Paradigma Orientado a Objetos

Relaciones

ASOCIACIÓN ENTRE CLASES

En la programación orientada a objetos el punto de partida para la construcción de un sistema es un proceso de abstracción y clasificación.

Los **objetos de una clase** se caracterizan por los mismos atributos y comportamiento, pero además **comparten entre sí el mismo modo de relacionarse con objetos de otras clases**.

Un objeto está **asociado** a otro objeto, si **tiene un** atributo de su clase.

La relación entre los objetos provoca una relación entre las clases, que se dicen **asociadas**.

CASO DE ESTUDIO: SIGNOS VITALES

*Los **signos vitales** son medidas de variaciones fisiológicas que permiten valorar las funciones corporales básicas.*

Dos de los signos vitales son:

*la **temperatura corporal***

*y la **presión arterial**.*

El profesional considera que existe un principio de alarma cuando estos valores escapan de los umbrales establecidos.

CASO DE ESTUDIO: SIGNOS VITALES

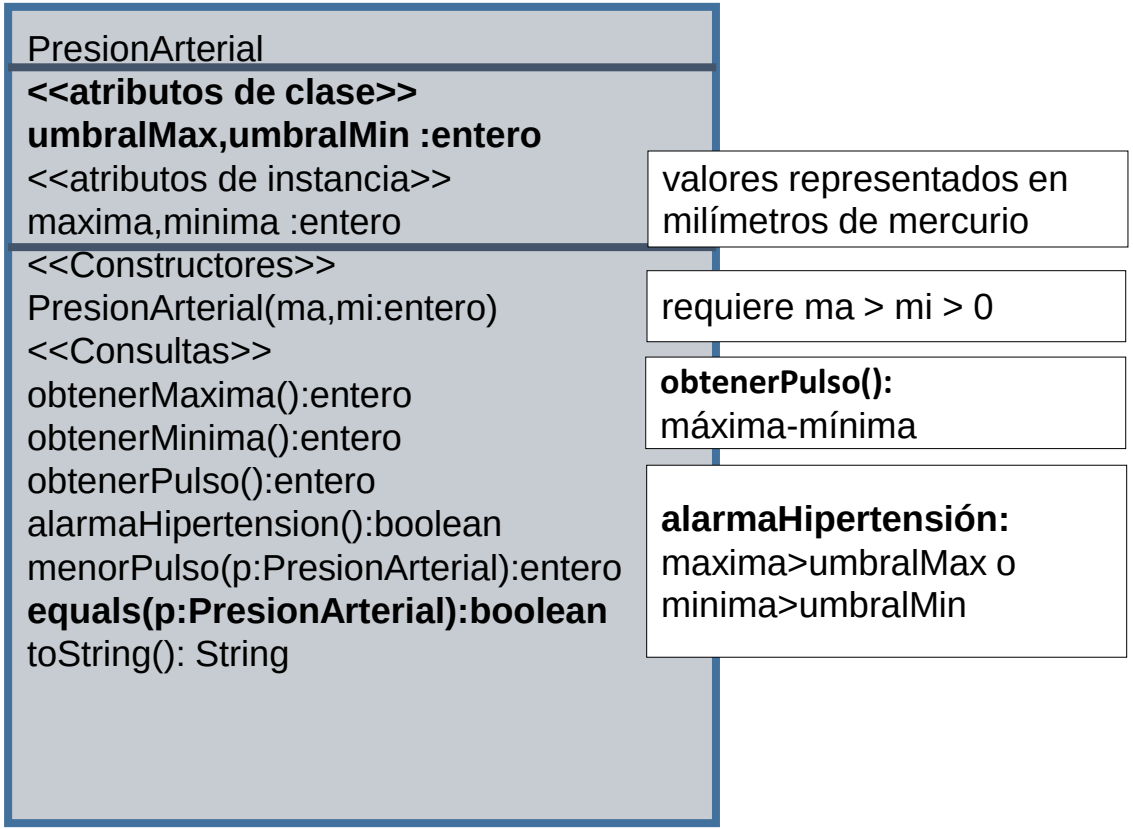
*La **presión arterial** es la fuerza de presión ejercida por la sangre circulante sobre las arterias y constituye uno de los principales **signos vitales** de un paciente.*

…

Los valores de la presión sanguínea se expresan en kilopascales (kPa) o en milímetros del mercurio (mmHg).

Para convertir de mmHg a kPa el valor se multiplica por 0,13.

CASO DE ESTUDIO: SIGNOS VITALES



CASO DE ESTUDIO: SIGNOS VITALES

valores representados en grados centígrados

SignosVitales(t:real, p:PresionArterial
requiere t > 0 p ligado

alarma:
alarmaHipertensión o temperatura>umbralTemp

SignosVitales

<<atributos de clase>>
umbralTemp:real
<<atributos de instancia>>
temperatura: real
presion: **PresionArterial**

<<Constructores>>
SignosVitales(t:real, **p:PresionArterial**)
<<Consultas>>
obtenerTemperatura():real
obtenerPresion():PresionArterial
alarma():boolean
equals(s:SignosVitales):boolean
toString(): String

CASO DE ESTUDIO: SIGNOS VITALES

PresionArterial

<<atributos de clase>>
umbralMax,umbralMin :entero
<<atributos de instancia>>
maxima,minima :entero

<<Constructores>>
PresionArterial(ma,mi:entero)
<<Consultas>>
obtenerMaximaMM():entero
obtenerMinimaMM():entero
obtenerMaximaHP():entero
obtenerMinimaHP().entero
obtenerPresionPulsoMM():entero
obtenerPresionPulsoHP():entero
alarmaHipertension():boolean
menorPulso(p:PresionArterial):entero
equals(p:PresionArterial):boolean
toString(): String

SignosVitales

<<atributos de clase>>
umbralTemp:real
<<atributos de instancia>>
temperatura: real
presion: **PresionArterial**

<<Constructores>>
SignosVitales(t:real,**p: PresionArterial**
)
<<Consultas>>
obtenerTemperatura():real
obtenerPresion(): PresionArterial
alarma ():boolean
equals(s:SignosVitales):boolean
toString(): String

CASO DE ESTUDIO: SIGNOS VITALES

```
class PresionArterial {
    /*Valores representados el milímetros de mercurio*/

    //Atributos de clase
    private static final int umbralMax=120;
    private static final int umbralMin=80;

    //Atributos de instancia
    private int maxima;
    private int minima;

    //Constructor
    public PresionArterial(int ma,int mi){
    //Requiere ma > mi > 0
        maxima = ma;
        minima = mi;
    }

    …
}
```

CASO DE ESTUDIO: SIGNOS VITALES

```
//Consultas
public int obtenerMaxima(){
  return maxima;
}
public int obtenerMinima(){
  return minima;
}
```

CASO DE ESTUDIO: SIGNOS VITALES

```
public int obtenerPresionPulso(){
  return maxima-minima;
}
public String toString (){
  return maxima+ " " +minima;
}
```

CASO DE ESTUDIO: SIGNOS VITALES

La clase **`SignosVitales`** **tieneUn** atributo de clase **`PresionArterial`**.

La clase **`SignosVitales`** puede acceder a cualquiera de los miembros públicos de la clase **`PresionArterial`**.

Los atributos están **encapsulados**, no son accesibles, fuera de la clase.

```
class SignosVitales{
    //Atributos de clase
    private static final int umbralTemp=38;

    //Atributos de instancia
    private float temperatura;
    private PresionArterial presion;
```

CASO DE ESTUDIO: SIGNOS VITALES

```
class SignosVitales{

    //Atributos de clase
    private static final umbralTemp=38;

    //Atributos de instancia
    private float temperatura;
    private PresionArterial presion ;

    //Constructor
    public SignosVitales (float t, PresionArterial p ){

    //Requiere t > 0  y p ligada
    temperatura = t;
    presion = p;
}
…
```

CASO DE ESTUDIO: SIGNOS VITALES

```
class SignosVitales{
    …
    //Consultas
    public float obtenerTemperatura (){
        return temperatura;
    }

    public PresionArterial obtenerPresion (){
        return presion ;
    }

    public boolean alarma(){
        return temperatura > umbralTemp || presion.alarmaHipertension() ;
    }
}
```

CASO DE ESTUDIO: SIGNOS VITALES

La clase **`SignosVitales`** define un método **`toString()`** que envía el mensaje **`toString`**() a un objeto de clase **`PresioArterial`**.

NO ES UNA LLAMADA RECURSIVA.

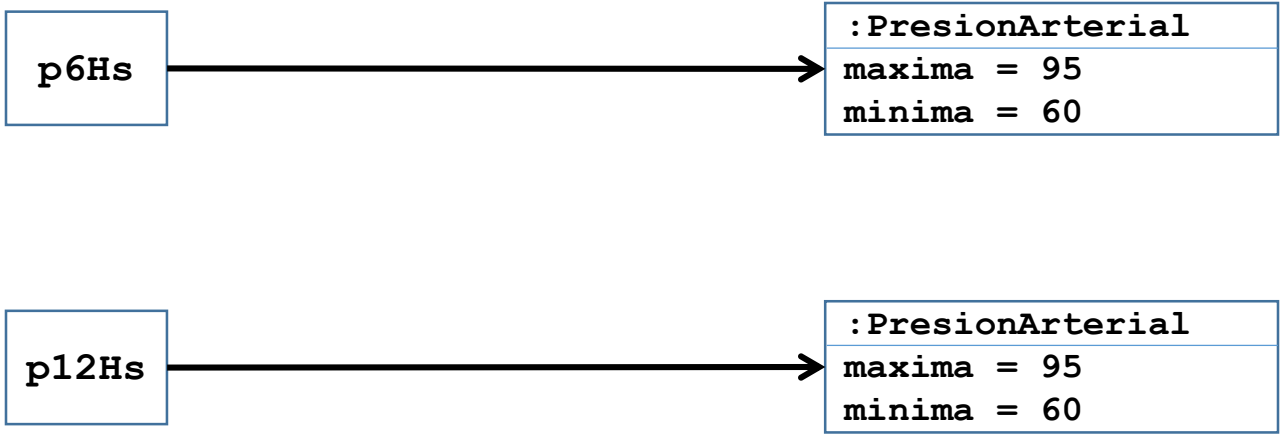
```
class SignosVitales{
…
    //Consultas
    public String toString(){
        return temperatura+" "+presion.toString();
    }
…
}
```

CASO DE ESTUDIO: SIGNOS VITALES

```
class Control{
    public static void main (String [] a){

    //Creación de objetos
    PresionArterial p6Hs= new PresionArterial(95,60);
    PresionArterial p12Hs= new PresionArterial(95,60);

    SignosVitales s6Hs = new SignosVitales(36.2,p6Hs);
    SignosVitales s12Hs = new SignosVitales(38,p12Hs);

    if (s6hs.alarma() || s12hs.alarma())
        System.out.println("Estado de alarma");
    }
}
```

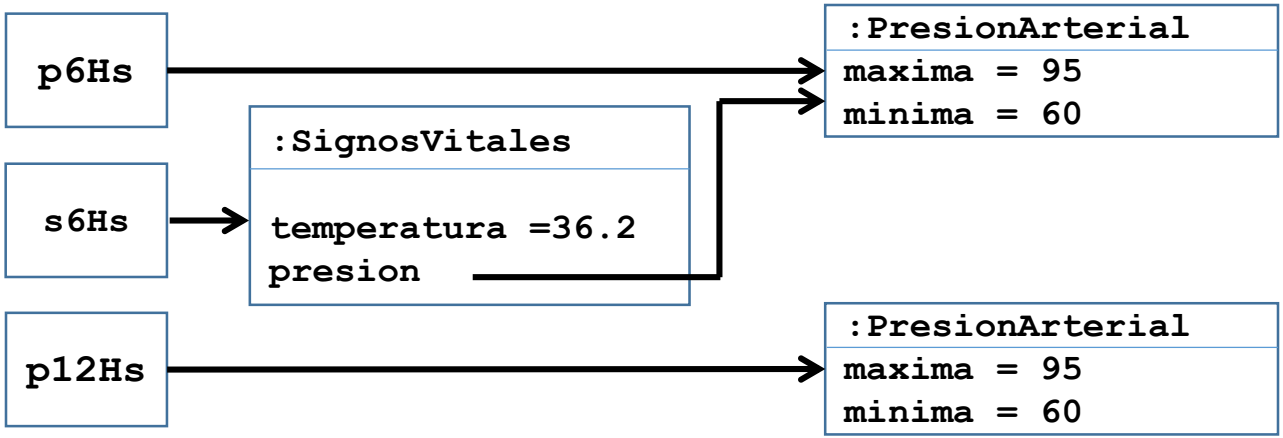
CASO DE ESTUDIO: SIGNOS VITALES

```
PresionArterial p6Hs= new PresionArterial(95,60);
PresionArterial p12Hs= new PresionArterial(95,60);
```



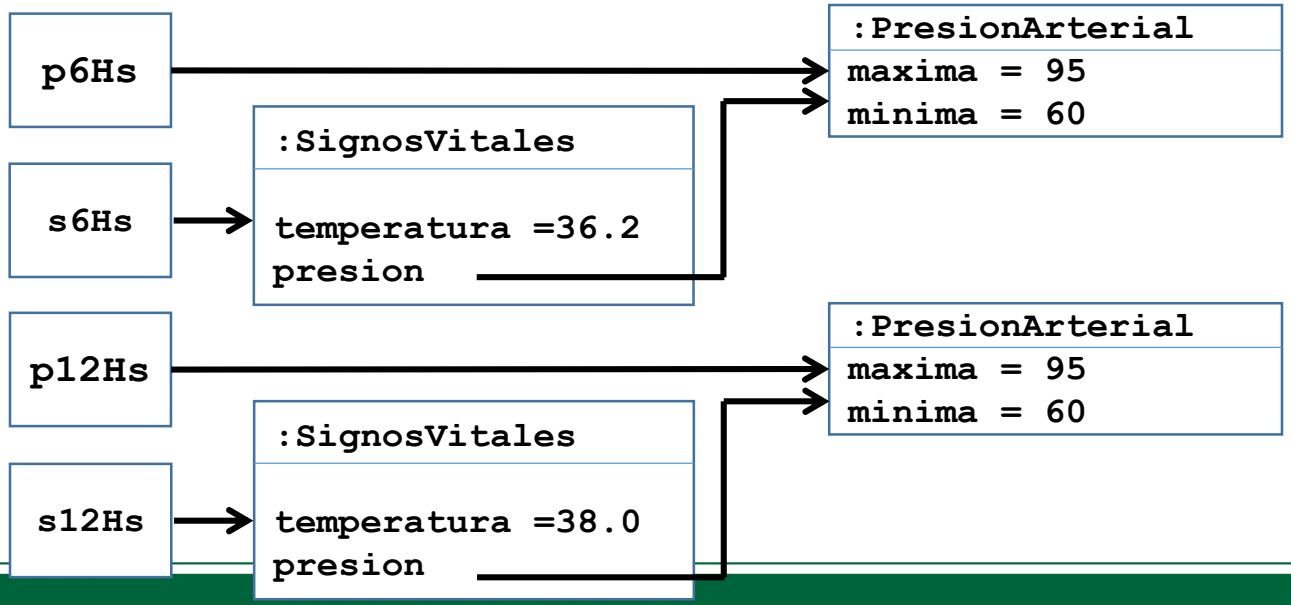
CASO DE ESTUDIO: SIGNOS VITALES

```
PresionArterial p6Hs= new PresionArterial(95,60);
PresionArterial p12Hs= new PresionArterial(95,60);
SignosVitales s6Hs = new SignosVitales(36.2,p6Hs);
```



CASO DE ESTUDIO: SIGNOS VITALES

```
PresionArterial p6Hs= new PresionArterial(95,60);
PresionArterial p12Hs= new PresionArterial(95,60);
SignosVitales s6Hs = new SignosVitales(36.2,p6Hs);
SignosVitales s12Hs = new SignosVitales(38,p12Hs);
```



PROVEEDORES Y CLIENTES

La clase **`PresionArterial`** **brinda** servicios que la clase **`SignosVitales`** **usa**.

