Dependencia y Asociación entre Clases

Los procesos de abstracción y clasificación permiten identificar, representar y agrupar a entidades que son semejantes de acuerdo a algún criterio. El criterio adoptado en una clasificación permite decidir si una entidad **pertenece** a una clase o no.

En la construcción de un modelo para una aerolínea seguramente se agrupen en una clase los vuelos que se realizan, en otra las ciudades a las que llegan los vuelos, otra clase estará conformada por los pilotos habilitados para comandar una nave y otra por las tarjetas de embarque.

Las entidades de una clase son semejantes de acuerdo al criterio elegido y además se **relacionan** de la misma manera con las entidades de otras clases. Una forma de relación es la **asociación** y se produce cuando el modelo de un objeto del problema **contiene** o **puede contener** al modelo de otro objeto del problema. Por ejemplo, todo vuelo **tiene** una ciudad como destino final y **puede tener** una o más ciudades en las que hace escala.

Otra forma de relación es la **dependencia** y se produce cuando el modelo de un objeto del problema **usa** al modelo de otro objeto. Por ejemplo, un pasajero usa su tarjeta de embarque para despachar su equipaje.

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 asociado a otro objeto, **tiene un** atributo de su clase. Por ejemplo, un objeto de la clase *Vuelo* **tiene un** atributo *destino* de la clase *Ciudad*. La relación entre los objetos provoca una relación entre las clases, que se dicen **asociadas**.

Un objeto depende de un objeto de otra clase, si **usa un** atributo de esta clase. Por ejemplo, un objeto de la clase *Pasajero* **usa un** atributo de la clase *TarjetaEmbarque*. La relación entre los objetos provoca una relación de dependencia entre las clases.

Cuando una clase está asociada a otra clase, los cambios en la segunda pueden tener un impacto en la primera. El mismo impacto se produce si se modifica una clase de la cual depende otra. Uno de los objetivos de la programación orientada a objetos es reducir el impacto de estos cambios.

Dependencia entre clases

Cuando una clase declara una variable local o un parámetro de otra clase, decimos que existe una **dependencia** entre la primera y la segunda y surge de una relación del tipo **usaUn** entre los objetos.

Caso de Estudio: Fábrica de Juguetes

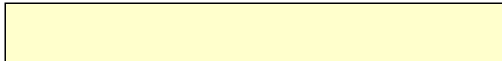
*En una fábrica de autos de juguete una parte de la producción la realizan **robots**. Cada robot tiene una carga de energía que se va consumiendo a medida que ejecuta las órdenes que recibe. Cada robot es capaz de conectarse de modo tal que se recargue su energía hasta su capacidad máxima de 5000 unidades. Esta acción puede ejecutarse ante una orden externa o puede iniciarla el robot mismo cuando su energía está por debajo de las 100 unidades.*

*Cada robot cuenta con piezas de diferentes tipos: ruedas, ópticas y chasis. 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. Es responsabilidad de cada servicio que consuma energía recargarla cuando corresponda.*

```
Robot
---------------------------------
<<atributos de clase>>
energiaMaxima : 5000
energiaMinima : 100
<<atributos de instancia>>
nroSerie:entero
energia: entero
ruedas: entero
opticas: entero
chasis: entero
---------------------------------
<<constructor>>
Robot(nro:entero,caja:Caja)
<<comandos>>
abrirCaja (caja: Caja)
recargar()
armarAuto()
<<consultas>>
obtenerEnergia (): entero
obtenerChasis () : entero
obtenerRuedas () : entero
obtenerOpticas () : entero
cantAutos() : entero
clone():Robot
equals(Robot r):boolean
---------------------------------
<<Responsabilidades>>
El constructor establece la
energía en el valor máximo y
las cantidades de piezas en
100.
Todos los servicios que
consumen energía deciden
recargar cuando energía es
menor que la mínima.
```

```
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)
vaciar()
<<consultas>>
obtenerChasis () : entero
obtenerRuedas () : entero
obtenerOpticas () : entero
equals(c:Caja):boolean
---------------------------------
```

```
Requiere que haya piezas
disponibles para armar un auto
```



`recargar()` recarga la energía del robot hasta llegar al máximo.

