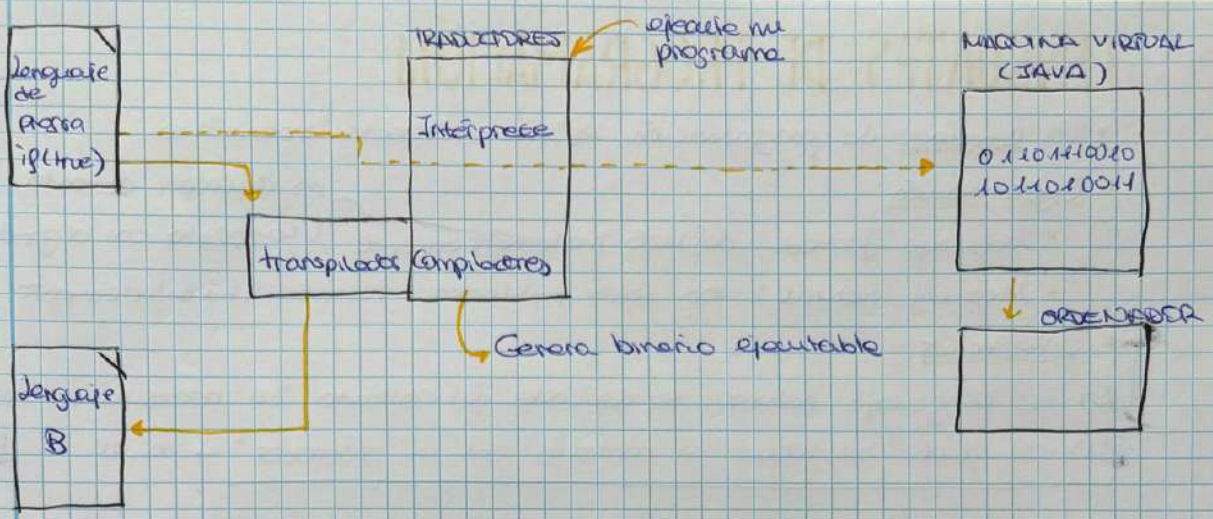
LENGUAJES DE PROGRAMACIÓN

- Un lenguaje de programación es una estructura matemática.
- Diferenciación:
  - Lenguaje formal: definido matemáticamente (completo, con algoritmos, ...) ← no dependen contexto
  - Lenguaje natural: son como hablamos nosotros habitualmente
- Diferencias:

En el lenguaje natural, se depende del contexto. Se necesita un nexo común.
"No se puede establecer un contexto con un ordenador" → se usa l. formal
Se están trabajando en ello "Ventana de contexto".
En programación se usa lenguajes de programación sin contexto.

- Tipos de lenguajes de programación:
  - Tipo máquina (bajo): más bajo nivel. Empieza donde hay 0 y hay un huecos con el 1 y con ello se hace el lenguaje binario. Actualmente no es usado porque hay millones de bits. ← perforadas
    - Lenguaje ensamblador: es múltiple y se diferencian de bajo nivel ya que dependen de la máquina en donde se va a ejecutar. (Necesita tocar hardware ya que usa operaciones básicas).
    - Lenguaje de alto nivel // Lenguaje de tercera generación: (gramática formal: se definen órdenes en un lenguaje sencillo humano para hacer operaciones sencillas). Primeros lenguajes con el concepto variable. Te da flexibilidad para abordar ideas (y velocidad). No son tan eficientes como el ensamblador.
    - Lenguajes orientados a problemas / de cuarta generación: es un lenguaje para resolver algún problema concreto, se aproximan mucho a un comando. El SQL es un ejemplo que no sirve para programar sino para hacer consulta de datos de una base de datos.
- ¿Necesitamos más lenguajes de programación? No pero sí.

Desde el punto de vista formal / matemático, no es necesario
Necesita adaptarse a un entorno de ejecución que puede ser más general o específico. (Java es general)
En la informática todo va rápido y las tecnologías se implantan y se desechan muy rápido. Continuamente salen nuevos nuevos lenguajes (ELM, Go, ...)
* Lo importante no es el lenguaje sino las estrategias para solucionarlo *



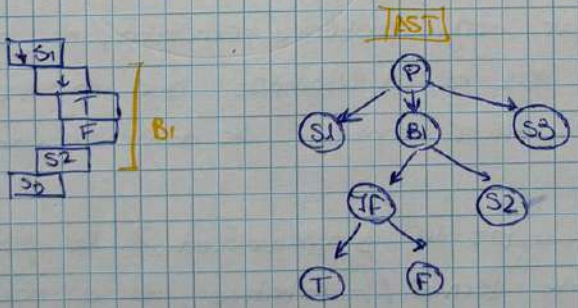
- ¿Quién hace el primer compilador de un lenguaje de programación nuevo?