Decimos que clase **`PresionArterial`** cumple el rol de **proveedora** y **`SignosVitales`** es su **cliente**.

La clase **`Control`** también **usa** los servicios de **`PresionArterial`** y además **usa** a **`SignosVitales`**.

De modo que **`SignosVitales`** es al mismo tiempo **cliente** y **proveedora**.

CONTRATO

La clase **`SignosVitales`** puede implementarse conociendo **qué** hace la clase **`PresionArterial`** , pero no **cómo** lo hace.

La clase **`PresionArterial`** puede implementarse sin saber que va a ser usada por la clase **`SignosVitales`**.

Es decir, cada clase debe conocer los servicios que brindan sus clases proveedoras, pero no necesita conocer quienes son sus clientes.

Cada clase va a ser verificada por separado y luego en conjunto con las demás clases relacionadas.

Las responsabilidades establecen un **contrato** entre una clase, sus clientes y sus proveedores.

DEPENDENCIA ENTRE CLASES

La **dependencia** entre clases se produce cuando una clase declara una **variable local** o un **parámetro** de otra clase.

Decimos que la relación entre objetos es del tipo **usaUn/usaUna.**

Notemos que un caso particular de dependencia se presenta entre la clase Tester con la clase que va a ser verificada.

CASO DE ESTUDIO: FABRICA DE JUGUETES

*En una fábrica de autos de juguete una parte de la producción la realizan **robots**.*

*Cada robot tiene un **número de serie**.*

En el momento que se crea el robot se establece su número de serie, que nunca va a cambiar.

*Cada robot tiene una **carga de energía** que se va consumiendo a medida que ejecuta las órdenes que recibe.*

CASO DE ESTUDIO: FABRICA DE JUGUETES

*Cada robot es capaz de conectarse de modo tal que se recargue su energía hasta su **capacidad máxima de 5000 unidades**.*

*Esta acción puede ejecutarse ante una orden externa o puede iniciarla el robot mismo cuando **su energía está por debajo de las 100 unidades**.*

El robot recarga su energía cuando después de armar un juguete, queda por debajo del mínimo.

CASO DE ESTUDIO: FABRICA DE JUGUETES

Cada robot tiene la capacidad para armar autos de juguete y cuenta con piezas de diferentes tipos: ruedas, ópticas y chasis.

Inicialmente comienza a trabajar con 100 piezas de cada tipo.

*La cantidad de piezas se incrementa cuando un robot recibe una orden de abrir una **caja de piezas** y se decrementa cuando arma un vehículo.*

Cada caja tiene piezas de todos los tipos.

Desarmar una caja cualquiera demanda 50 unidades de energía.

Armar un auto consume 70 unidades de energía, 4 ruedas, 6 ópticas y 1 chasis.

CASO DE ESTUDIO: FABRICA DE JUGUETES

Robot

<<atributos de clase>>

energiaMaxima : 5000
energiaMinima : 100

<<atributos de instancia>>

nroSerie:entero
energia: entero
ruedas: entero
opticas: entero
chasis: entero
…

Caja

<<atributos de instancia>>

ruedas: entero
opticas: entero
chasis: entero
…

CASO DE ESTUDIO: FABRICA DE JUGUETES

Robot

...

<<constructores>>

Robot(nro: entero)

<<comandos>>

abrirCaja (**c: Caja**)

...

Robot(nro: entero)
Inicializa la energía en el valor máximo y las cantidades de piezas en 100.

abrirCaja (c: Caja)
Requiere que se vacíe la caja después de que el robot la abra

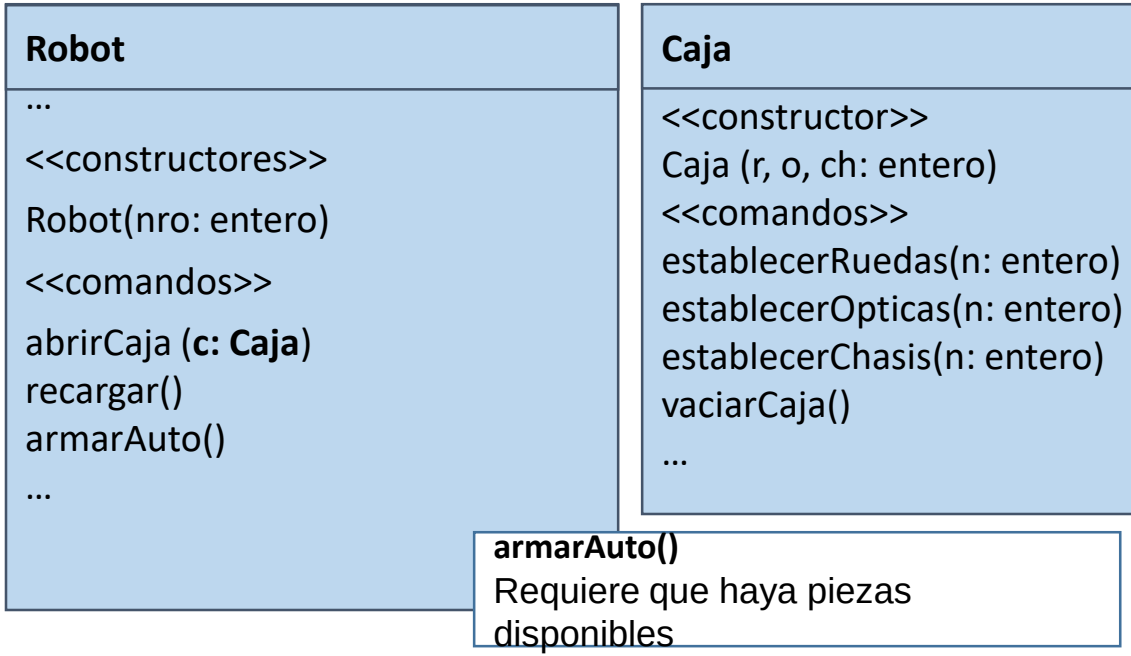
CASO DE ESTUDIO: FABRICA DE JUGUETES

Robot
… <<constructores>> Robot(nro: entero) <<comandos>> abrirCaja (**c: Caja**) …

Caja
<<atributos de instancia>> ruedas: entero opticas: entero chasis: entero <<constructor>> Caja (r, o, ch: entero) <<comandos>> establecerRuedas(n: entero) establecerOpticas(n: entero) establecerChasis(n: entero) vaciarCaja()

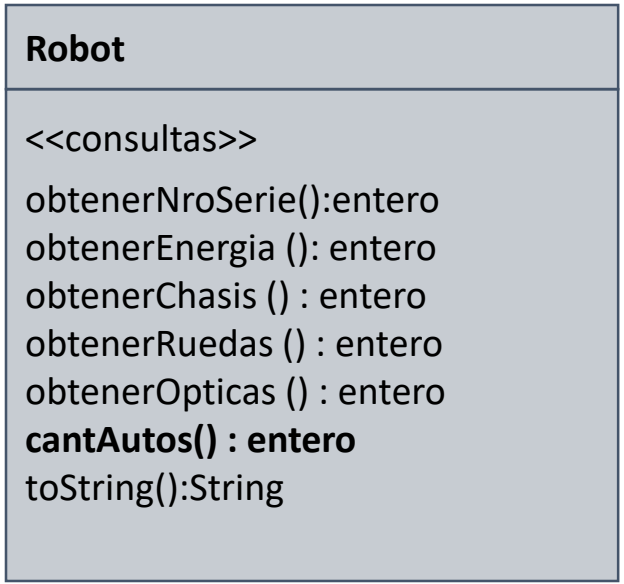
Existe una **relación de dependencia** entre las clases **`Robot`** y **`Caja`**.

CASO DE ESTUDIO: FABRICA DE JUGUETES



La clase que usa a **`Robot`** debe asumir la responsabilidad de verificar si el robot tiene piezas disponibles antes de enviar el mensaje **`armarAuto`**.

CASO DE ESTUDIO: FABRICA DE JUGUETES



La consulta **`cantAutos()`** puede usarse para decidir si hay piezas disponibles para armar un auto.

CASO DE ESTUDIO: FABRICA DE JUGUETES

Robot
<<Responsabilidades>> Todos los servicios que consumen energía deciden recargar cuando energía es menor que la mínima.

CASO DE ESTUDIO: FABRICA DE JUGUETES

- **`recargar()`** recarga la energía del robot hasta llegar al máximo.
- **`abrirCaja(c:Caja)`** aumenta las piezas disponibles de acuerdo a las cantidades de la caja. Requiere que la clase que envía el mensaje, vacíe la caja después de darle la orden abrirCaja al robot.
- **`armarAuto()`** decrementa las piezas disponibles, requiere que la clase que envía el mensaje, haya controlado si hay piezas disponibles antes de enviar el mensaje armarAuto a un robot.
- **`cantAutos():entero`** retorna la cantidad de autos que puede armar el robot con las piezas que tiene disponibles, sin desarmar una caja.

CASO DE ESTUDIO: FABRICA DE JUGUETES

Robot

<<atributos de clase>>

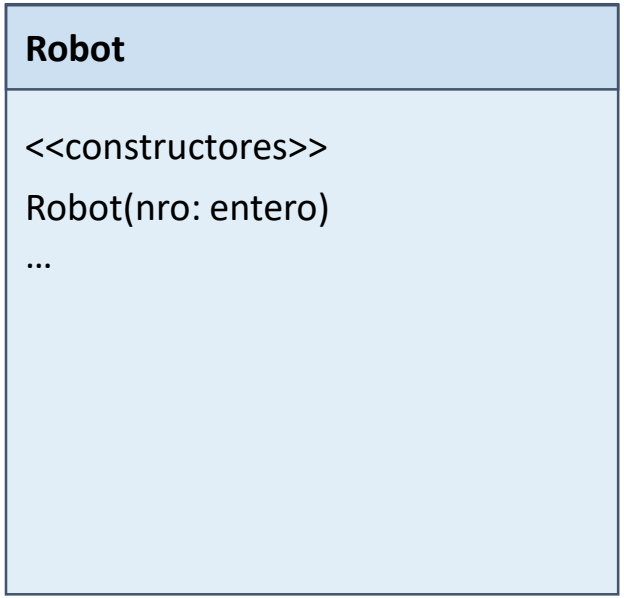
energiaMaxima : 5000
energiaMinima : 100

<<atributos de instancia>>

nroSerie:entero
energia: entero
ruedas: entero
opticas: entero
chasis: entero
…

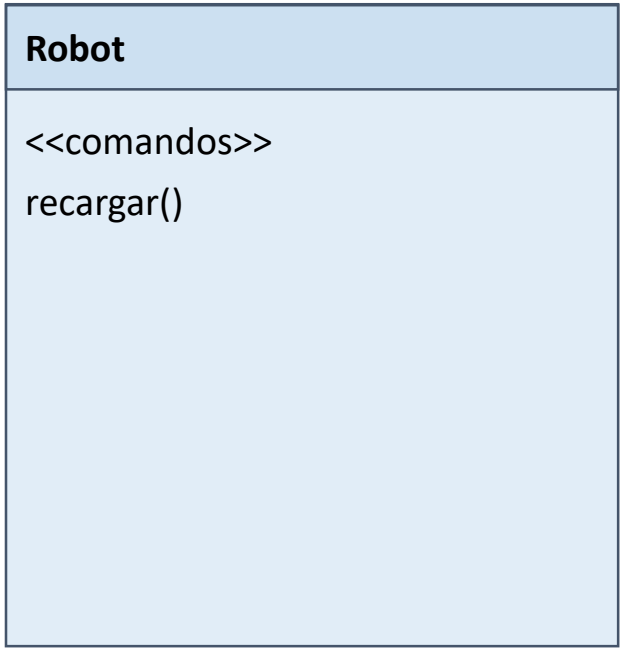
```
class Robot {

    //atributos de clase
    private static final int energiaMaxima = 5000;
    private static final int   energiaMinima = 100;

    //atributos de instancia
    private int nroSerie;
    private int energia;
    private int ruedas;
    private int opticas;
    private int chasis;
    …
}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES



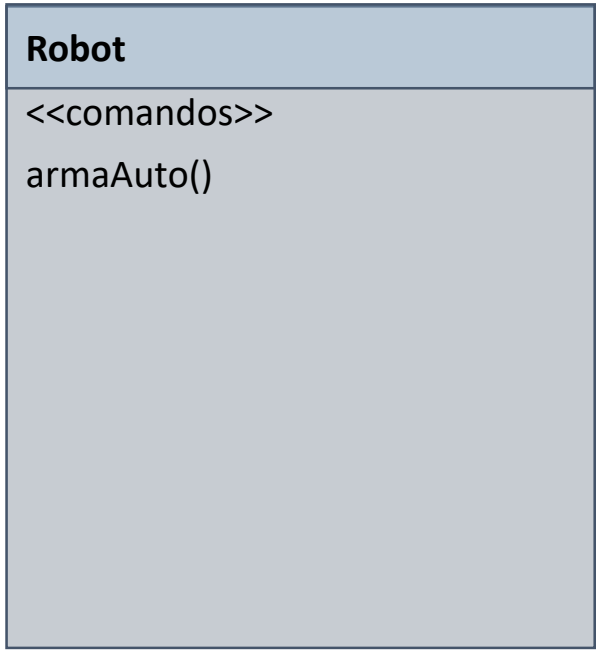

```
class Robot {
    …
    //Constructores
    public Robot (int nro){
        nroSerie = nro;
        energia = energiaMaxima;
        ruedas = 100;
        opticas = 100;
        chasis = 100;
    }
    …
}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES



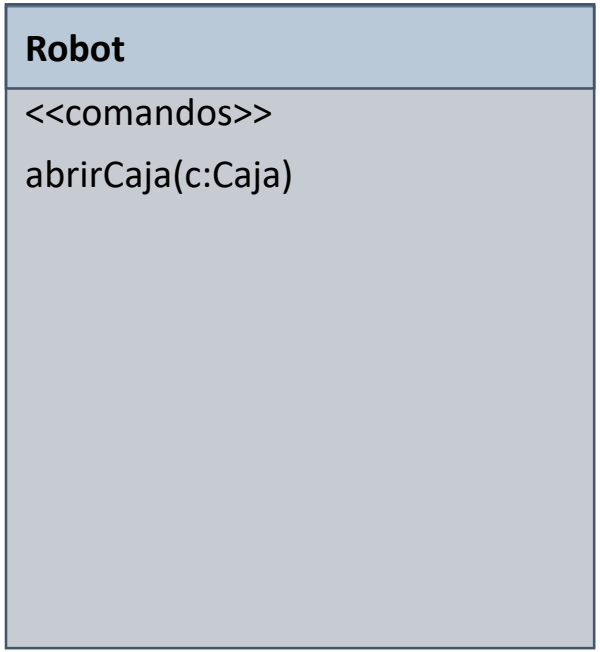

```
public void recargar(){
   energia = energiaMaxima;
}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES




```
public void armarAuto () {
    /* Requiere que se haya
     * controlado si hay piezas
     * disponibles*/
    ruedas -= 4 ;
    opticas -= 6;
    energia -= 70;
    chasis --;
    /*Controla  si es necesario
    recargar energía*/
    if (energia < energiaMinima)
        this.recargar();
}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES



El comando **`abrirCaja`** recibe como parámetro un objeto de clase **`Caja`**.