`abrirCaja (caja:Caja)` aumenta las piezas disponibles de acuerdo a las cantidades de la caja y la vacía. Requiere `caja` ligada.

`armarAuto()` decrementa las piezas disponibles, requiere que se 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.

En este caso, 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.

La implementación parcial de `Robot` será:

```
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;
//Constructor
public Robot (int nro,Caja caja){
  nroSerie = nro;
  energia=energiaMaxima;
  ruedas = caja.obtenerRuedas();
  opticas = caja.obtenerOpticas();
  chasis = caja.obtenerChasis();
  caja.vaciar();}
//Comandos
public void recargar(){
  energia=energiaMaxima;}
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(); }
public void abrirCaja (Caja caja) {
/*Aumenta sus cantidades según las de la caja y la vacía. Requiere
caja ligada*/
  ruedas += caja.obtenerRuedas();
  opticas += caja.obtenerOpticas();
  chasis += caja.obtenerChasis();
  energia -= 50;
```

```
  caja.vaciar();
/*Controla  si es necesario recargar energía*/
 if (energia < energiaMinima)
    this.recargar();}
//Consultas
public int obtenerRuedas(){
  return ruedas;}
public int obtenerOpticas(){
  return opticas;}
public int obtenerChasis(){
  return chasis;}
public int obtenerNroSerie(){
  return nroSerie;}
public int obtenerEnergia(){
  return energia;}
public String toString(){
  return nroSerie+" "+ruedas+" "+opticas+" "+chasis;}}
```

Entre las clases `Robot` y `Caja` se establece una relación de **dependencia**. Un parámetro de un servicio provisto por la clase `Robot`, es de tipo `Caja`.

El comando `abrirCaja(Caja caja)` provoca un **efecto colateral** sobre el parámetro. Es decir, no solo modifica el estado interno del robot que recibe el mensaje, sino también sobre la caja que pasa como parámetro. Si se modifica la signatura de los servicios provistos por la clase `Caja`, los cambios impactan en la clase `Robot`.
Ejercicio: Implemente la clase Caja y complete la clase Robot de acuerdo al diseño.

La clase `FabricaJuguetes` usa los servicios provistos por Robot:

```
class FabricaJuguetes{
…
public void producir() {
  Caja caja = new Caja(100,150,25);
  Robot unRobot;
  unRobot = new Robot(111,caja);
  …
  System.out.println(caja.obtenerRuedas());
  …
  if (unRobot.cantAutos() == 0)
    unRobot.abrirCaja(caja);

  unRobot.armarAuto();
  …}}
```

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

Por ejemplo, en `caja.obtenerRuedas()` el mensaje `obtenerRuedas()` lo recibe un objeto de clase `Caja`, de modo que se ejecuta el método provisto por esa clase. Si en el método `producir()` de la clase `FabricaJuguetes`, el objeto `unRobot` recibe el mensaje `obtenerRuedas()` se ejecuta el método provisto por la clase `Robot`, porque `unRobot` es de clase `Robot`.

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

Asociación entre clases

Cuando una clase **tiene un atributo** de otra clase, ambas clases están **asociadas** y la relación es de tipo **tieneUn**. En general, entre dos clases asociadas también hay una relación de dependencia, algunos de los servicios provistos por una clase recibirán parámetros o retornarán un resultado de la clase asociada. Así, una relación de tipo **tieneUn** en general provoca una relación de tipo **usaUn**.

Caso de Estudio: Signos Vitales

Los signos vitales son medidas de variaciones fisiológicas que permiten valorar las funciones corporales básicas. Dos de los principales signos vitales son la temperatura corporal y la presión arterial. Se considera que existe un principio de alarma cuando estos valores escapan de los umbrales establecidos. 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. La temperatura se mide en centígrados.