El primero propiamente dicho se hizo en binario, a partir de ahí lo extienden (muchas versiones) → requiere mucho tiempo.

- Hay un nuevo tipo de traductor que se llama transpilador. Coge 1 lenguaje lo transpila a otro lenguaje (traduce de un lenguaje a otro, adaptar un programa a otro).

Java tiene una máquina virtual para la cual se crea el binario y el ordenador esta abajo.

- Hay lenguajes donde es obligatorio usar el mismo estilo porque sino no compila.
- Objeto: representación intermedia (abstracta) entre el programa (problema)
- AST: (árbol de sintaxis abstracta). Cómo se comprime compilo y se transforma en un lenguaje abstracto es lenguaje objeto.



- Entornos de ejecución: donde se van a ejecutar el programa. Podemos ejecutar en máquinas físicas o virtuales.

¿cuál es mejor?

Sobre el rendimiento, lo binario el metal (Java)

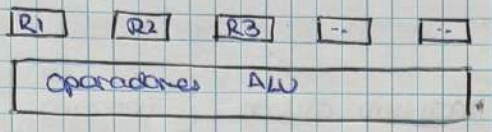
¿Flexibilidad?

Es mejor la máquina virtual porque todo está a la vista y

podeis jugar con ello.

Según el tipo de ejecución: máquinas tipo registros o de pilas.

- máquina registros de Von-Neumann -

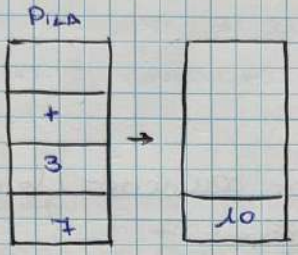


Punto de vista ingeniería.

R1 + R3 → R4
7 3 10

- autómata de pila -

Punto de vista electrónica.

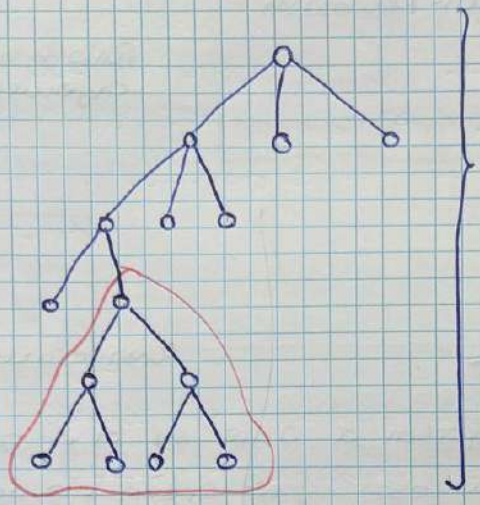


Importan el orden en el que mete las cosas y el orden lo da el arbol (AST)

Ordenadores de pocos recursos y poca memoria.

PARADIGMA DE PROGRAMACIÓN.

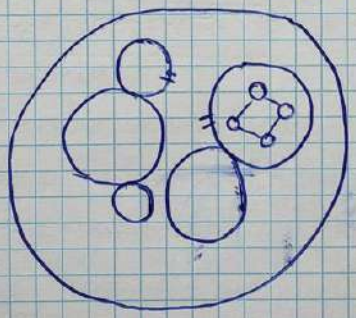
Problemas:



Mecanismo divide y vencerás.
(nos ayuda a dividir los problemas pero no los abstrae).

Lo que hacemos es cortar ramas y nos olvidamos (de manera intuitiva).
Hay otras maneras de abordar los programas.

- Escuelas de pensamiento:
  - Programación funcional: paradigma "VIVA LAS MATES" (todo función, es un punto de vista matemático) se anidan. Alteran como piensas y actúas.
  - Programación orientada a objetos: intuitivos, cómo pensamos los problemas los humanos. Dentro de "contextos" (segmentamos)



Interface - GUI (interface gráfica usual)

Define cómo se usa.
(frontera de un sistema con otro)

Si el problema es grande, necesitamos abstraer partes y cambiar el foco de atención de una cosa a otra.