CASO DE ESTUDIO: FABRICA DE JUGUETES

```
public void abrirCaja (Caja caja) {
    /*Aumenta sus cantidades según las de la caja*/
    ruedas += caja.obtenerRuedas();
    opticas += caja.obtenerOpticas();
    chasis += caja.obtenerChasis();
    energia -= 50;

    /*Controla  si es necesario recargar energía*/
    if (energia < energiaMinima)
        this.recargar();
}
```

Existe una relación de dependencia entre **`Robot`** y **`Caja`**

CASO DE ESTUDIO: FABRICA DE JUGUETES

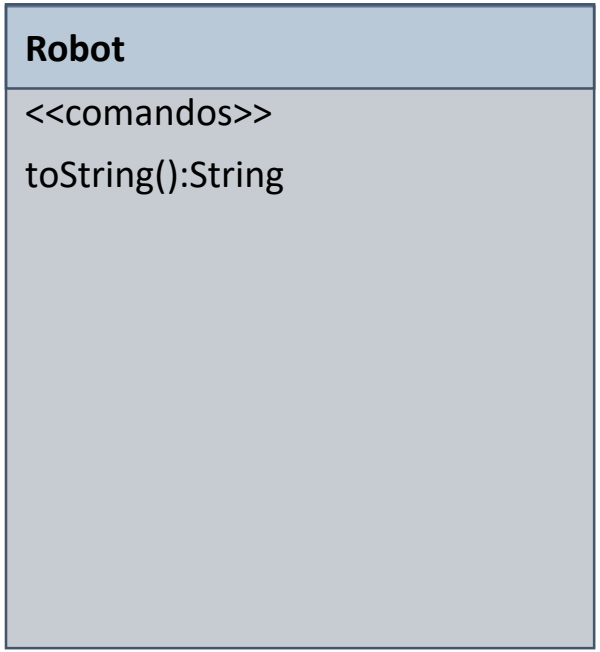
Robot

…
<<consultas>>

obtenerNroSerie(): entero
obtenerEnergia (): entero
obtenerChasis () : entero
obtenerRuedas () : entero
obtenerOpticas () : entero

```
//Consultas
public int obtenerNroSerie( ){
  return nroSerie;}

public int obtenerEnergia( ){
  return energia;}

public int obtenerRuedas( ){
  return ruedas;}

public int obtenerOpticas( ){
  return opticas;}

public int obtenerChasis( ){
  return chasis;}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES




```
public String toString(){
  return nroSerie+" "
         ruedas+" "+opticas+
         " "+ chasis;
}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES

El contrato requiere que **usa** a **`Robot`** controle si es posible armar el auto.

```
class FabricaJuguetes{
…
    public void producir {
      Robot unRobot;
      unRobot = new Robot(111);
      Caja c = new Caja(100,150,25);
      …
      if (unRobot.cantAutos() > 0){
          unRobot.armarAuto();
      …
    }
}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES

Los valores de los atributos de instancia se establecen en la creación del objeto y se modifican cuando se arman autos o se abren cajas.

Los atributos, los comandos y las consultas de **`Caja`** tienen los mismos nombres que en **`Robot`**, cuando un objeto reciba un mensaje su clase determina el método que va a ejecutarse.

La clase **`FabricaJuguetes`** depende de las clases **`Robot`** y **`Caja`** porque tiene variables locales de esas clases.

CASO DE ESTUDIO: FABRICA DE JUGUETES

Robot
…
<<constructores>>
Robot(nro: entero)
<<comandos>>
abrirCaja (c: Caja)
recargar()
armarAuto():**boolean**
…

Caja
…
<<constructor>>
Caja (r,o,ch: entero)
<<comandos>>
establecerRuedas(n: entero)
establecerOpticas(n: entero)
establecerChasis(n: entero)

Controla si hay piezas disponibles. Si no hay piezas disponibles retorna false.

El cambio de diseño no modifica la **funcionalidad** de la clase, sino su **responsabilidad**.

CASO DE ESTUDIO: FABRICA DE JUGUETES

```
public boolean armarAuto () {
    /*Controla si hay piezas disponibles. Si no hay piezas disponibles retorna false.*/
    boolean hayPiezas = false;
    if(cantAutos()> 0){
        hayPiezas = true;
        ruedas -= 4 ;
        opticas -=6;
        energia -= 70;
        chasis --;
        /*Controla  si es necesario recargar energía*/
        if (energia < energiaMinima)
            this.recargar();
    }
    return hayPiezas;
}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES

Las responsabilidades establecidas por el diseñador requieren que la clase **Robot** controle si es posible armar el auto y retorne un valor boolean.

```
class FabricaJuguetes{
…
public void producir {
  Robot unRobot;
  unRobot = new Robot(111);
  …

  if (unRobot.armarAuto())
    …
}
}
```

CASO DE ESTUDIO: FABRICA DE JUGUETES

Las responsabilidades establecidas por el diseñador requieren que la clase que envía el mensaje **abrirCaja** a un objeto de clase **Robot** vacíe la caja después de que el robot la abra.

```
class FabricaJuguetes{
…
public void producir {
  Robot unRobot;
  unRobot = new Robot(111);
  Caja c = new Caja(100,150,25);
  …
  unRobot.abrirCaja(c);
  c.vaciarCaja();
  …
}
}
```

RELACIONES ENTRE CLASES

Un sistema orientado a objetos se construye a partir de una colección de **clases relacionadas entre sí.**

Algunas clases cumplen el rol de **proveedoras** de servicios.

Algunas clases cumplen el rol de **clientes** de los servicios que brindan las clases proveedoras.

Algunas clases son proveedoras de servicios y a su vez clientes de otras clases proveedoras.

Las funcionalidades y responsabilidades establecen un **contrato** entre las clases relacionados.

RELACIONES ENTRE CLASES

Asociación

La **asociación** es una forma de relación entre clases y se produce cuando el modelo de un objeto del problema **contiene** o **puede contener** al modelo de otro objeto del problema.

Dependencia

La **dependencia** es una forma de relación entre clases y se produce cuando el modelo de un objeto del problema **usa** al modelo de otro objeto.

DEPENDENCIA ENTRE CLASES

Cuando una clase A brinda un servicio que **declara una variable local, retorna un resultado** o **recibe un parámetro** de otra clase B, decimos que:

- la clase A depende de la clase B y
- la relación es de tipo **usaUn**.

En particular existe una relación de dependencia entre una clase Tester y la clase verificada.

ASOCIACIÓN ENTRE CLASES

Cuando una clase A **tiene un** atributo de instancia de otra clase B, decimos que

- las clases están **asociadas** y
- la relación es de tipo **tieneUn**.

Entre dos clases asociadas en general también se establece una relación de dependencia.

Algunos de los servicios de la clase cliente, recibirán como parámetro o retornarán un resultado de la clase proveedora.

Así, una clase A que **tiene** un atributo de la clase B, **usa** a la clase B.

CASO DE ESTUDIO: VIDEOJUEGO

*En un videojuego algunos de los personajes son **aliens**.*

*Los aliens tienen cierta cantidad de **antenas** y de **manos**, que determinan su capacidad sensora y su capacidad de lucha respectivamente.*

*Cada alien tiene una **cantidad de vidas** que inicialmente es 5 y se van reduciendo cada vez que recibe una herida.*

Cuando está muerto ya no tienen efecto las heridas.

CASO DE ESTUDIO: VIDEOJUEGO

Cuando un alien logra llegar a la base recupera 2 vidas, sin superar nunca el valor 5.

*Cada alien **tiene una nave**, cada nave tiene una **velocidad** y una **cantidad de combustible** en el tanque.*

Ambos atributos pueden aumentar o disminuir de acuerdo a un parámetro que puede ser positivo o negativo.

La fuerza de un alien se calcula como la capacidad sensora, más su capacidad de lucha, todo multiplicado por el número de vidas por un quinto de la velocidad de la nave.

CASO DE ESTUDIO: VIDEOJUEGO

Alien

<<Atributos de clase>>
maxVidas:entero
<<Atributos de instancia>>
Nave: **NaveEspacial**
vidas:entero
antenas:entero
manos:entero

<<Constructores>>
Alien (n:NaveEspacial,v,a,m:entero)
<<Comandos>>
recuperaVidas()
recibeHerida()
copy(a:Alien)
establecerNave(n:NaveEspacial)
establecerAntenas(p:entero)
establecerManos(p:entero)
establecerVidas(p:entero)

Alien(n:NaveEspacial,
v,a,m:entero)
Requiere n ligado y v <= 5

copy(a:Alien)
Requiere a ligado

CASO DE ESTUDIO: VIDEOJUEGO

Alien

<<Atributos de clase>>
maxVidas:entero
<<Atributos de instancia>>
Nave: **NaveEspacial**
vidas:entero
antenas:entero
manos:entero

…
<<Consultas>>
obtenerNave():NaveEspacial
obtenerVidas():entero
obtenerAntenas():entero
obtenerManos():entero
obtenerFuerza():real
clone():Alien
equals(a: Alien): boolean

obtenerFuerza():real
Requieren Nave ligado

equals(a: Alien): boolean
Requiere a ligado

CASO DE ESTUDIO: VIDEOJUEGO

NaveEspacial

<<Atributos de instancia>>
velocidad: entero
combustible: entero

<<Constructores>>
NaveEspacial(v, c: entero)

<<Comandos>>
cambiarVelocidad(v: entero)
cambiarCombustible(c: entero)
copy(n: NaveEspacial)

<<Consultas>>
obtenerVelocidad(): entero
obtenerCombustible(): entero
equals(n:NaveEspacial): boolean
clone(): NaveEspacial

NaveEspacial(v, c: entero)
Requiere c>=0, v>=0

cambiarCombustible(p: entero)
Si combustible+p >=0
  combustible = combustible+p

cambiarVelocidad(p:entero)
Si velocidad+p >=0
  velocidad = velocidad+p

CASO DE ESTUDIO: VIDEOJUEGO

La clase **`Alien`** es **cliente** de la clase **proveedora `NaveEspacial`**.

La clase **`Alien`** solo necesita conocer la **signatura** de los servicios provistos por **`NaveEspacial`**.

La clase **`NaveEspacial`** no necesita conocer a la clase **`Alien`** ni a ninguna de sus clases clientes.

CASO DE ESTUDIO: VIDEOJUEGO

```
class NaveEspacial {

    //Atributos de instancia
    private int velocidad;
    private int combustible;

    //Constructor
    public NaveEspacial(int v,int c){
        //Requiere v>= 0 c>= 0
        velocidad = v;
        combustible = c;
    }
   ...
}
```

CASO DE ESTUDIO: VIDEOJUEGO

```
//Comandos
public void cambiarVelocidad(int p){
    if (velocidad+p >=0)
        velocidad= velocidad+p;}

public void cambiarCombustible (int p){
    if (combustible+p >=0)
        combustible = combustible+p;}

//Consultas
public int obtenerVelocidad( ){
    return velocidad;}

public int obtenerCombustible ( ){
    return combustible;}
```

CASO DE ESTUDIO: VIDEOJUEGO

```
class NaveEspacial {

    //Atributos de instancia
    private int velocidad;
    private int combustible;

    …

    public void copy(NaveEspacial n){
    //Requiere n ligada.
        velocidad= n.obtenerVelocidad();
        combustible= n.obtenerCombustible();
    }
}
```

El comando **copy** copia el estado interno del objeto ligado al parámetro, en el estado interno del objeto que recibe el mensaje.

CASO DE ESTUDIO: VIDEOJUEGO

```
class NaveEspacial {
    //Atributos de instancia
    private int velocidad;
    private int combustible;

    …

    public boolean equals(NaveEspacial n){
        //Requiere n ligada
        return velocidad== n.obtenerVelocidad() && combustible== n.obtenerCombustible();
    }
}
```

La consulta **`equals`** decide si dos objetos son equivalentes, esto es, retorna true si el objeto que recibe el mensaje tiene el mismo estado interno que el objeto ligado al parámetro.

CASO DE ESTUDIO: VIDEOJUEGO

```
class NaveEspacial {
//Atributos de instancia
private int velocidad;
private int combustible;


public NaveEspacial clone(){
  return new NaveEspacial(velocidad,
combustible);
}
```

La consulta **`clone`** retorna un nuevo objeto con el mismo estado interno que el objeto que recibió el mensaje.

CASO DE ESTUDIO: VIDEOJUEGO

Alien
<<Atributos de clase>> maxVidas:entero <<Atributos de instancia>> Nave: **NaveEspacial** vidas:entero antenas:entero manos:entero

```
class Alien {
//Atributos de clase
private static final int
maxVidas = 5;
//Atributos de instancia
private NaveEspacial Nave;
private int vidas;
private int antenas;
private int manos;
}
```

Las clases **`Alien`** y **`NaveEspacial`** están **asociadas**.

La clase **`Alien`** tiene un atributo de clase **`NaveEspacial`**

CASO DE ESTUDIO: VIDEOJUEGO

Alien
<<Constructores>> Alien (n:NaveEspacial, v,a,m:entero)

```
//Constructor
public Alien (NaveEspacial n,
                  int v, int a,
                  int m){
// Requiere n ligado y v <=5
  Nave = n;
  vidas = v;
  antenas = a;
  manos = m;
}
```

Cuando se crea un objeto de clase **`Alien`** queda **asociado** a un objeto de clase **`NaveEspacial`**.

La clase cliente de **`Alien`** tiene la responsabilidad de asegurar que el parámetro real está ligado.