PresionArterial	Signos Vitales
<<atributos de clase>> umbralMax,umbralMin :real <<atributos de instancia>> maxima,minima :real	<<atributos de clase>> umbralTemp:real <<atributos de instancia>> temperatura: real presion :PresionArterial
<<Constructores>> PresionArterial(ma,mi:real) <<Consultas>> obtenerMaximaMM():real obtenerMinimaMM():real obtenerMaximaHP():real obtenerMinimaHP().real obtenerPresionPulsoMM():real obtenerPresionPulsoHP():real alarmaHipertension():boolean	<<Constructores>> SignosVitales(t:real, p:PresionArterial) <<Consultas>> obtenerTemperatura():real obtenerPresion():PresionArterial alarma ():boolean
Requiere máxima > mínima y ambos mayores a 0. Los valores están expresados en milímetros de mercurio	Requiere temperatura > 0, expresada en grados.

El **estado interno** de cualquier objeto de clase `PresionArterial` mantiene los valores de dos atributos de instancia, `maxima` y `minima`, representados en milímetros de mercurio. Los valores de `maxima` y `mínima` expresados en hectopascales no se mantienen en el estado interno, se computan cuando se invocan los métodos `obtenerMaximaHP()` y `obtenerMinimaHP()`.

La clase `PresionArterial` fue implementada en el capítulo anterior. La implementación de `SignosVitales` es:

```
class SignosVitales{
//Atributos de clase
private static final float umbralTemp=38.0;
//Atributos de instancia
private float temperatura;
private PresionArterial presion ;
//Constructor
```

```
public SignosVitales (float t, PresionArterial p){
//Requiere t > 0, expresada en grados
  temperatura = t;
  presion = p;}
//Consultas
public float obtenerTemperatura (){
  return temperatura;}
public PresionArterial obtenerPresion (){
  return presion ;}
public boolean alarma(){
  return temperatura>umbralTemp || presion.alarmaHipertension();}
}
```

Las clases `SignosVitales` y `PresionArterial` están asociadas. La clase `SignosVitales` **tieneUn** atributo de clase `PresionArterial`. Además, `SignosVitales` **usaA** `PresionArterial`, su constructor recibe un parámetro de clase `PresionArterial` y la consulta `obtenerPresion()` retorna un resultado de esa misma clase.

La clase `SignosVitales` puede acceder a cualquiera de los miembros públicos de la clase `PresionArterial`. Los atributos han sido declarados como privados, por lo tanto no son accesibles.

La clase `Control` usa los servicios provistos por `SignosVitales` y `PresionArterial`:

```
class Control{
…
public void controlDiario(int mi1,int ma1,
                           int mi2, int ma2,
                           float t1,float t2){
PresionArterial p6Hs, p12Hs;
SignosVitales sv6Hs,sv12Hs;
p6Hs = new PresionArterial(mi1,ma1);
p12Hs= new PresionArterial(ma1,ma2);
sv6Hs = new SignosVitales(t1,p6Hs);
sv12Hs = new SignosVitales(t2,p12Hs);
if (sv6Hs.obtenerTemperatura() ==sv12Hs.obtenerTemperatura())
  System.out.println("Temperatura Estable");}
}
```

Existe en este caso una relación de dependencia entre `Control` y las clases `SignosVitales` y `PresionArterial`.

Representación por referencia de clases asociadas

La representación a través de referencias surge para modelar la asociación entre clases, como se ilustra con el siguiente caso de estudio:

Caso de Estudio: Aliens y Naves

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 máximo de vidas,

esto es, 5. El comando establecerVidas requiere un parámetro menor o igual al máximo de vidas.

Cada alien está asociado a 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.

```
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
```

```
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()
establecerNave(n:NaveEspacial)
 copy(a:Alien)
<<Consultas>>
obtenerNave():NaveEspacial
obtenerVidas():entero
obtenerAntenas():entero
obtenerManos():entero
obtenerFuerza():entero
clone():Alien
```

Las clases `Alien` y `NaveEspacial` están asociadas, la clase `Alien` **tiene un** atributo de clase `NaveEspacial`. En el **problema**, dos o más aliens pueden estar asociados a una misma nave espacial. En la **solución**, dos o más objetos de clase `Alien` pueden mantener referencias a un mismo objeto de clase `Nave`. También es posible que dos naves tengan el mismo estado interno, pero distinta identidad.

