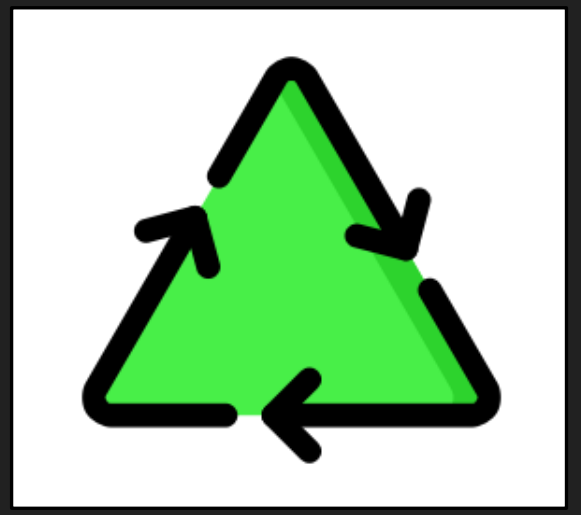
5. **Recomendaciones finales**
   - 5.1 Reutilización de código
   - 5.2 Extensibilidad
   - 5.3 ¡Encapsula!
   - 5.4 ¡Usa interfaces!
   - 5.5 Composición frente a herencia

5. Recomendaciones finales

5.1 Reutilización de código

- Los dos parámetros más relevantes del desarrollo son **costo** y **tiempo**
- Trata siempre de **reutilizar el código existente** en nuevos proyectos
- Los **patrones** mejoran la **flexibilidad** y **reutilización** de componentes
- **Tres niveles** de reutilización (Erich Gamma, ***Gang of 4***):
  - **Nivel bajo**: clase, biblioteca y contenedor
  - **Nivel medio**: patrón de diseño
  - **Nivel alto**: *framework* (JUnit, Spring…) **¡No nos llames, nosotros te llamamos a ti!** (*Hollywood*)

Reutilización



5. Recomendaciones finales

5.2 Extensibilidad

- El desarrollo de software está sujeto a **continuos cambios**
- Durante el desarrollo:
  - Se comprende mejor el problema cuando nos ponemos manos a la obra
  - Los requisitos de nuestros clientes son cambiantes (el mercado cambia)
  - Las tecnologías **evolucionan** (y no podemos hacer nada para evitarlo)

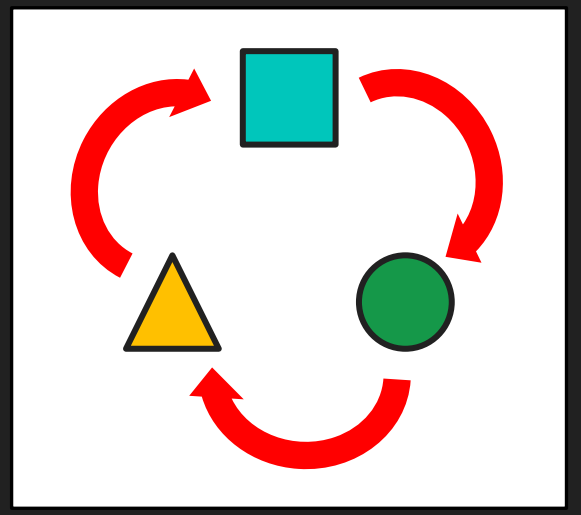
~~**funcionalidad alucinante**~~

nueva funcionalidad más alucinante

5. Recomendaciones finales

5.3 ¡Encapsula!