CASO DE ESTUDIO: VIDEOJUEGO

Alien
<<Comandos>> recuperaVidas() recibeHerida() establecerNave(n:NaveEspacial) establecerAntenas(p:entero) establecerManos(p:entero) establecerVidas(p:entero)

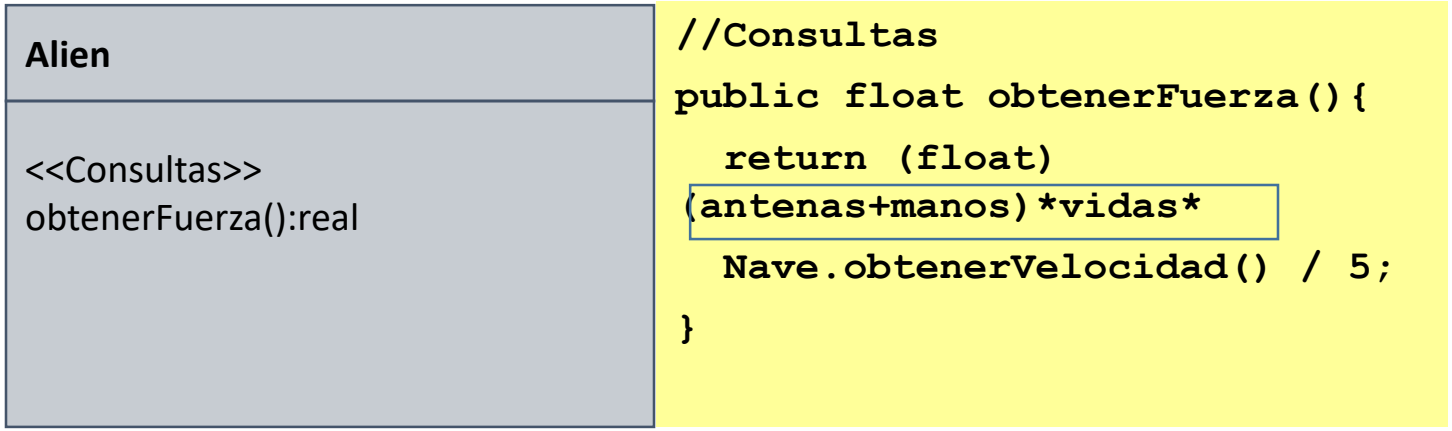
```
//Comandos
public void recuperaVidas(){
  if (vidas+2 > maxVidas)
    vidas = maxVidas;
  else
    vidas = vidas+2;  }

public void recibeHerida(){
  if (vida > 0) vida--;
}
```

CASO DE ESTUDIO: VIDEOJUEGO

Alien
<<Consultas>> obtenerNave():NaveEspacial obtenerVidas():entero obtenerAntenas():entero obtenerManos():entero

```
//Consultas
public NaveEspacial
        obtenerNave(){
  return Nave;
}
public int obtenerVidas(){
  return vidas;
}
public int obtenerAntenas(){
  return antenas;
}
public int obtenerManos (){
  return manos;
}
```

CASO DE ESTUDIO: VIDEOJUEGO



La consulta **obtenerFuerza** de la clase **Alien** envía el mensaje **obtenerVelocidad** a un objeto de la clase asociada **Nave**.

La clase **Alien** asume que el atributo de instancia **Nave** está ligada, ya que este es un compromiso asumido por las clases que la usan.

CASO DE ESTUDIO: VIDEOJUEGO

```
public void copy (Alien a){
//Requiere a ligada
   Nave = a.obtenerNave();
   vidas = a.obtenerVidas();
   antenas= a.obtenerAntenas();
   manos = a.obtenerManos();
}
```

Copy superficial

El comando **copy** asigna al alien que recibe el mensaje la misma cantidad de vidas, antenas y manos que el alien que pasa como parámetro y **lo asocia también a la misma nave**.

CASO DE ESTUDIO: VIDEOJUEGO

```
public void copy (Alien a){
//Requiere a ligada
   Nave = a.obtenerNave();
   vidas = a.obtenerVidas();
   antenas= a.obtenerAntenas();
   manos = a.obtenerManos();
}
```

Se modifica la identidad del atributo de instancia **`Nave`**.

CASO DE ESTUDIO: VIDEOJUEGO

```
public Alien clone(){
    return new Alien(Nave,vidas,
                manos,antenas);
}
```

Clone superficial

La consulta **`clone`** crea y devuelve un alien con la misma cantidad de vidas, antenas y manos que el alien que recibe el mensaje y **asociado también a la misma nave**.

CASO DE ESTUDIO: VIDEOJUEGO

```
public boolean equals (Alien a){
//Requiere a ligada
return (Nave == a.obtenerNave()&&
    vidas == a.obtenerVidas()&&
    manos == a.obtenerManos()&&
    antenas == a.obtenerAntenas());
}
```

equals superficial

La consulta **`equals`** retorna true si el alien que recibe el mensaje y el que pasa como parámetro son **equivalentes**.

Dos aliens son equivalentes si tienen la misma cantidad de vidas, manos y antenas y ambos están asociados a la misma nave. Es decir los atributos de instancia **`Nave`** de los dos objetos de clase **`Alien`** tienen la misma **identidad**.

CASO DE ESTUDIO: VIDEOJUEGO

```
public boolean equals (Alien a){
//Requiere a ligada
return (Nave == a.obtenerNave()&&
    vidas == a.obtenerVidas()&&
    manos == a.obtenerManos()&&
    antenas == a.obtenerAntenas());
}
```

El operador relacional compara la identidad de la nave del alien que recibió el mensaje con la identidad de la nave del alien que recibe como parámetro.

CASO DE ESTUDIO: VIDEOJUEGO

```
class VideoJuego{
    public static void main (String s[]){
        NaveEspacial n1 = new NaveEspacial (100,45);
        NaveEspacial n2 = new NaveEspacial (100,65);
        Alien a1,a2,a3,a4;
        a1 = new Alien(n1,3,2,4);
        a4 = new Alien(n2,2,3,4);
        a2 = a1.clone();
        a3 = a1;
        a4.copy(a1);
        System.out.println(a1==a2);
        System.out.println(a1==a3);
        System.out.println(a1==a4);
        a1.recibeHerida();
        System.out.println(a1==a2);
        System.out.println(a1==a3);
        System.out.println(a1==a4);
    }
}
```

CASO DE ESTUDIO: VIDEOJUEGO

```
class VideoJuego{
    public static void main (String s[]){
      NaveEspacial n1 = new NaveEspacial (100,45);
      NaveEspacial n2 = new NaveEspacial (100,65);
      Alien a1,a2,a3,a4;
      a1 = new Alien(n1,3,2,4);
      a4 = new Alien(n2,2,3,4);
      a2 = a1.clone();
      a3 = a1;
      System.out.println(a1.equals((a2));
      System.out.println(a1.equals((a3));
      System.out.println(a1.equals((a4));
      a1.recibeHerida();
      a4.copy(a1);
      System.out.println(a1.equals((a2));
      System.out.println(a1.equals((a3));
      System.out.println(a1.equals((a4));
    }
}
```

CASO DE ESTUDIO: VIDEOJUEGO

```
public void copy(Alien a){
//Requiere a ligado y Nave ligado en los dos aliens
   Nave.copy(a.obtenerNave());
   vidas = a.obtenerVidas();
   antenas= a.obtenerAntenas();
   manos = a.obtenerManos();
}
```

Copy en profundidad

Copia en el alien que recibe el mensaje la cantidad de vidas, antenas y manos del alien que pasa como parámetro.

En la nave asociada al alien que recibe el mensaje, **copia el estado interno de la nave asociada** al alien que pasa como parámetro.

CASO DE ESTUDIO: VIDEOJUEGO

```
public void copy(Alien a){
//Requiere a ligado y Nave ligado en los dos aliens
   Nave.copy(a.obtenerNave());
   vidas = a.obtenerVidas();
   antenas= a.obtenerAntenas();
   manos = a.obtenerManos();
}
```

Observemos que **no cambia la identidad** del atributo de instancia **`Nave`** sino el estado interno del objeto ligado a la variable.

CASO DE ESTUDIO: VIDEOJUEGO

```
public Alien clone(){
   NaveEspacial n = Nave.clone();
   return new Alien (n,vidas,manos,antenas);
}
```

Clone en profundidad

Crea un alien con la misma cantidad de vidas, manos y antenas que el alien que recibe el mensaje y **lo asocia a una nueva nave**, equivalente a la nave del alien que recibe el mensaje **`clone`**.

CASO DE ESTUDIO: VIDEOJUEGO

```
public boolean equals (Alien a){
/*Requiere a ligada y Nave ligada en los dos
aliens*/
 return
  (Nave.equals(a.obtenerNave())&&
   vidas == a.obtenerVidas()&&
   manos == a.obtenerManos()&&
   antenas == a.obtenerAntenas());
}
```

Equals en profundidad

En este caso, se considera que dos aliens son equivalentes tienen la misma cantidad de vidas, manos y antenas y están asociados a naves equivalentes.

CASO DE ESTUDIO: VIDEOJUEGO

```
public void copy(Alien a){
//Requiere a ligado y Nave ligado en los dos aliens
   nave = a.obtenerNave();
   vidas = a.obtenerVidas();
   antenas= a.obtenerAntenas();
   manos = a.obtenerManos();
}
```

Copy superficial

```
public void copy(Alien a){
//Requiere a ligado y Nave ligado en los dos aliens
   Nave.copy(a.obtenerNave());
   vidas = a.obtenerVidas();
   antenas= a.obtenerAntenas();
   manos = a.obtenerManos();
}
```

Copy en profundidad

CASO DE ESTUDIO: VIDEOJUEGO

```
public Alien cloneS(){
    return new
Alien(nave,vidas,manos,ant
}
```

Clone superficial

```
public Alien cloneP(){
  NaveEspacial n = nave.clone();
  return new Alien(n,vidas,manos,antenas);
}
```

Clone en profundidad

78

CASO DE ESTUDIO: VIDEOJUEGO

```
public boolean equalsS (Alien a){
 return Nave == a.obtenerNave()&&
        vidas == a.obtenerVidas()&&
        manos == a.obtenerManos()&&
        antenas == a.obtenerAntenas());
}
```

equals superficial

```
public boolean equalsP (Alien a){
 return
Nave.equals(a.obtenerNave())&&
   vidas == a.obtenerVidas()&&
   manos == a.obtenerManos()&&
   antenas == a.obtenerAntenas());
}
```

equals en profundidad

CASO DE ESTUDIO: VIDEOJUEGO

En un videojuego algunos de los personajes son aliens. Los aliens tienen cierta cantidad de antenas y de manos, que determinan su capacidad sensora y su capacidad de lucha respectivamente.

*Cada alien tiene un nombre y una cantidad de vidas que **inicialmente es 5** y se van reduciendo cada vez que recibe una herida. Cuando está muerto ya no tienen efecto las heridas. Cuando un alien logra llegar a la base recupera 2 vidas, sin superar nunca el valor 5.*

Cada alien tiene una nave, cada nave tiene una velocidad y una cantidad de combustible en el tanque. Ambos atributos pueden aumentar o disminuir de acuerdo a un parámetro que puede ser positivo o negativo.

La fuerza de un alien se calcula como la capacidad sensora, más su capacidad de lucha, todo multiplicado por el número de vidas.

Alien

<<Atributos de clase>>
maxVidas:entero
<<Atributos de instancia>>
Nave: **NaveEspacial**
vidas:entero
antenas:entero
manos:entero

<<Constructores>>
Alien (n:NaveEspacial,a,m:entero)
<<Comandos>>
recuperaVidas()
recibeHerida()
copy(a:Alien)
<<Consultas>>
obtenerNave():NaveEspacial
obtenerVidas():entero
obtenerAntenas():entero
obtenerManos():entero
obtenerFuerza():real
clone():Alien
equals(a:Alien): boolean

```
class Alien {
//Atributos de clase
private static final int
maxVidas = 5;
//Atributos de instancia
private NaveEspacial Nave;
private int vidas;
private int antenas;
private int manos;
//Constructor
public Alien (NaveEspacial n,
                  int a, int
m){
   Nave = n;
   vidas = maxVidas;
   antenas = a;
   manos = m;
}
}
```

El cambio afecta a la implementación de la consulta **`clone()`**.

PROVEEDORES Y CLIENTES

En la programación orientada objetos, cada **objeto de software** creado en ejecución modela a un **objeto del problema** identificado en la etapa de diseño.

El **estado interno** de un objeto puede contener **referencias** a otros objetos, de modo que la asociación entre clases se modela en ejecución a través de referencias entre objetos.

Así, la modificación de los atributos de instancia de una clase, modifica la representación de los objetos de software de esa clase, no de las clases asociadas.

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

Una empresa de telefonía celular ofrece distintos planes a sus abonados.