La implementación de `NaveEspacial` es:

```
class NaveEspacial {
//Atributos de instancia
private int velocidad;
private int combustible;
//Constructor
public
  NaveEspacial(int v,int c){
  velocidad= v;
  combustible= c;}
//Comandos
```

```
public void cambiarVelocidad(int v){
  velocidad =+v;}
public void cambiarCombustible(int c){
  combustible =+c;}
public void copy (NaveEspacial n){
velocidad=n.obtenerVelocidad();
combustible=n.obtenerCombustible();}
//Consultas
public int obtenerVelocidad(){
  return velocidad;}
public int obtenerCombustible (){
  return combustible;}
public boolean equals (NaveEspacial n){
  return velocidad ==n.obtenerVelocidad()&&
        combustible==n.obtenerCombustible();}
public NaveEspacial clone(){
  return new NaveEspacial(velocidad, combustible);}}
```

La implementación de `Alien` de acuerdo al diseño propuesto:

```
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;}
//Comandos
public void establecerVidas(int v){
//Requiere v < maxVidas
  vidas = v;}
public void recuperaVidas(){
  if (vidas+2 > maxVidas)
    vidas = maxVidas;
  else
    vidas = vidas+2;}
public void recibeHerida(){
  if (vidas > 0) vidas--;}
public void copy (Alien a){
  Nave = a.obtenerNave();
  vidas = a.obtenerVidas();
  antenas = a.obtenerAntenas();
  manos = a.obtenerManos();}
public void establecerNave(NaveEspacial n){
Nave = n;}
//Consultas
public NaveEspacial obtenerNave(){
  return Nave;}
public int obtenerVidas(){
  return vidas;}
```

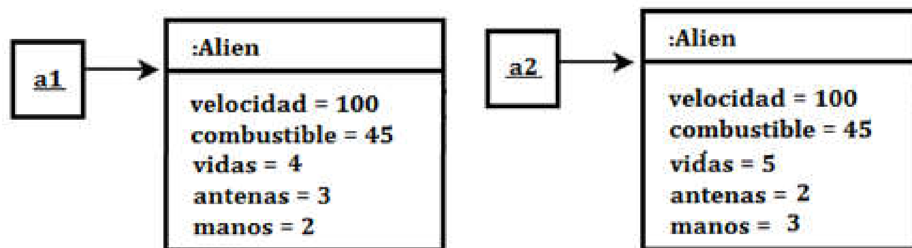
```
public int obtenerAntenas(){
  return antenas;}
public int obtenerManos (){
  return manos;}
public int obtenerFuerza(){
  return (antenas+manos)*vidas;}
public Alien clone(){
  NaveEspacial n = Nave;
  int m = manos;
  int a = antenas;
  int v = vidas;
  Alien c = new Alien(n,a,m);
  c.establecerVidas(v);
  return c;}
public boolean equals(Alien a){
  return Nave == a.obtenerNave() &&
         manos == a.obtenerManos() &&
         antenas == a.obtenerAntenas() &&
         vidas == a.obtenerVidas();}}
```

El constructor y el comando `establecerNave` de la clase `Alien` reciben como parámetro una variable de la clase asociada `NaveEspacial`. La consulta `obtenerNave` retorna una referencia a un objeto de la clase `NaveEspacial`. Se crea entonces una dependencia entre `Alien` y `NaveEspacial`, consecuencia de la asociación entre clases. Si la signatura de los métodos de la clase `NaveEspacial` cambia, la modificación puede afectar a la clase `Alien`.

A continuación *se asume* una **representación extendida** para objetos asociados y se analizan sus inconvenientes. Al ejecutarse el siguiente bloque de instrucciones del método `main`:

```
class testVideoJuego{
public static void main (String s[]){
  NaveEspacial n = new NaveEspacial(100,45);
  Alien a1,a2;
  a1 = new Alien(n,3,2);
  a2 = new Alien(n,2,3);
  a1.recibeHerida();
```

El diagrama de objetos para la representación extendida sería:



Es decir, el estado interno de un objeto de clase `Alien` mantiene los atributos de esa clase, más los atributos de la clase asociada `NaveEspacial`. Si dos o más objetos de la clase `Alien` están asociados a una misma instancia de la clase `NaveEspacial`, la estructura completa de cada objeto de la clase `NaveEspacial` se mantendrá en cada objeto de clase `Alien`.