Objetos: separamos los programas por partes sin necesidad de saber todos los detalles solo se necesita establecer un contrato / interface. "gráfica"

ABSTRACCIÓN:

La abstracción permite olvidarte de los detalles usando librerías, componentes, objetos, clases de otras personas. Por tanto es una manera de programar que usa el método de reutilización.
A veces se necesita conocer detalles pero si es muy fácil usar otro programa no se hace porque reutilizar. (80/20)

(

ENCAPSULAMIENTO

Lo que se busca es que no se vean los detalles de implementación. (Problemas para quien lo crea no es visto)
Genero la funcionalidad en partes reducidas y controladas sin exponerlas para que sea más fácil el mantenimiento si tienes que hacer una modificación.
Tiene relación con el principio de ocultación. No solo encapsule la funcionalidad sino que lo mete en un módulo que hace que no se vea.
Hay un tipo de programación orientada a objetos que se llama programación orientada a componentes. Se ve sobre todo en programas de sistemas en 3 dimensiones. (ej: videojuegos)

MODULARIDAD.

Cuando trabajamos con programación orientada a objetos programamos clases (objetos). Los objetos para funcionar deben colaborar entre sí.
El principio de modularidad dice que dependiendo de lo que necesite en un momento determinado, trabajaré con un conjunto clase/objeto como módulo.
Las distintas clases de objetos se dividen en pequeños subproblemas que actúan de forma coordinada y depende de donde esté el usuario.
Existen una serie de herramientas como la herencia y el polimorfismo.
Son herramientas de diseño en donde puedes utilizar 1, otra, ambas, una combinación para conseguir diseñar los problemas.
La herencia es un tipo de operación de reutilización de código
En el polimorfismo dadas 2 clases llamadas igual y aparentemente tienen la misma funcionalidad pero en verdad es diferente (pila / cola)

RECOLECCIÓN DE BASURA.

Cuando se programa lo que tienes por debajo es un ordenador y de igual que estructura o problema de programación usa.
P. estructurada: debes saber todo lo que hay y por tanto controlar que creas y que no crees
P. objeto: te olvidas del problema y del ordenador que hay debajo (no piensas en la memoria / disco / etc)
Implica que cuando creas y destruyes los objetos debe de crearse en algún lado.
Las funcionalidades de recolección de basura se encargan

~~programación~~ orientación a objetos

Es una función que se ejecute cada cierto milisegundos que hace una pasada por nuestro programa para ver que objetos han dejado de interesarse y eso mata y en la siguiente pasada del recolector, recoge la basura y libera la memoria (automático) (No es obligatorio que lo tenga ya que se puede hacer manualmente)

CLASE

CLASE

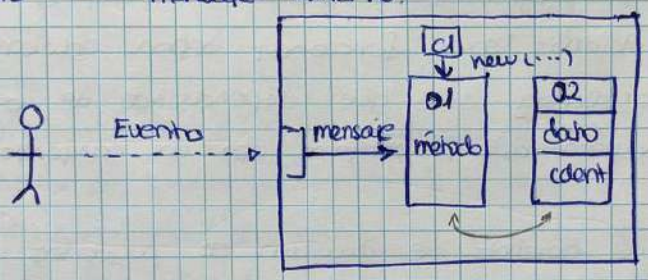
OBJETOS → implementación

MÉTODOS → funcionalidad de la clase

MENSAJES → invocación de un método

EVENTO → mensaje externo.

* Diferencia entre 2 objetos:
  - identificador / referencia
  - estado interno (los valores de los atributos)



ATRIBUTO / PROPIEDAD → dato, variable de clase.

DIAGRAMAS (cómo pinto un programa?)

Diagrama casos de uso: se estudia el establecimiento de requisitos.

Diagrama de clases: diagrama de la solución propuesta (entre Análisis y Diseño)

Diagrama secuencial: secuencia lógica de ejecución. Si yo ejecuto que va a pasar?

Diagrama de estados: objeto con estado interno especial (no dice el estado).

Diagrama de componentes: nos da los módulos a alto nivel.

Fases canónicas del desarrollo de software

- Análisis
- Diseño
- Programación
- Pruebas

(A veces está antes el establecimiento de requisitos.) y generar doc. con las funcionalidades a programar (como contrato).