*Un **plan** tiene:*

- *un código,*
- *un costo mensual y*
- *establece un tope para el número de mensajes de texto y un tope de créditos que los abonados consumen con sus llamadas a números dentro de la comunidad y a otros móviles fuera de la comunidad.*

*Una **línea** tiene:*

- *un número asociado,*
- ***un plan** y*
- *una cantidad de consumos a líneas dentro de la comunidad y a líneas móviles fuera de la comunidad.*

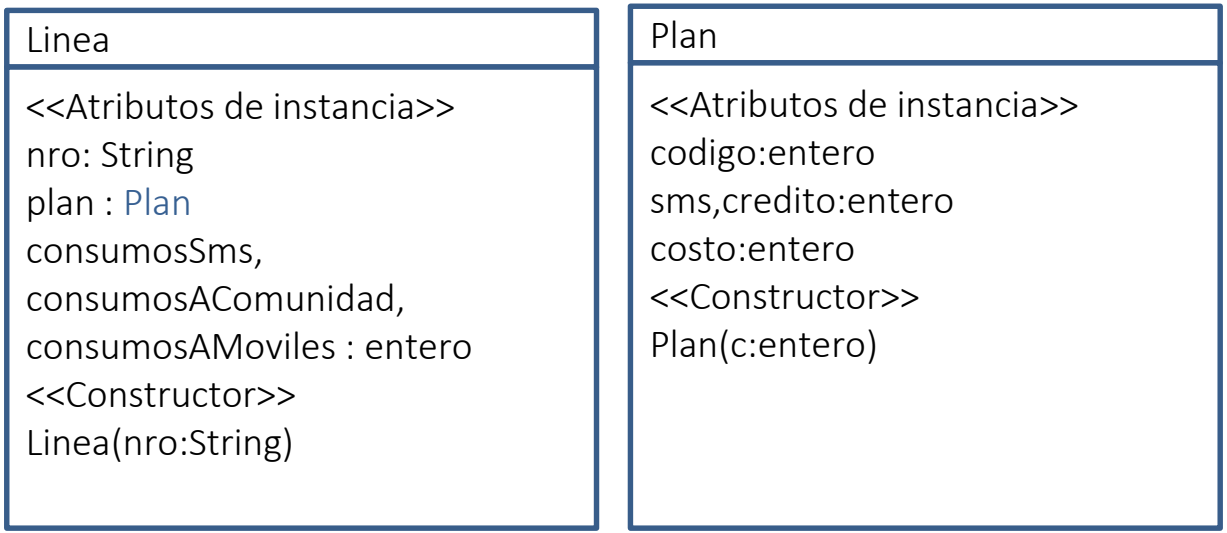
CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

La cantidad de créditos de un plan se consume de modo diferente según la llamada se realice dentro de la comunidad o fuera de ella:

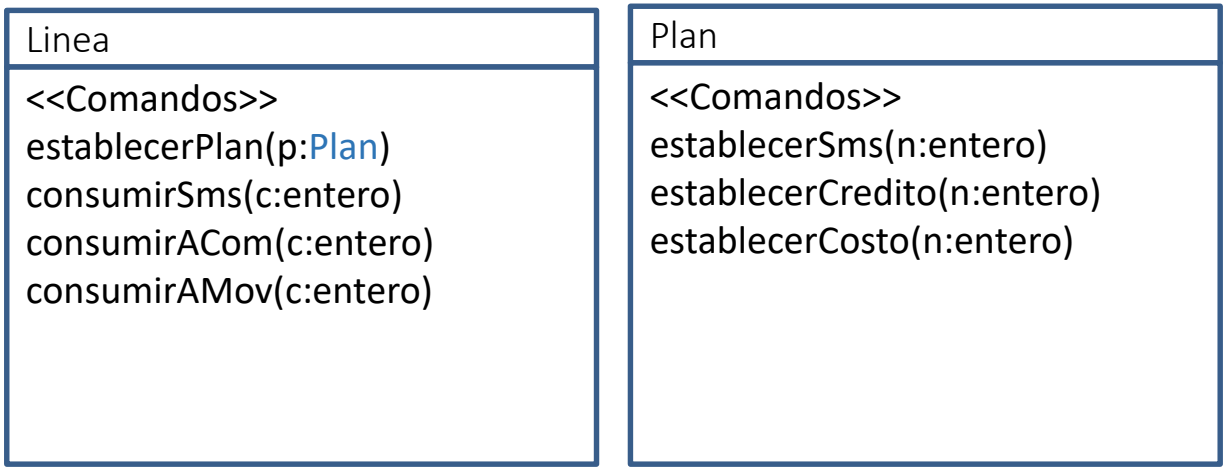
- *un minuto (o fracción) de llamada dentro de la comunidad consume 1 crédito,*
- *un minuto a una línea móvil fuera de la comunidad consume 2 créditos.*

Dos líneas se consideran equivalentes si tienen números equivalentes y los mismos valores en los demás atributos, en particular si están ligadas a un mismo plan.

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL



CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL



CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

Linea
<<Consultas>> obtenerNro():String obtenerPlan():Plan obtenerConsumosSms():entero obtenerConsumosAComunidad():entero obtenerConsumosAMoviles():entero consumoCredito():entero smsDisponibles():entero creditoDisponible():entero equals(l:Linea):boolean toString():String

Plan
<<Consultas>> obtenerSms():entero obtenerCredito():entero obtenerCosto():entero toString():String equals(p:Plan):boolean

equals(l:Linea):boolean
Si l es null retorna false

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

Linea
<<Responsablidades>> Requiere que se establezca el plan antes de consumir o ejecutar las consultas. Controla que no se consume más allá del crédito disponible Requiere que el valor consumido es siempre mayor a 0

Toda la entrada y salida se implementa fuera de la clase.

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

Cada objeto de clase **`Linea` tiene un** atributo de instancia de clase **`Plan`**.

El atributo de instancia **`plan`** no es visible desde el exterior de la clase **`Linea`**.

En ejecución este atributo mantiene una *referencia* a un objeto de clase **`Plan`**.

En la realidad a modelar, probablemente varias líneas correspondan a un mismo plan, en ejecución varias instancias de **`Linea`** referenciarán a un mismo objeto de clase **`Plan`**.

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
class Linea {
    /*Requiere que
    se  establezca el plan antes de consumir o ejecutar las consultas
    se controle que no se consume más allá del crédito disponible.*/

    //Atributos de Instancia
    private String nro;
    private Plan plan ;
    private int consumosSMS;
    private int consumosAComunidad;
    private int consumosAMoviles;

    //Constructor
    public Linea (String n){
        nro = n;
    }
}
```

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
//Comandos
public void establecerPlan(Plan p){
    //Requiere p ligado
    plan = p;
}

/*Requiere que se controle que no se consume más allá del crédito disponible.*/
public void consumirSMS(int n){
    if (consumoSMS+n < plan.obtenerSMS() )
        consumosSMS =+ n;
    else consumoSMS = plan.obtenerSMS();
}
```

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
public void consumirAComunidad(int n){
   if (consumoAComunidad+n < plan.obtenerCredito())
      consumosAComunidad =+ n;
   else
     consumosAComunidad= plan.obtenerCredito() - consumosAMoviles;
}


public void consumirAMoviles(int n){
   if (consumoAMoviles+2*n < plan.obtenerCredito())
      consumosAComunidad =+ 2*n;
   else
     consumosAMoviles= plan.obtenerCredito() - consumosAComunidad;
}
```

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
//Consultas
public String obtenerNro(){
  return nro;
}

public Plan obtenerPlan(){
  return plan;
}

public int obtenerConsumosSMS(){
  return consumosSMS;
}

public int obtenerConsumosAComunidad(){
  return consumosAComunidad;
}

public int obtenerConsumosAMoviles(){
  return consumosAMoviles;
}
```

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
public int consumoCredito(){
   return consumosAComunidad + consumosAMoviles*2;
}
public int smsDisponibles(){
//Requiere el plan ligado
   return plan.obtenerSMS() - consumosSMS;
}
public int creditoDisponible(){
//Requiere el plan ligado
   return plan.obtenerCredito() - consumoCredito();
}
```

La clase **Linea** es cliente de la clase **Plan**, **usa** los servicios provistos por **Plan**.

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
public String toString(){
//Requiere el plan ligado
  return nro + " " + consumoSMS + " " +
         consumoAComunidad+ " " + consumoAMoviles+
         " " + plan.toString();
}
public boolean equals(Linea l){
boolean e=false;
if (l != null)
  e = nro.equals(l.obtenerNro()) &&
  consumoSMS == l.obtenerConsumoSMS()&&
  consumoAComunidad==l.obtenerConsumoAComunidad()&&
  consumoAMoviles == l.obtenerConsumoAMoviles() &&
  plan == l.obtenerPlan();
  return e;}
```

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

El método **`equals`** compara en profundidad el atributo de instancia **`nro`** y en forma superficial el atributo de instancia **`plan`**.

Esto es, para que dos líneas sean equivalentes los atributos **`nro`** tienen que ser **equivalentes** y los atributos **`plan`** tienen que tener la misma **identidad**.

Observemos que en este caso no se asume que el parámetro está ligado.

Si la variable **`l`** no está ligada el **`equals`** retorna **`false`**.

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
class ventas{
public static void main(String[] a){
  Plan p1 = new Plan(123);
  p1.establecerSms(100);
  p1.establecerCredito(200);
  p1.establecerCosto(150);
  Plan p2 = new Plan(124);
  p2.establecerSms(200);
  p2.establecerCredito(300);
  p2.establecerCosto(200);
  …
```

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
class ventas{
public static void main(String[] a){
…
   String tel11 = "2916324567";
   String tel12 = "2916324568";
   String tel21 = "2912585234";

   Linea lin11 = new Linea(tel11);
   lin11.establecerPlan(p1);
   Linea lin12 = new Linea(tel12);
   lin12.establecerPlan(p1);
   Linea lin21 = new Linea(tel21);
   lin21.establecerPlan(p1);
   …
```

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

Una asignación al atributo de instancia **`plan`**, cambia el valor de la variable, la referencia a un objeto, pero no el estado interno del objeto mismo de clase **`Plan`**.

En particular el método **`establecerPlan(Plan p)`** en la clase **`Linea`** modifica el atributo de instancia **`plan`**, esto es el valor de la variable.

La clase **`Linea`** asume que cuando un objeto reciba el mensaje **`creditoDisponible()`** su atributo de instancia **`plan`** estará ligado.

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

```
class ventas{
public static void main(String [] a){
…
   System.out.println(lin11.toString());
   p1 = p2;
   System.out.println(lin11.toString());
   lin11.establecerPlan(p2);
   System.out.println(lin11.toString());
   p2.establecerCosto(220);
   System.out.println(lin11.toString());
…
```

CASO DE ESTUDIO: PLAN DE TELÉFONO MÓVIL

Cada **línea** telefónica está representada en ejecución por un **objeto de software** que en su estado interno mantiene los valores de los atributos que la caracterizan.

Cada **plan** también está representado por un único **objeto de software**, independientemente de cuántas líneas corresponden a ese plan.

Cada objeto de software de clase **`Linea`** mantiene una referencia a un objeto de software de clase **`Plan`**.

Todos los objetos de software que modelen líneas con un mismo plan, mantendrán referencias a un mismo objeto de clase **`Plan`**.

PROVEEDORES Y CLIENTES

Cuando dos clases están relacionadas una asume el rol de **CLIENTE** y otra asume el rol de **PROVEEDORA**.

Entre la clase **CLIENTE** y la clase **PROVEEDORA** se establece un **contrato** a través del cual cada una asume algunas **responsabilidades**.

Cada servicio provisto por la clase **PROVEEDORA** va a ser **usado** desde una clase **CLIENTE,** de acuerdo a las condiciones que establece el **contrato**.

Un mismo problema puede modelarse de maneras diferentes.

En cada diseño alternativo la asignación de responsabilidades puede variar.

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

*En un negocio se desea mantener información referida a las **facturas** imputadas a los vendedores.*

*De cada **factura** se mantiene:*

*el **número**,*

*el **monto** y*

*el **vendedor** que realizó la venta.*

*De cada **vendedor** se mantiene:*

*el **nombre** del vendedor y*

*las **ventas acumuladas**.*

Cuando se registra una factura, las ventas acumuladas del vendedor aumentan de acuerdo al monto de la factura.

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Factura
<<atributos de instancia>> nroFact: String montoFact: real vendedor: **Vendedor**
<< Constructores>> Factura(nro: String, m: float, ven: Vendedor) ...

Vendedor
<<atributos de instancia>> nombre: String vtaAcum: real
<<Constructor>> Vendedor(n: String) …

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Factura

<<atributos de instancia>>

nroFact: String
montoFact: real
vendedor: **Vendedor**

...

<<Comandos>>

establecerVendedor(ven:Vendedor)
establecerMontoFact(m: real)

<<Consultas>>

obtenerNroFact(): String
obtenerVendedor(): Vendedor
obtenerMontoFact(): real

Vendedor

<<atributos de instancia>>

nombre: String
vtaAcum: real

...

<<Comandos>>

establecerVtaAcum(m: real)
actualizarVtaAcum(m: real)

<<Consultas>>

obtenerNombre(): String
obtenerVtaAcum(): real

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Las clases **`Factura`**, **`String`** y **`Vendedor`** están asociadas, la relación es de tipo **tieneUn**.

Un comando de la clase **`Factura`** puede recibir como parámetro una variable de una clase asociada, como por ejemplo **`Vendedor`**.

Una consulta puede retornar una referencia a un objeto de alguna de las clases asociadas.

En este caso se crea también una dependencia, consecuencia de la asociación entre clases.

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Cuando se emite una factura la modificación de las ventas acumuladas del vendedor puede hacerse a través de un mensaje enviado:

- desde un servicio provisto por la clase **`Factura`** que registra la venta, como en la primera solución que propondremos.
- desde el método que invoca al servicio de la clase **`Factura`** que registra la venta, como en la segunda solución que propondremos.

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

- El diseñador del sistema establece la **responsabilidad** de cada clase.
- El implementador debe generar código adecuado para garantizar que cada clase cumple con sus responsabilidades.

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Alternativa 1

Factura

<<atributos de instancia>>

nroFact: String
montoFact: real
Vendedor: **Vendedor**

...

<<Responsabilidades>>

Cuando se crea una factura se actualiza las ventas acumuladas del vendedor. Requiere que nro y ven sean referencias ligadas.

Vendedor

<<atributos de instancia>>
nombre: String

vtaAcum: real

...

<<Responsabilidades>>

Requiere que nom sea una referencia ligada.

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Alternativa 2

Factura

<<atributos de instancia>>

nroFact: String
montoFact: real
vendedor: **Vendedor**

...

<<Responsabilidades>>

Requiere que se modifiquen las ventas acumuladas del vendedor consistentemente. Requiere que nro y ven sean referencias ligadas.

Vendedor

<<atributos de instancia>>
nombre: String

vtaAcum: real

...

<<Responsabilidades>>

Requiere que nom sea una referencia ligada.

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Alternativa 1

```
public  Factura(String n, float m, Vendedor ven){
/*Crea una factura, guarda número, monto y vendedor,
y actualiza la vtaAcum con el mismo monto. Requiere
que n y ven estén ligadas */

 nroFact = n;
 montoFact = m;
 vendedor = ven;
 vendedor.actualizarVtaAcum(m);
}
```

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Alternativa 1

```
class Ventas {
…
String num1 = new String("A-0001");
String num2 = new String("A-0002");


Vendedor v = new Vendedor("Gomez");
…
Factura f1 = new Factura (num1, 1500, v);
Factura f2 = new Factura (num2, 1200, v);

…
}
```

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Alternativa 2

```
public  Factura  (String n, float m, Vendedor ven){
/* Requiere que se actualicen las ventas acumuladas
del vendedor consistetemente */

 nroFact = n;
 montoFact = m;
 vendedor = ven;
}
```

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Alternativa 2

```
class Ventas {
…
String num1 = new String("A-0001");
String num2 = new String("A-0002");

Vendedor v = new Vendedor("Gomez");
…
Factura f1 = new Factura (num1, 1500, v);
v.actualizarVtaAcum(1500);
Factura f2 = new Factura (num2, 1200, v);
v.actualizarVtaAcum(1200);

…
}
```

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

```
class Vendedor{
//Atributos de Instancia
private String nombre;
private float vtaAcum;
//Constructores
public Vendedor(String nom){
//Requiere nom ligada
  nombre = nom;
}
//Comandos
public void actualizarVtaAcum(float s) {
  vtaAcum += s;}
…
}
```

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

La clase **`Vendedor`** se implementa de la misma manera, sea cual sea la alternativa de diseño elegida para **`Factura`**.

Si el diseñador elige una de las alternativas y cambia de decisión una vez que las clases están implementadas, el cambio va a requerir modificar tanto la clase proveedora, como todas las clases que la usan.

CASO DE ESTUDIO: FACTURA Y CUENTA CORRIENTE

Una modificación de diseño que cambia las responsabilidades afecta a la colección de clases asociadas.