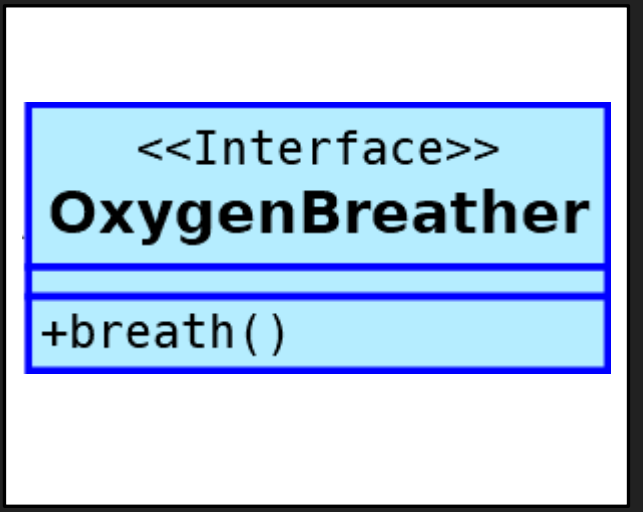
- Hay que **encapsular** el código que cambie, para minimizar cambios
- Separación de los aspectos que cambian de los que no
- No olvides descomponer la aplicación en **módulos**
- Diferentes tipos de encapsulación:
  - **A nivel de método**: aísla el código que cambie y encapsúlalo en métodos
  - **A nivel de clase**: en lugar de añadir nuevas responsabilidades a una clase, crea nuevas clases



5. Recomendaciones finales

5.4 ¡Usa interfaces!

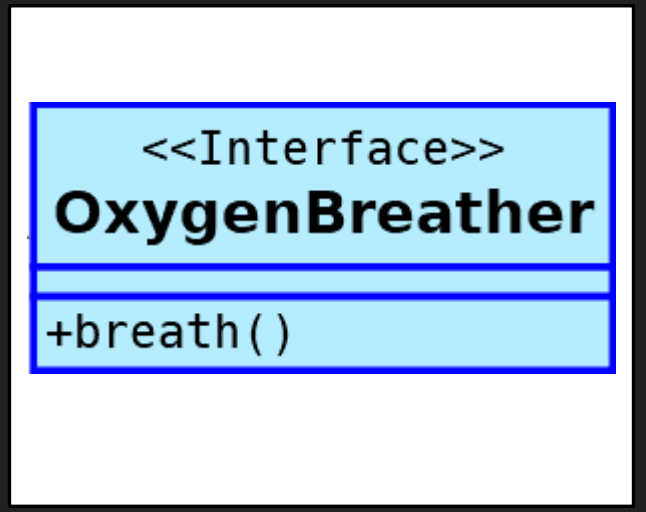
- Se debe programar a una **interfaz**, y no a una **implementación**
- Con esto evitamos depender de clases concretas
- La colaboración entre clases se establece con una **dependencia**
- Existe una forma más eficaz: las **interfaces**



5. Recomendaciones finales

5.4 ¡Usa interfaces!

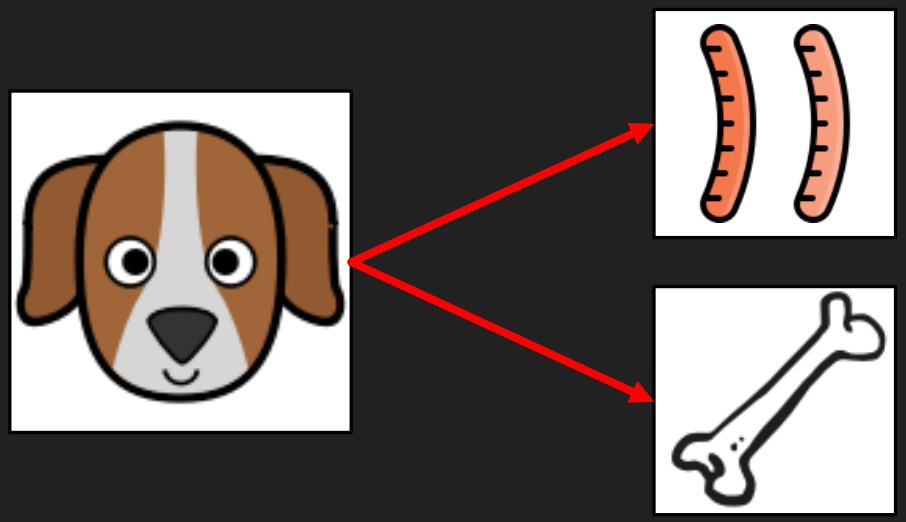
- Pasos para establecer colaboración con interfaces:
  1. Establece lo que un objeto **necesita** del otro
  2. Describe esos métodos en una **interfaz** o clase abstracta
  3. Las clases dependientes deberán **implementar** dicha interfaz
  4. Después, la clase dependiente dependerá de la interfaz