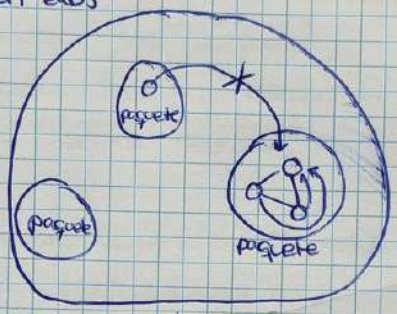
(cuestión de esferas)

ESTRUCTURA / ORGANIZACIÓN

PAQUETES

La POO nos permite trabajar con distintos niveles de modularidad. Los atributos y los métodos de una clase pueden ser públicos o privados (encapsulación). También hay un nivel medio métodos y atributos protegidos

Lo público se ve desde el exterior y lo privado desde el interior.
Lo protegido es ~~entre~~ dual (en parte público y privado)
Para saber si es público o privado nos basamos en qué clases quieren acceder con ellos



Los paquetes se establecen:

paclase es.ucm.matcomp.mp.ej1.eja.e1.

(se parece mucho a una estructura de directorios, carpetas)

Ejemplo: paclase es.ucm.matcomp.mp.ej1.eja.e1

```
publico class MiClase {
sólo clases → protected int dato;
del paquete
pueden acceder
}
```

```
publico class MiSegundaClase {
}
```

ANOTACIONES

Las anotaciones es una manera de dirigir como queremos que se comporte el lenguaje.

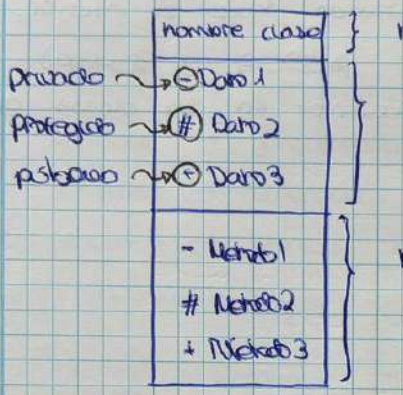
Ej: @Override
public String toString() { ← toString ya existen y lo reescribimos al nuevo que has creado.
}

Las anotaciones no las declaramos nosotros aunque se puede, no es un proceso habitual (es raro) y requiere programación. No están pensadas para que todos los programadores declaren anotaciones.
Las anotaciones nos sirven para decirle a Java cosas con las que tiene que tener cuidado porque hay que operar de una

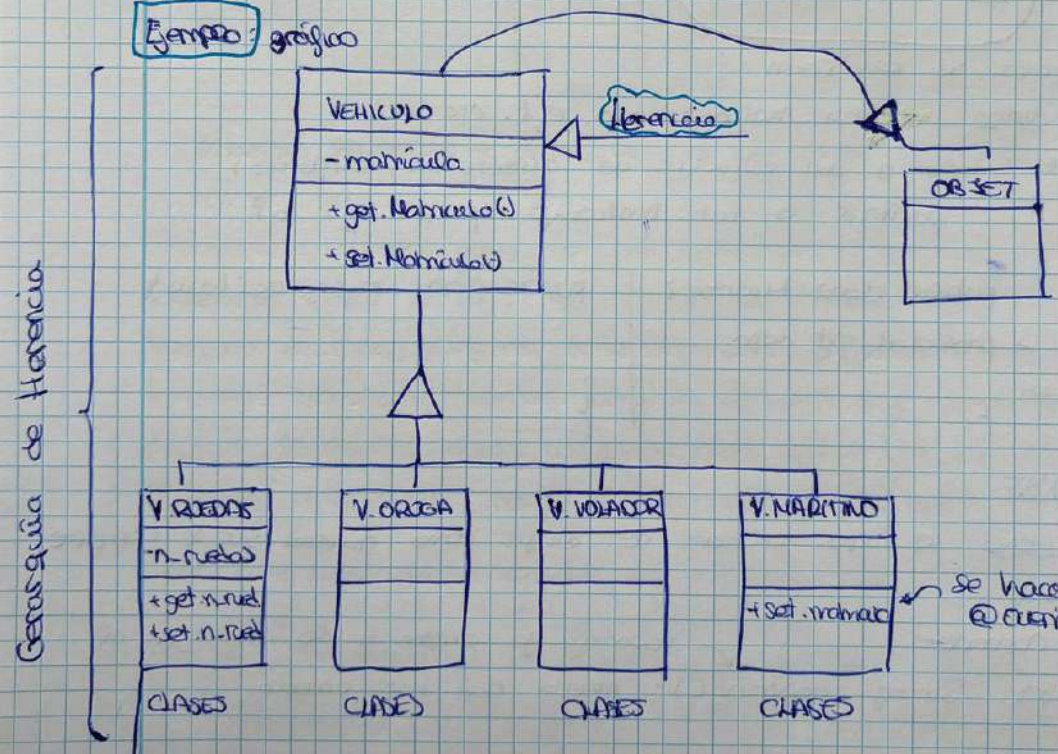
determinada manera.
La anotación @ dice a Java que llamas a un método como uno ya predefinido y que cuando llamas al método hagas que use el tuyo y no el predeterminado.
Todas las clases están heredadas de la clase objeto que define unos métodos especiales como toString().