Si solo se modifica una de las clases, por ejemplo la clase Vendedor, va a producirse un **error de aplicación**, que pasa desapercibido para el compilador.

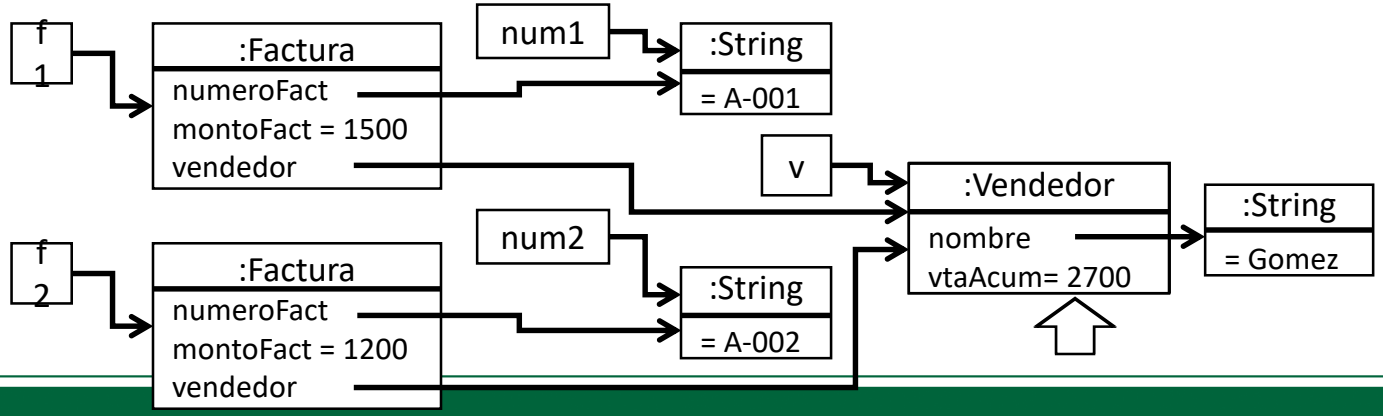
REPRESENTACIÓN EN MEMORIA

- Una variable de tipo clase mantiene una referencia a un objeto de software.
- El atributo de instancia `vendedor` mantiene una referencia a un objeto de clase `Vendedor`.
- Todos los objetos de software que modelen facturas emitidas para un mismo Vendedor, estarán ligados a una misma cuenta corriente, esto es, a un mismo o objeto de clase Vendedor.

REPRESENTACIÓN EN MEMORIA

Representación por Referencia (ALTERNATIVA 2)

```
String num1 = new String("A-0001");
String num2 = new String("A-0002");
CtaCte v = new Vendedor ("Gomez");
…
Factura f1 = new Factura (num1,1500,v);
v.actualizarSaldo (1500);
Factura f2 = new Factura (num2,1200,v);
v.actualizarSaldo (1200);
```



REPRESENTACIÓN EN MEMORIA

Representación por Referencia

- Cada objeto de software modela a un objeto del problema identificado en la etapa de diseño.
- El estado interno de un objeto puede contener referencias a otros objetos, de modo que un sistema complejo puede modelarse a partir de objetos simples.
- La modificación de la representación de un objeto no afecta a la representación de los objetos que lo referencian.
- En este caso si cambia la representación interna de la clase **`Vendedor`**, la modificación no afecta a la clase **`Factura`**.

Identidad y Equivalencias entre clases asociadas

Termostato
<<Atributos de instancia>> panel,actual:entero
<<Constructor>> Termostato(p,a:entero) <<Comandos>> establecerPanel(p:entero) establecerActual(a:entero) copy (t:Termostato) <<Consultas>> obtenerPanel():entero obtenerActual ():entero regulado():boolean equals(t:Termostato):boolean clone():Termostato toString():String

En un sistema de automatización de viviendas se modelan diferentes tipos de dispositivos, uno de los más simples es un termostato.

establecerPanel(p:entero)

El parámetro lo ingresó el usuario

establecerActual(a:entero)

El parámetro fue leido de un sensor

regulado():boolean

El termostato está regulado si la última temperatura sensada es la que estableció el usuario en el panel

Identidad y Equivalencias entre clases asociadas

```
class Termostato{
//Atributos de instancia
private int panel;
private int actual;
//Constructor
public Termostato(int p,int a){
  panel = p; actual = a;
}
```

Identidad y Equivalencias entre clases asociadas

```
//Comandos
public void establecerPanel(int p){
/*La establece el usuario a través del panel*/
  panel = p;
}
public void establecerActual(int p){
//La lee un sensor
  actual= p;
}
```

```
public void copy (Termostato t){
  panel = t.obtenerPanel();
  actual =  t.obtenerActual();
}
```

Identidad y Equivalencias entre clases asociadas

```
//Consultas
public int obtenerPanel(){
 return panel;
}
public int obtenerActual(){
 return actual;
}
public boolean regulado(){
/*El termostato está regulado si la última temperatura
sensada es la que estableció el usuario en el panel*/
  return panel == actual;
}
```

Identidad y Equivalencias entre clases asociadas

```
public Termostato clone(){
  return new Termostato (panel,actual);
}
        public boolean equals (Termostato t){
          return panel == t.obtenerPanel() &&
                  actual == t.obtenerActual();
        }
```

Identidad y Equivalencias entre clases asociadas

Termostato
<<Atributos de instancia>> panel,actual:entero
<<Constructor>> Termostato(p,a:entero) <<Comandos>> establecerPanel(p:entero) establecerActual(a:entero) copy (t:Termostato) <<Consultas>> obtenerPanel():entero obtenerActual ():entero regulado():boolean equals(t:Termostato):boolean clone():Termostato toString():String

Termotanque
<<Atributos de instancia>> mechero:boolean capacidad:entero termostato:Termostato
<<Constructor>> Termotanque(t:Termostato, c:entero) <<Comandos>> encender() apagar() establecerTermostato(t:Termostato) establecerCapacidad(c:entero) copy (m:Termotanque)

Identidad y Equivalencias entre clases asociadas

Termostato
<<Atributos de instancia>> panel,actual:entero
<<Constructor>> Termostato(p,a:entero) <<Comandos>> establecerPanel(p:entero) establecerActual(a:entero) copy (t:Termostato) <<Consultas>> obtenerPanel():entero obtenerActual ():entero regulado():boolean equals(t:Termostato):boolean clone():Termostato toString():String

Termotanque
<<Atributos de instancia>> mechero:boolean capacidad:entero termostato:Termostato
… <<Consultas>> encendido():boolean obtenerCapacidad():entero obtenerTermostato():Termostato equals(t:Termotanque):boolean clone():Termotanque toString():String

Identidad y Equivalencias entre clases asociadas

```
class Termostato{
//Atributos de instancia
private int panel;
private int actual;
//Constructor
public Termostato(int p,int a){
  panel = p; actual = a;}
```

```
class TermoTanque{
//Atributos de instancia
private boolean mechero;
private int capacidad;
private Termostato termostato;
//Constructor
public TermoTanque (Termostato t , int c){
//Requiere t ligado
  mechero = false; capacidad=c; termostato = t;
}
```

Identidad y Equivalencias entre clases asociadas

```
//Comandos
public void encender(){
  mechero = true;
}
public void apagar(){
  mechero = false;
}
public void establecerCapacidad(int c){
  capacidad = c;
}
public void establecerTermostato(Termostato t){
//Requiere t ligado
  termostato = t;
}
```

Identidad y Equivalencias entre clases asociadas

```
//Consultas
public boolean encendido(){
  return mechero;
}
public int obtenerCapacidad{
  return capacidad;
}
public Termostato obtenerTermostato{
  return termostato;
}
```

Identidad y Equivalencias entre clases asociadas

En este caso de estudio la clase **`Termotanque`** no brinda un método **`obtenerMechero()`** tenemos que tenerlo en cuenta al implementar **`equals`** y **`copy`**

Además, el constructor inicializa uno de los atributos en un valor constante, debemos tenerlo en cuenta cuando implementamos **`clone.`**

Identidad y Equivalencias entre clases asociadas

- El comando **copy** modifica el estado interno del objeto que recibe el mensaje con el estado interno del objeto que pasa como parámetro.

- En la **copia superficial** los dos termotanques quedan asociados a un mismo termostato. La referencia al termostato del termotanque que pasa como parámetro, se asigna al atributo de instancia **`termostato`**, del termotanque que recibe el mensaje.

- En la **copia profundidad** los dos termotanques quedan asociados a distintos termostatos, equivalentes entre sí. El estado interno del termostato del termotanque que pasa como parámetro, se copia en el estado interno del termostato asociado al termotanque que recibe el mensaje.

Identidad y Equivalencias entre clases asociadas

```
public void copy (TermoTanque t ){
//Requiere t ligado
  if (t.encendido()) mechero= true;
  else mechero = false;
  capacidad = t.obtenerCapacidad();
  termostato =  t.obtenerTermostato();
}
```

Superficial

```
public void copy (TermoTanque t ){
//Requiere t ligado
  if (t.encendido()) mechero= true;
  else mechero = false;
  capacidad = t.obtenerCapacidad();
  termostato.copy(t.obtenerTermostato());
}
```

Profundidad

Identidad y Equivalencias entre clases asociadas

La consulta **equals** compara el estado interno del objeto que recibe el mensaje con el estado interno del objeto que pasa como parámetro.

La **igualdad superficial** computa true si los dos termotanques tienen el mismo valor en los atributos mechero y capacidad, y están asociados a un mismo termostato, esto es la referencias son iguales.

La **igualdad en profundidad** computa true si los dos termotanques tienen el mismo valor en los atributos mechero y capacidad, y están asociados a termostatos equivalentes.

Identidad y Equivalencias entre clases asociadas

```
public boolean equals (TermoTanque t){
//Requiere t ligado
  return (mechero && t.encendido() ||
         !mechero  && !t.encendido()) &&
         capacidad == t.obtenerCapacidad() &&
         termostato == t.obtenerTermostato();
}
```

Superficial

```
public boolean equals (TermoTanque t){
//Requiere t ligado
  return (mechero && t.encendido() ||
         !mechero  && !t.encendido()) &&
          capacidad == t.obtenerCapacidad() &&
        termostato.equals(t.obtenerTermostato());
}
```

Profundidad

Identidad y Equivalencias entre clases asociadas

La consulta **clone** crea y retorna como resultado un objeto con el mismo estado interno que el objeto que recibe el mensaje.

En la **clonación superficial** los dos termotanques quedan asociados a un mismo termostato.

En la **clonación en profundidad** el nuevo termotanque queda asociado a un nuevo termostato, que es un clon del termostato del termotanque que recibe el mensaje. Así, los termostatos de los dos termotanques, tienen distinta identidad pero son equivalentes.

Identidad y Equivalencias entre clases asociadas

```
public TermoTanque clone(){
  Termotanque t = new TermoTanque(termostato,
                          capacidad);
  if (mechero)
    t.encender();
  return t;
}
```

```
    public TermoTanque clone(){
     Termotanque t = new TermoTanque(termostato.clone(),
                                                  capacidad);
      if (mechero)
        t.encender();
      return t;
    }
```

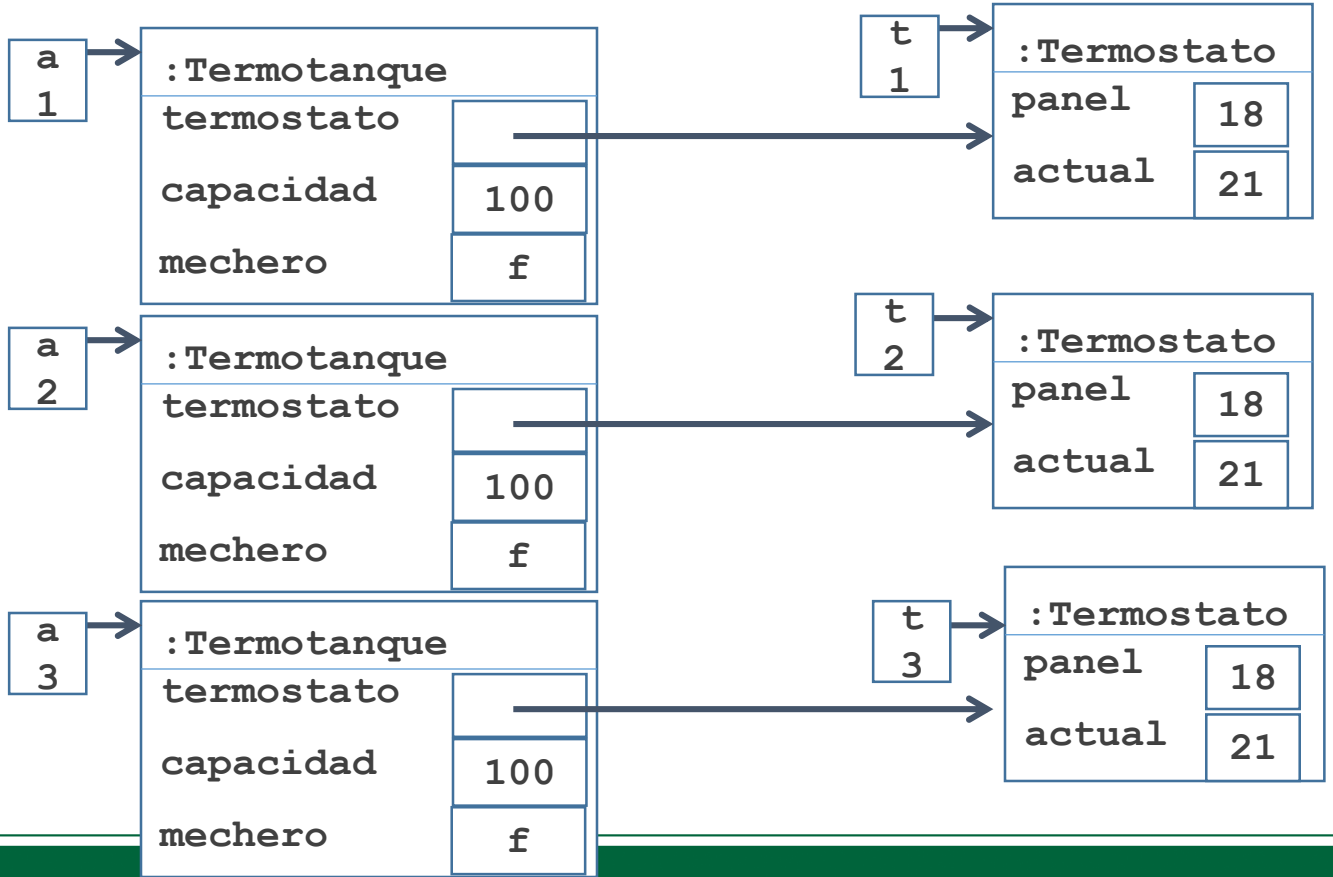
Identidad y Equivalencias entre clases asociadas

```
public String toString(){
   String estado;
   if (mechero) estado = "encendido";
   else estado = "apagado";
   return estado+" "+capacidad+ " "
              +termostato.toString();
}
```