5. Recomendaciones finales

5.4 ¡Usa interfaces!

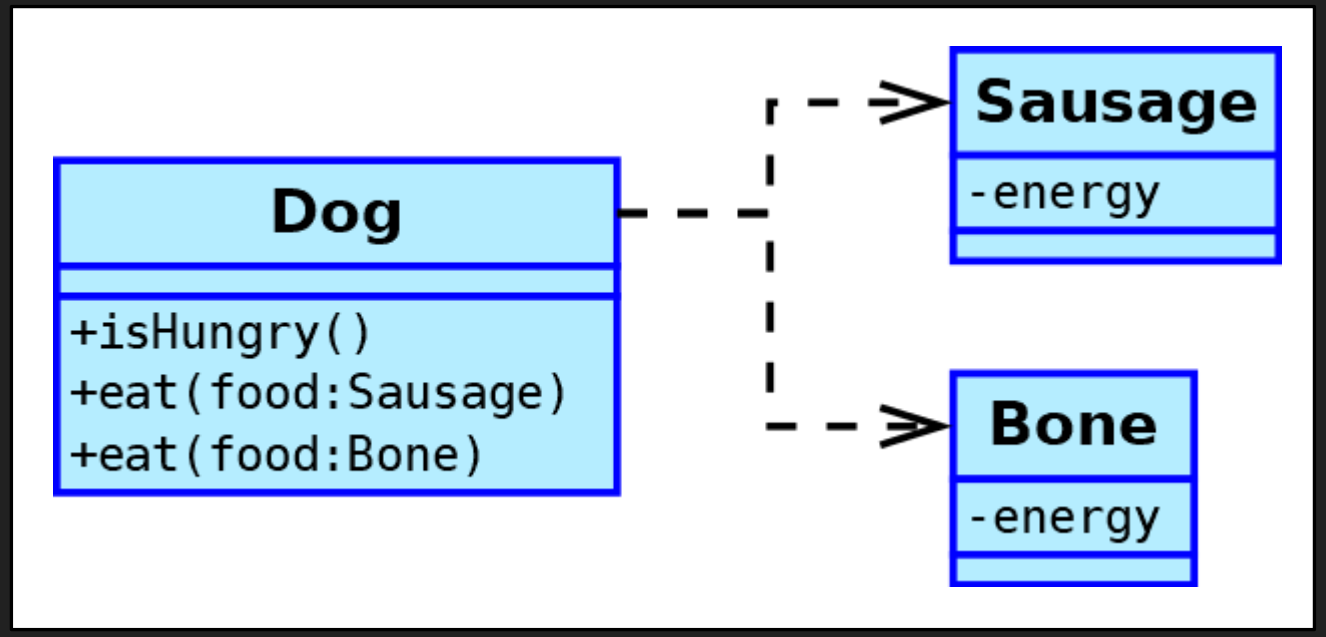
- **Ejemplo**: dar de comer a un perrito
  - A un perrito podemos darle de comer salchichas o un hueso, por ejemplo



5. Recomendaciones finales

5.4 ¡Usa interfaces!