HERENCIA

- Clases abstractas.



Ejemplo gráfico



Especializamos el comportamiento de los datos de un caso más concreto (puede no ser el definitivo).
Todo lo que tiene el padre lo tiene la hija pero hay cosas que quedan privadas y los "hijos" solo acceden a las cosas públicas o protegidas.


```
V_RUEDAS V1 = new V_RUEDAS
V1.setMatricula("848...a")
```

Ejemplo en código:

```
public class Vehiculo {
    private String matricula;
    public String getMatricula() {
        return matricula
    }
    public void setMatricula (String m){
        this.matricula = m;
    }
}
=
public class V_RUEDAS extends Vehiculo {
    private int V_ruedas;
    public int getNRuedas() {
        return nRuedas;
    }
    public void setNRuedas(int n){
        this.n_ruedas = n;
    }
}
-
public class V_Volador extends Vehiculo { } ← VACÍA (hereda de vehículo)
=
public class Motocicleta extends V_Ruedas { }
-
```

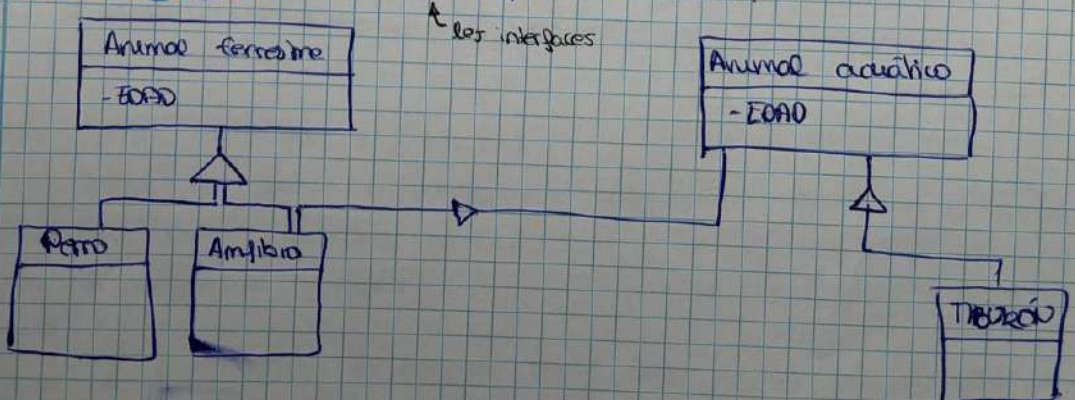
En vehículo nunca puedes instanciarlo = crear una variable de tipo Vehículo y hacerle un new.

Hay que indicar que vehículo es abstracto y que no se puede concretar en un objeto de esa clase. Me permite modelar el problema bien.

Se hace → se indica → añadiendo public abstract class.

Herencia multiple: (En Java no, en C++ sí).

↑ los interfaces



En Java se utilizan interfaces en vez de herencia multiple

Ej
```
interface terrestre { // Firma de métodos
    public int getEdad ();  <- No se implementa, solo cabecera
}
```

=
```
interface acuatico {
    public int get....()
}
```

Despues de definirlas hay que implementarlas / programarlas

Ej:
```
class anfibio implements terrestre, acuatico
    public int getEdad () {
        ==
    }
```

Te obliga a implementar los métodos de otras clases. Puedes establecer el comportamiento.

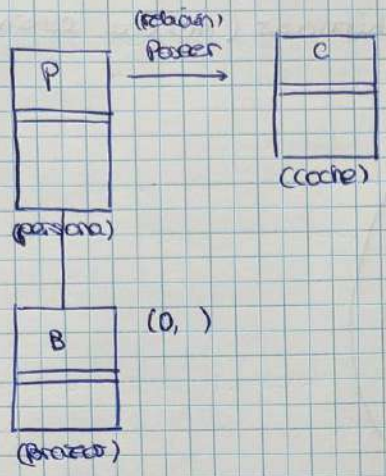
Puedes establecerlos sin necesidad de hacer herencias y programas como quieras.