Identidad y Equivalencias entre clases asociadas

```
Termostato t1 = new Termostato (18,21);
Termostato t2 = new Termostato (18,21);
Termostato t3 = t1.clone();

TermoTanque a1 = new TermoTanque(t1,100);
TermoTanque a2 = new TermoTanque(t2,100);
TermoTanque a3 = new TermoTanque(t3,100);

boolean b1 = a1.equals(a2);
boolean b2 = a1.equals(a3);
```

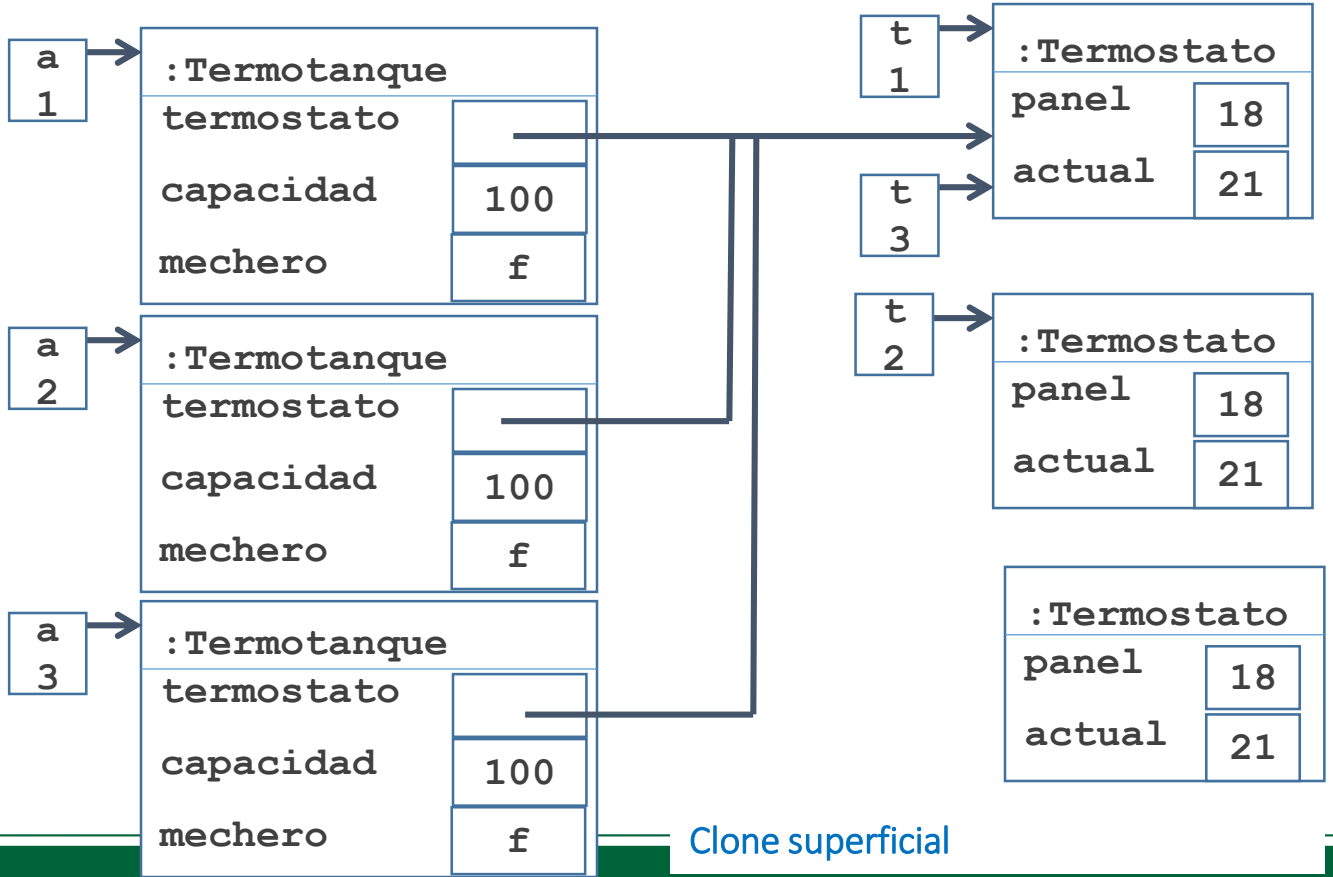
Identidad y Equivalencias entre clases asociadas



Identidad y Equivalencias entre clases asociadas

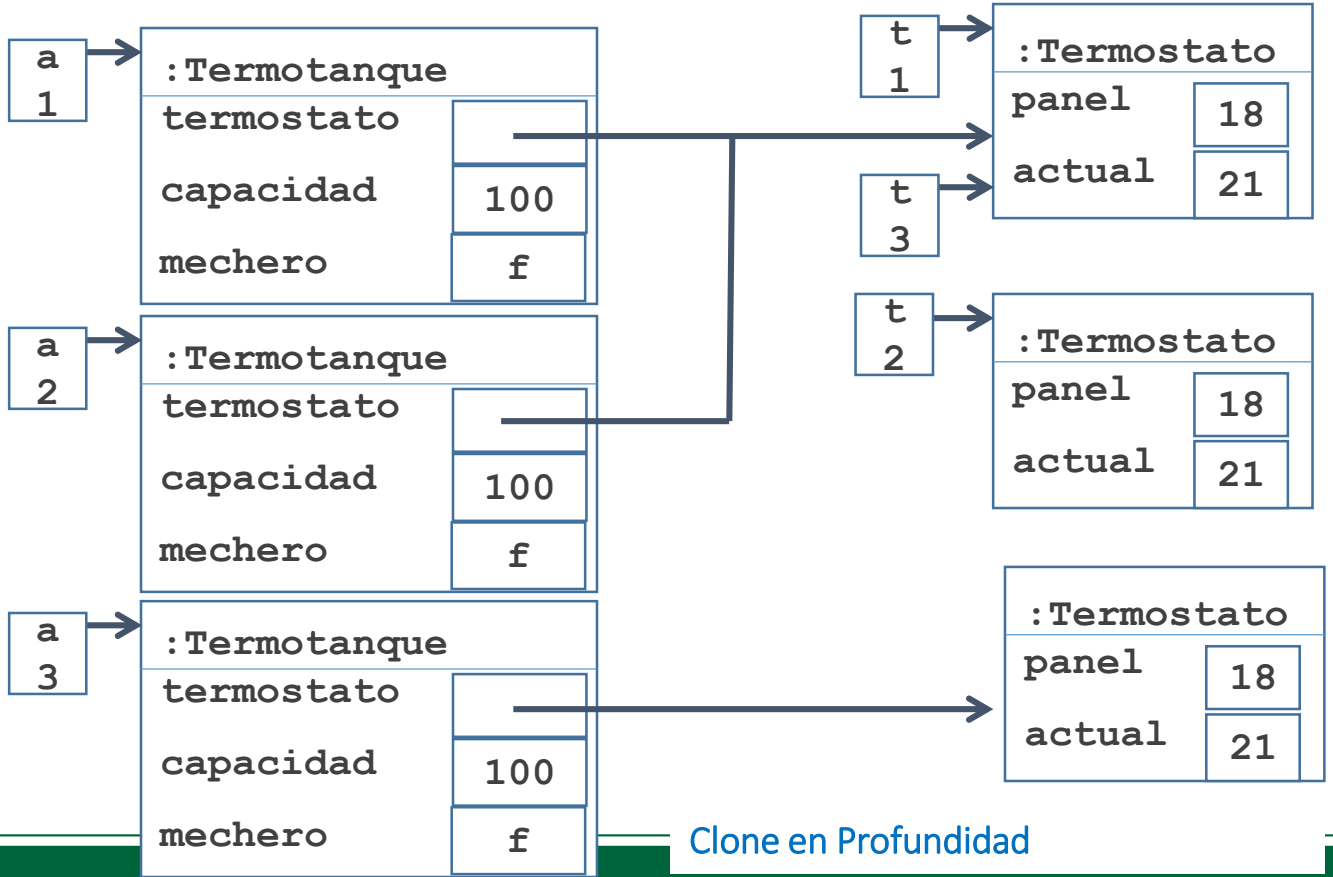
```
Termostato t1 = new Termostato (18,21);
Termostato t2 = new Termostato (18,21);
Termostato t3 = t1;

TermoTanque a1 = new TermoTanque(t1,100);
TermoTanque a2 = new TermoTanque(t3,100);
TermoTanque a3 = a1.clone();

boolean b1 = a1.equals(a2);
boolean b2 = a1.equals(a3);
```

Identidad y Equivalencias entre clases asociadas



Clone superficial

Identidad y Equivalencias entre clases asociadas



Clone en Profundidad

MULTIPLICIDAD EN LAS RELACIONES

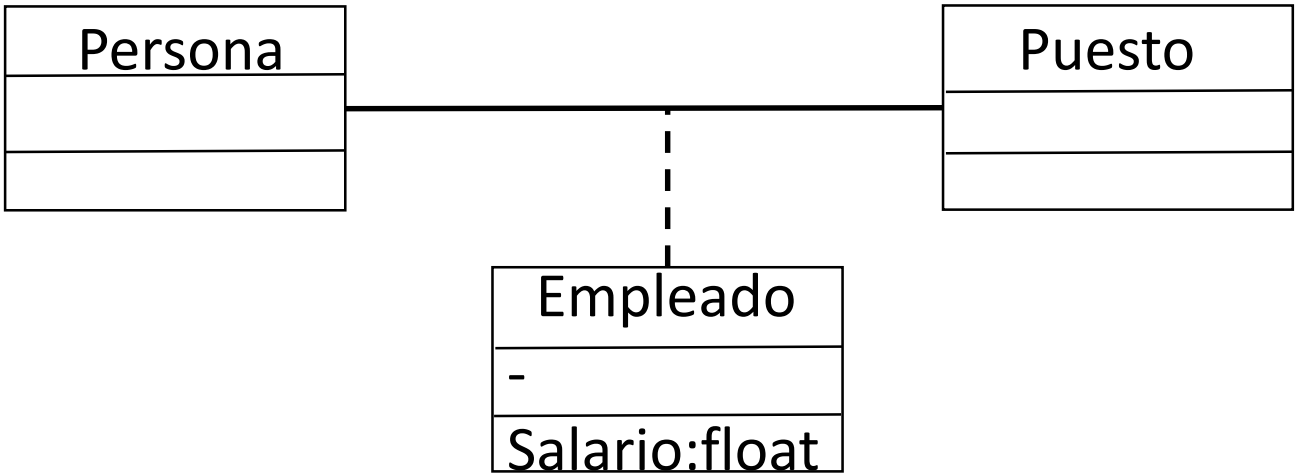
- Representa la cantidad de objetos de una clase que se relacionan con un objeto de la clase asociada. Puede ser:

- uno a muchos: 1..* (1..n)
- 0 a muchos: 0..* (0..n)
- x a y (enteros) (x .. y)
- m (entero) (m denota el entero).
- m , n (enteros) (m ó n)

CLASE ASOCIACION

- Cuando una clase se conecta a una asociación se denomina clase asociación.
- La clase asociación no se conecta a ninguno de los extremos de la asociación, sino que se conecta a la asociación real, a través de una línea punteada.
- Se utiliza para añadir información extra en un enlace.

- Clase asociación Empleado:



AGREGACION

- La agregación es un tipo de asociación que indica que una clase es parte de otra clase.
- Una de las clases juega un papel importante dentro de la relación con las otras clases.
- Permite la representación de relaciones como «maestro y esclavo», «todo y parte de» o «compuesto y componentes»

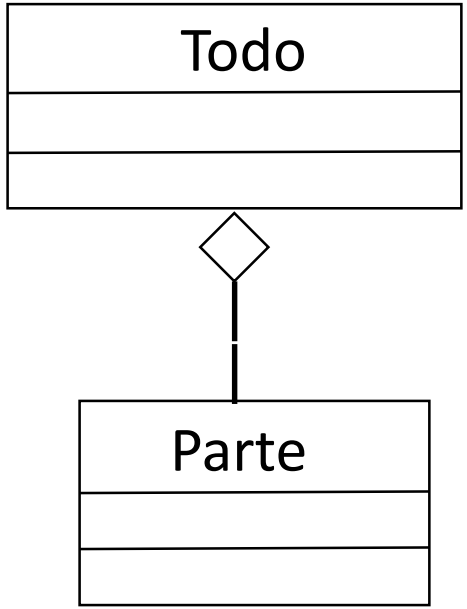
AGREGACION

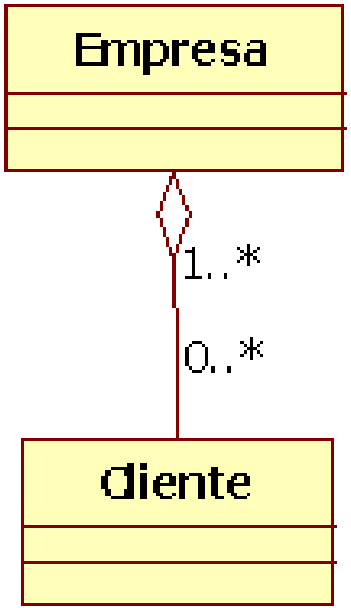
Esta relación también es conocida como *“forma parte de …”*

por ejemplo si se tiene la clase: computadora, esta tiene un monitor, que es otra clase; el monitor forma parte de la computadora; por lo tanto tienen una relación de agregación.

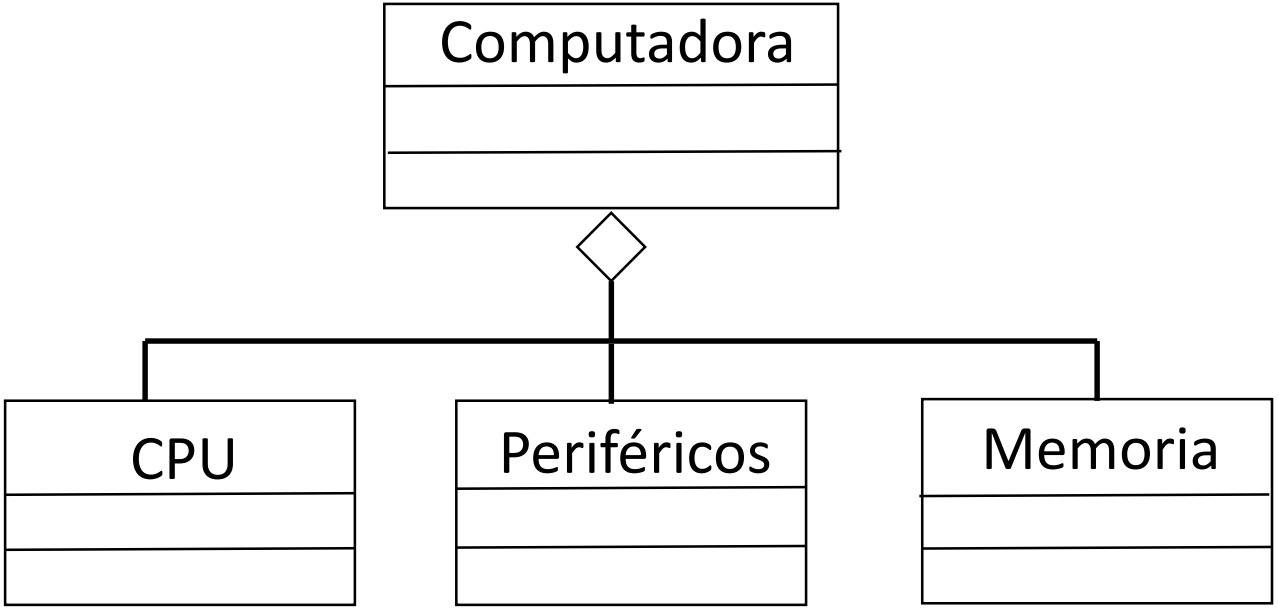
- Se representa con un rombo a continuación de la clase que representa el **todo** «propietaria» y una línea recta que apunta a la clase que representa la **parte** «poseída».
- Esta relación se conoce como «tiene un» ya que el todo tiene sus partes; una clase es parte de otra clase.