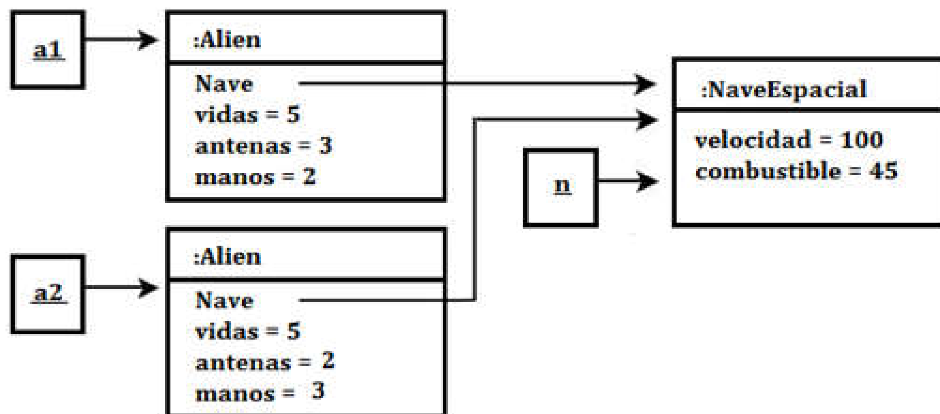
La representación extendida provoca varios inconvenientes. En primer lugar, es necesario agregar un atributo a cada nave que permita identificarla, por ejemplo un código. En caso contrario, si los estados internos de dos objetos de clase `Alien` coinciden, no es posible decidir si están ligados a la misma nave o a naves diferentes que coinciden en la velocidad y en la cantidad de combustible.

En segundo lugar, cada vez que se modifica el valor de uno de los atributos que caracterizan a un objeto, es necesario modificar el estado interno de todos los objetos a los cuales está asociado. En este caso de estudio, cada vez que cambie la `velocidad` de una nave se debería modificar el estado interno de todos los aliens asociados a esa nave.

Otro inconveniente importante de la representación extendida es que si se modifica la representación de la clase `NaveEspacial`, por ejemplo, porque se agrega un atributo de instancia `motores`, este cambio va a afectar a la representación de todas las clases que usan los servicios de `NaveEspacial`. Los cambios en una clase provocan un impacto en la representación de los objetos de las clases asociadas.

Estos inconvenientes se solucionan usando una **representación por referencia**. El estado interno de un objeto contiene referencias a los objetos de las clases asociadas, de modo que un sistema complejo puede modelarse a partir de objetos simples. La modificación de una clase, no afecta la representación de los objetos de las clases asociadas.

Luego de la ejecución del mismo bloque de instrucciones, el diagrama de objetos considerando una representación por referencia es:



En este caso, si cambia la estructura del estado interno de un objeto de clase `NaveEspacial`, por ejemplo, porque se agregan nuevos atributos, no cambia la estructura del estado interno de los objetos de clase `Alien`. Si se modifican los valores de los atributos de una nave, por ejemplo, porque se reduce el combustible, no se modifican los valores de los atributos de los aliens que referencian a esa nave.

Igualdad y equivalencia entre objetos de clases asociadas

La representación por referencia afecta al diseño y la implementación de algunos servicios, en particular a los métodos `equals`, `copy` y `clone`. Cuando una clase está asociada a otra, la igualdad, copia y clonación se puede hacer en forma **superficial** o en **profundidad**.

Para decidir si dos objetos de una clase son iguales en forma superficial, se evalúa si están ligados a una misma instancia de la clase asociada. De manera análoga, la copia superficial modifica el estado interno del objeto que recibe el mensaje, con los valores almacenados en el objeto que pasa como parámetro. La clonación superficial retorna como resultado un objeto que tiene exactamente el mismo estado interno que el objeto que recibió el mensaje, incluyendo las referencias a objetos de las clases asociadas.