- **Ejemplo**: dar de comer a un perrito (en UML, con dependencias)

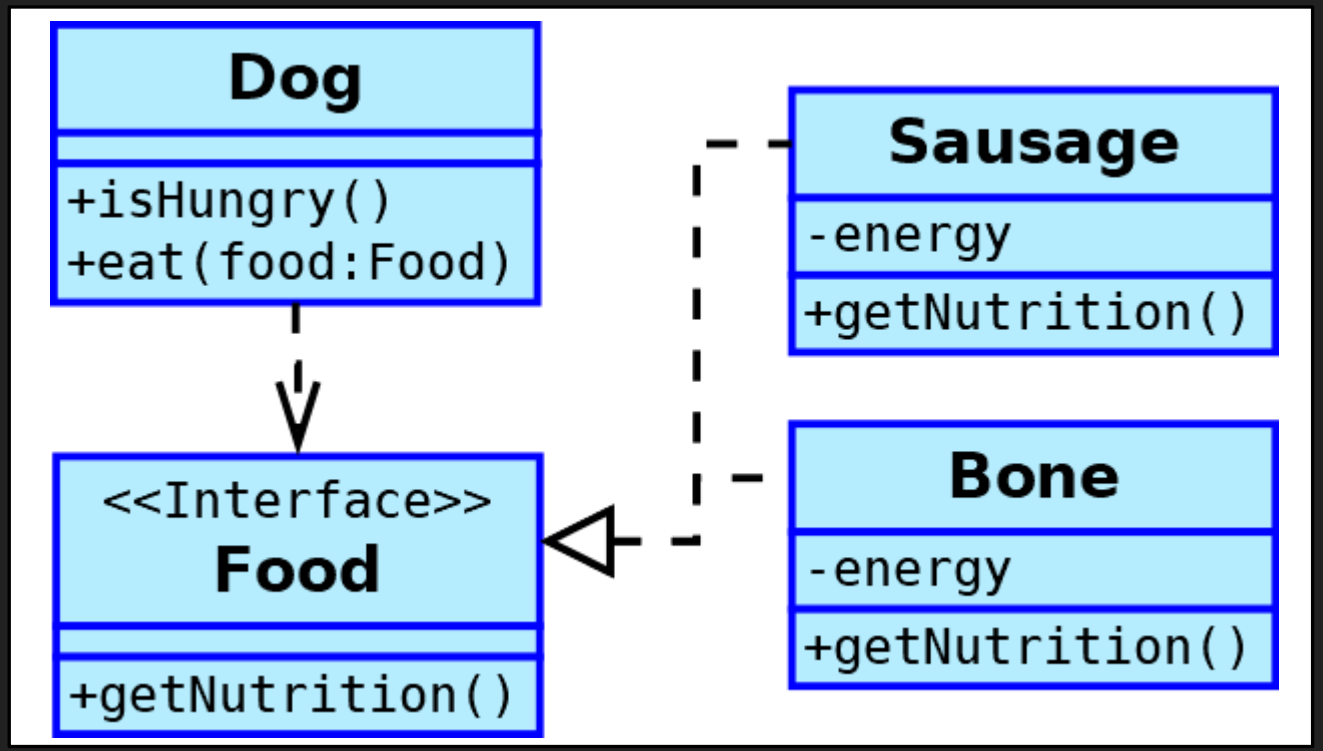


Este modelo es muy inflexible, dependiente y presenta un alto acoplamiento

5. Recomendaciones finales

5.4 ¡Usa interfaces!