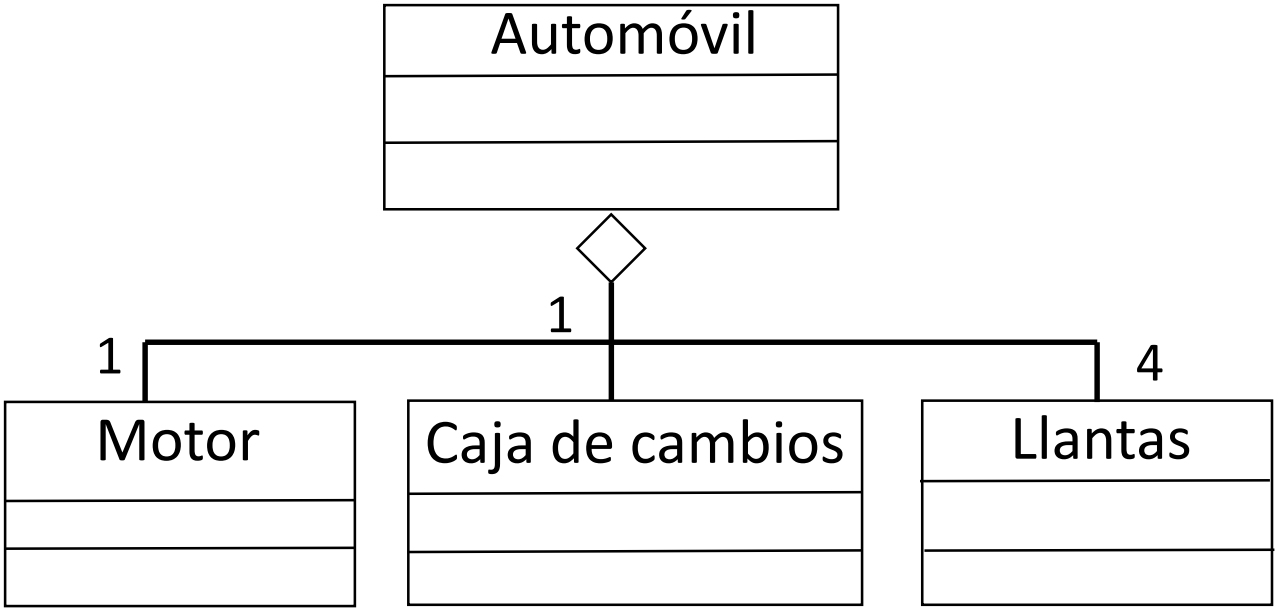
La destrucción del todo **no** conlleva la destrucción de las partes

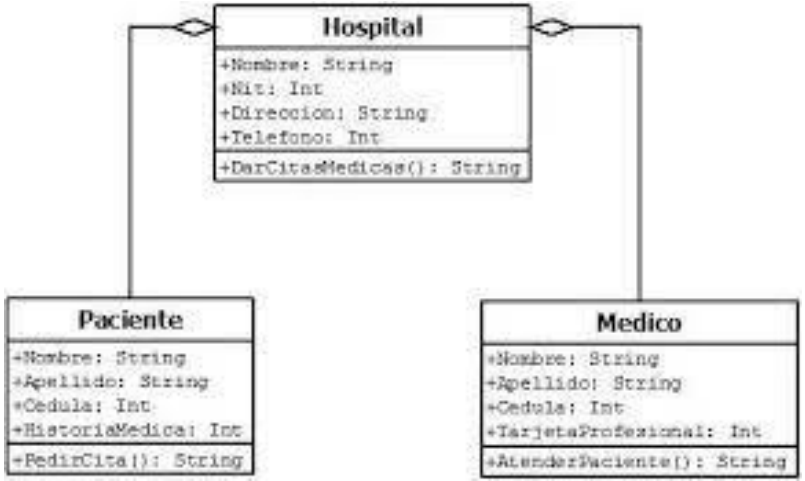



Empresa
1..*
0..*
Cliente

EJEMPLO



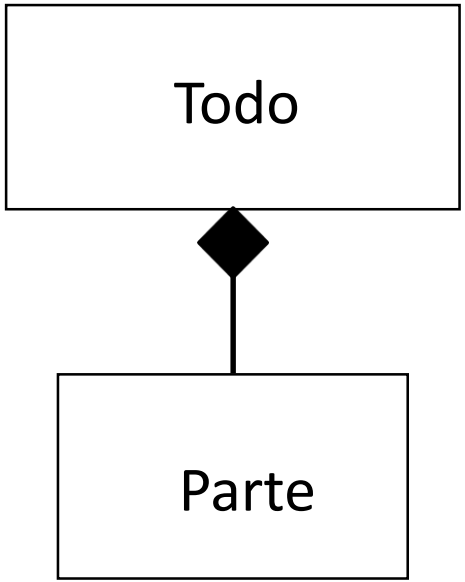

Automóvil
1
1
4
Motor
Caja de cambios
Llantas


Hospital
+Nombre: String
+Nit: Int
+Direccion: String
+Telefono: Int
+DarCitasMedicas(): String
Paciente
+Nombre: String
+Apellido: String
+Cedula: Int
+HistoriaMedica: Int
+PedirCita(): String
Medico
+Nombre: String
+Apellido: String
+Cedula: Int
+TarjetaProfesional: Int
+AtenderPaciente(): String

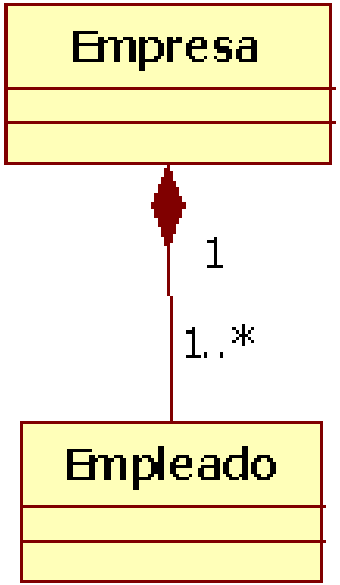
COMPOSICION

- Es un tipo especial de agregación que impone algunas restricciones: si el objeto completo se copia o se borra (elimina), sus partes se copian o se suprimen con el.
- Una composición es un tipo de relación entre clases que indica que una clase contiene (o está compuesta por) objetos de otras clases.

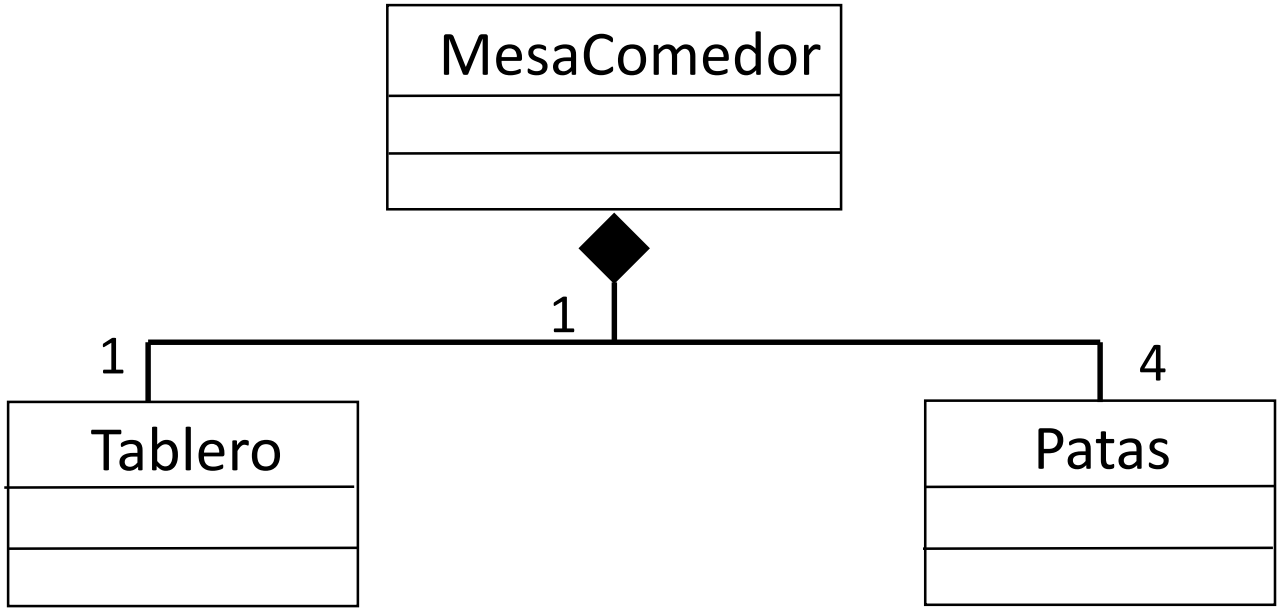
- Representa una relación fuerte entre clases.
- A diferencia de la agregación, la composición indica que la relación entre los objetos es de tipo “parte/todo”.
- Se representa por un rombo, igual que la agregación, excepto que el rombo está relleno.

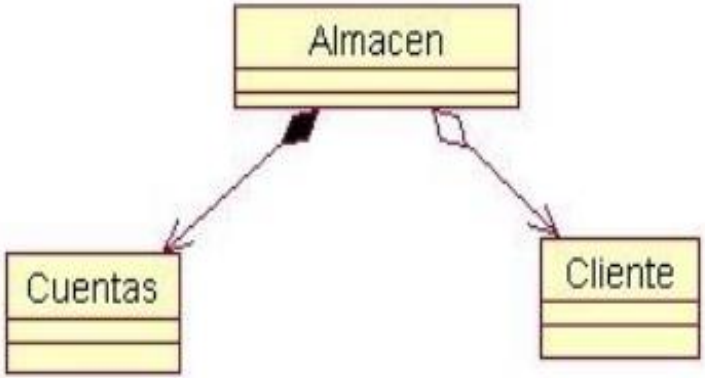


La supresión del todo **conlleva** la supresión de la parte.


Empresa
1
1..*
Empleado

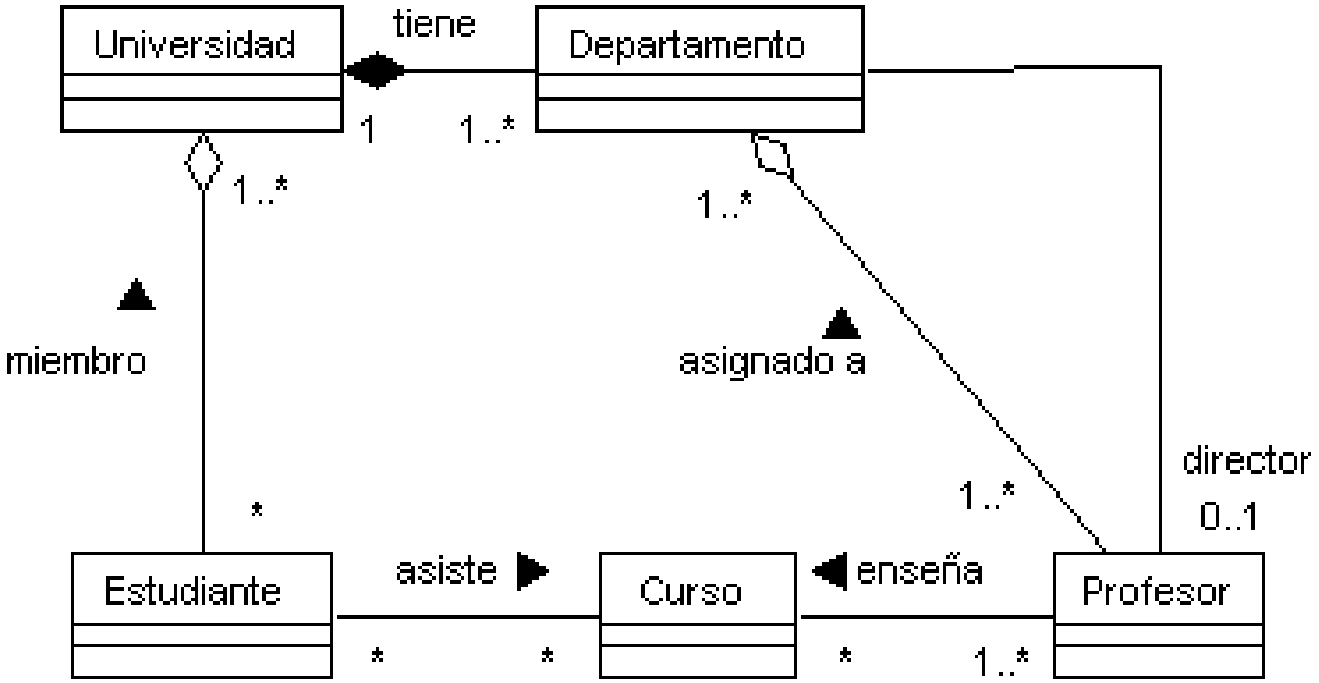
EJEMPLO




Almacen
Cuentas
Cliente

- Aquí las flechas tienen punta ya que están inclinadas.
- El rombo vacío indica una relación de agregación (siendo la clase Almacén “el todo” y la clase cliente representa “la parte”).
- El rombo relleno indica una relación de composición (Si desaparece el almacén, desaparecen las cuentas).

- Un Almacén posee Clientes y Cuentas. Los rombos van en la clase que está formada por las otras: el todo y las partes.
- Cuando se destruye la clase Almacén también es destruida la clase Cuenta asociada, en cambio no es afectada la clase Cliente asociada.


Universidad
tiene
Departamento
1
1..*
1..*
1..*
miembro
asignado a
1..*
director
0..1
*
Estudiante
asiste
Curso
enseña
Profesor
*
*
*
1..*