Una interface la puedes declarar para 1 variable.

```
terrestre t1 = newPerro ()
```
↑
interface.

MULTIPLICIDAD / CARDINALIDAD.

La cardinalidad es una manera de poder modelar las relaciones entre clases en cuanto al número de elementos que esas clases tienen en relación.

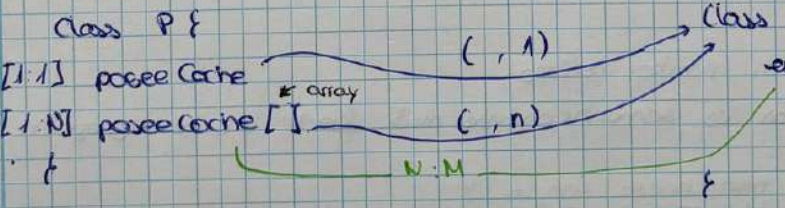


mínimos : 0, 1
máximos : n, 1
} CARDINALIDAD

→	←
(0,1)	(0,1)
(0,n)	(0,n)
(1,1)	(1,1)
(1,n)	(1,n)

Las ~~asociaciones~~ condiciones de contorno dice que está fuera y dentro del contorno.
El cero dice que no hay relación (ausencia). El 1 dice que sí hay relación.
Para el máximo pueden ser o 1 relación o múltiples relaciones.
(máx 1, máx n)
El mín / max no son valores sino que representan condiciones de contorno
Puede ser que el máximo tenga multiplicidad y entonces es n o que no la tenga y entonces sea 1. (n es + de 1)
Hay un conjunto de posibilidades
(0,1) (0,n) (1,1), (1,n) ← ¿Cuál es la correcta? Depende.

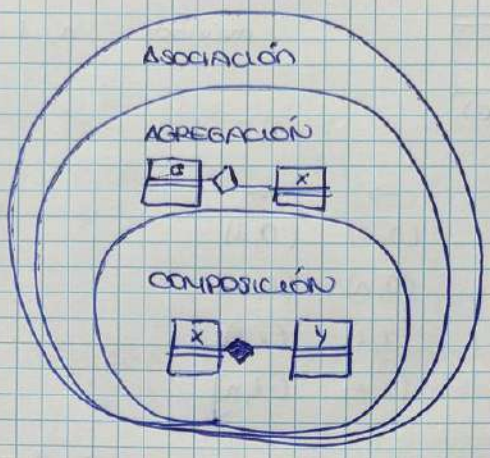
ASOCIACIÓN.
La relación entre 2 clases de tipo asociación se establece mediante variables.



Las formas normales ≠ [N:M]
Entre base de datos e ingeniería software tenemos el mismo problema, las soluciones varían.
Hay 3 categorías posibles

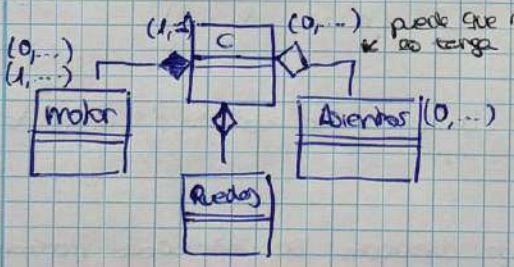
1:1
1:N
N:M
} Combinan el máximo de 1 lado con el máximo del otro.

Cada vez que meto una variable en unos atributos de una clase a otra, estableceo la relación entre ellas.
La herencia es estático, ~~la asociación es~~
Las relaciones de asociación son híbridas (mezcla estático, dinámico)



AGREGACIÓN

La relación de agregación se marca con un rombo.
La diferencia es semántica, sintácticamente se programa igual.
A nivel de diseño UML sí hay diferencia.
La relación de agregación representa que existe algún tipo de "propiedad" entre 1 clase y otra. La clase padre "posee" gran parte de la clase hija



La diferencia entre ◇ y — es semántica.
Sin rombo, el motor existe sin depender del coche
Con rombo, el coche en parte se compone de componentes externos que existen a su vez pero existe un grado semántico de propiedad. →
(El motor sólo puede funcionar para ese coche)

COMPOSICIÓN

Se dibuja con un rombo relleno.
En la composición es semánticamente más fuerte que la agregación y asociación.
La clase X → relacionada con la Y existe porque existe la clase Y y la clase Y existe porque existe la clase X
A la hora de programar es igual, la ~~diferencia~~ diferencia es, que estén intentando decir con unas relaciones u otras.