- **Ejemplo**: dar de comer a un perrito (en UML, usando la interfaz Food)



La dependencia se establece mediante la interfaz Food, y cualquier implementación de ésta es una posible comida para nuestro perrito

5. Recomendaciones finales

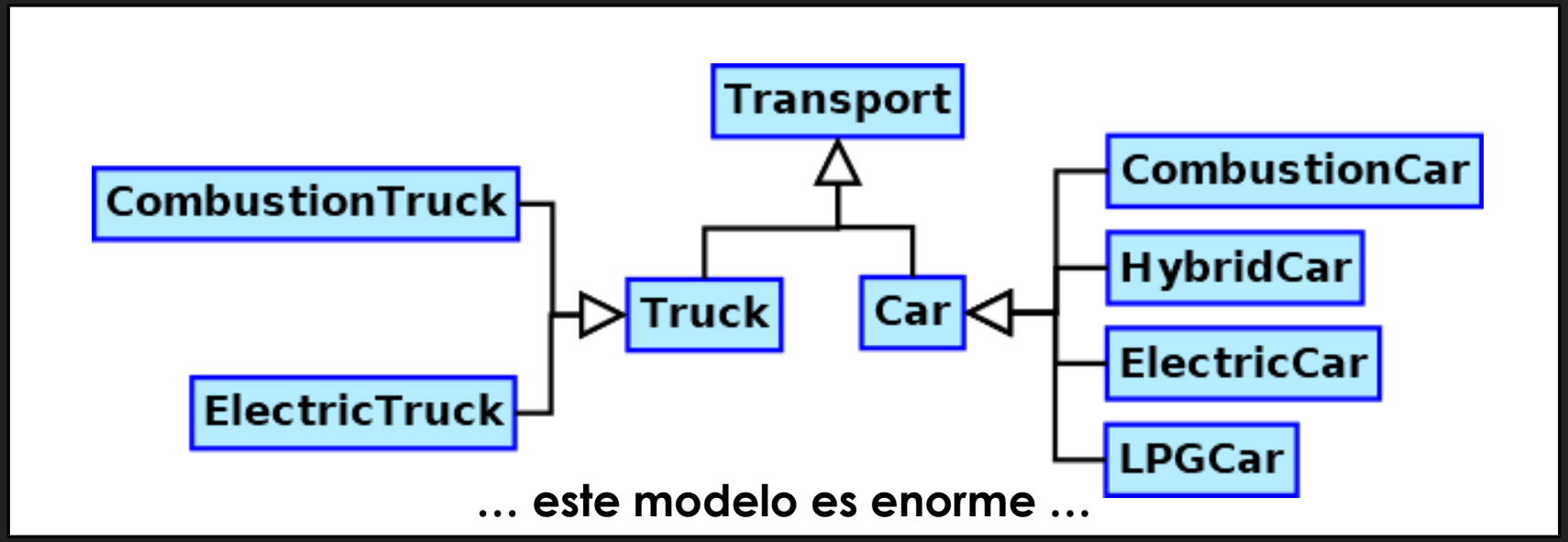
5.5 Composición frente a herencia

- Aunque la herencia favorece la reutilización de código, tiene algunos inconvenientes:
  1. Las subclases **no pueden reducir la interfaz** de la superclase
  2. Al sobrescribir un método, hay que **asegurar que su comportamiento es compatible** con la superclase
  3. La herencia **rompe la encapsulación** de la superclase
  4. Las subclases presentan un **alto acoplamiento** respecto a su superclase
  5. Si hay varias dimensiones de herencia, pueden surgir **toneladas de clases**
- La herencia es una relación **es un** (is a), mientras que la composición es **tiene** (has)

5. Recomendaciones finales

5.5 Composición frente a herencia

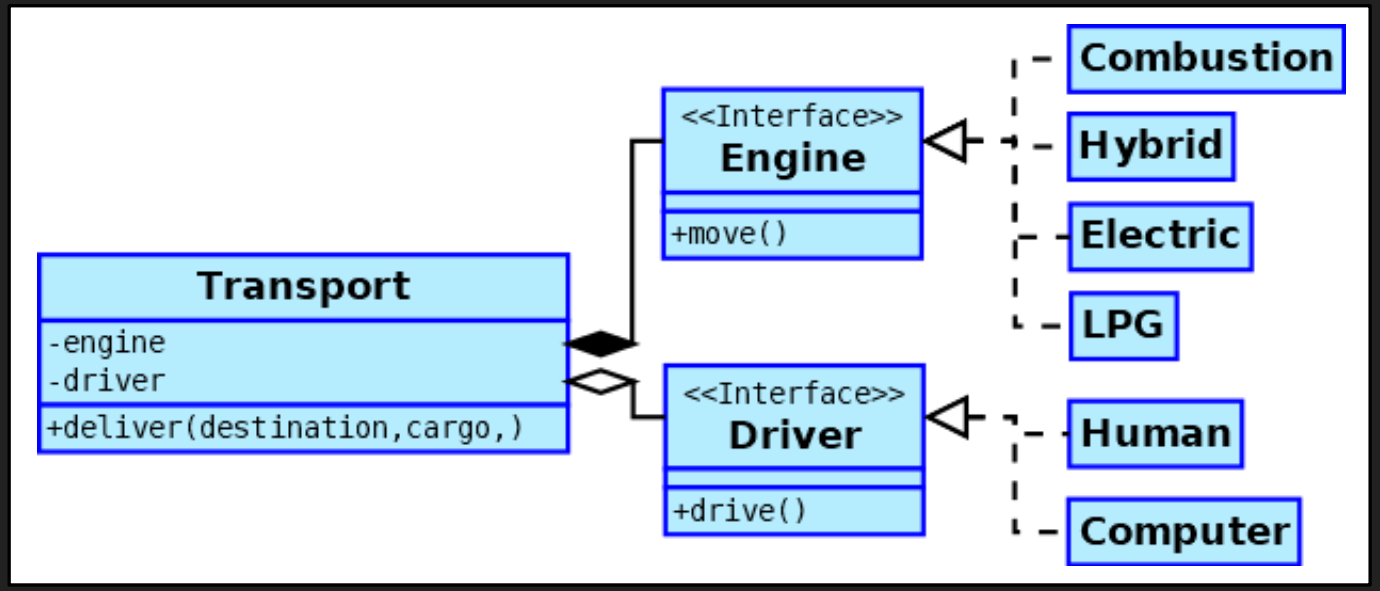
- **Ejemplo**: tipos de vehículos, usando **herencia** (es un)
- Hay demasiadas clases. Imagina representar el piloto automático (explosión combinatoria)



5. Recomendaciones finales

5.5 Composición frente a herencia

- **Ejemplo**: tipos de vehículos, usando **composición** (tiene)
- En este modelo, los objetos **delegan su comportamiento** en otros objetos



El comportamiento se puede sustituir en tiempo de ejecución

Contenidos de la sección

6. **¿Y ahora qué?**
   - 6.1 Ejemplos y ejercicios

6. ¿Y ahora qué?

6.1 Ejemplos y ejercicios

- Ahora realizaremos ejercicios y prácticas con **UML**
- En el siguiente **sprint**, aprovecharemos para crear el **diagrama de clases** de nuestro proyecto

Información complementaria

- **Descargar Dia Diagram Editor**: enlace
- **Tutorial Diagrama de Clases con UML**: enlace
- **Ejemplo práctico Diagrama de Clases con UML**: enlace
- **How to create UML diagram with Dia Diagram Editor**: enlace

Créditos de las imágenes y figuras

Cliparts e iconos

- **Obtenidos mediante la herramienta web IconFinder** (según sus disposiciones):
  - Diapositivas 1, 12, 14, 20, 57, 62-64, 67
  - Según la plataforma IconFinder, dicho material puede usarse libremente (free comercial use)
  - A fecha de edición de este material, todos los cliparts son free for comercial use (sin restricciones)

Resto de diagramas, gráficas e imágenes

- Se han desarrollado en **PowerPoint** y se han incrustado en esta presentación
- Todos estos materiales se han desarrollado por el autor
- Para el resto de recursos se han especificado sus fabricantes, propietarios o enlaces
- Si no se especifica copyright, el recurso es de desarrollo propio