Cuando la igualdad es en profundidad la exigencia es menor, se requiere que los estados internos de los objetos asociados sean equivalentes, aunque las referencias sean distintas. En el caso de la copia en profundidad, el estado interno del objeto que recibe el mensaje se modifica con los valores de los atributos del objeto que se recibe como parámetro, excepto las referencias, que no se modifican, pero sí se modifica el estado interno de los objetos asociados. La clonación en profundidad retorna un nuevo objeto con el mismo estado interno que el objeto que recibe el mensaje, excepto las referencias que estarán ligadas a clones de los objetos asociados.

Para ilustrar la diferencia, se retoma el análisis del caso de estudio que asocia a las clases `Alien` y `NaveEspacial`. Los siguientes métodos implementan entonces **copia, clonación e igualdad superficial**:

```
public void copy (Alien a){
//Requiere a ligada
  Nave = a.obtenerNave();
  vidas = a.obtenerVidas();
  antenas = a.obtenerAntenas();
  manos = a.obtenerManos();}
public Alien clone(){
  NaveEspacial n = Nave;
  int m = manos;
  int a = antenas;
  int v = vidas;
  Alien c = new Alien(n,a,m);
  c.establecerVidas(v);
  return c;}
public boolean equals(Alien a){
//Requiere a ligada
  return Nave == a.obtenerNave() &&
         manos == a.obtenerManos() &&
         antenas == a.obtenerAntenas() &&
         vidas == a.obtenerVidas();}
```

El comando `copy` copia el estado interno del objeto de clase `Alien` que se recibe como parámetro, en el estado interno del objeto de clase `Alien` que recibe el mensaje. Así, el valor del atributo `Nave` del objeto ligado al parámetro `a`, se asigna al atributo `Nave` del objeto que recibe el mensaje `copy`.

La consulta `clone` retorna un nuevo objeto de clase `Alien` que referencia a la misma nave que el objeto que recibió el mensaje. La siguiente implementación de `clone()` es equivalente a la anterior:

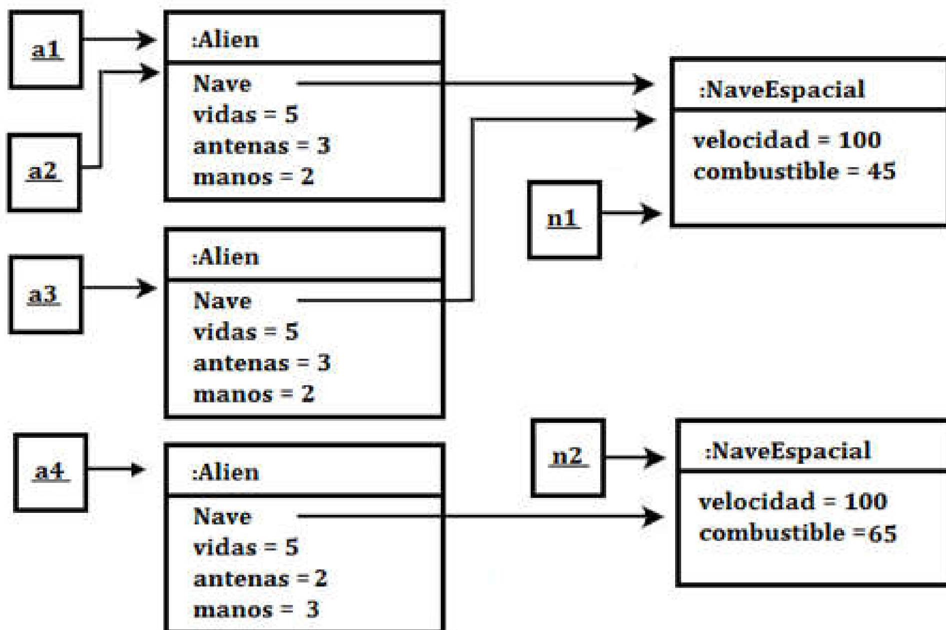
```
public Alien clone(){
  Alien c = new Alien(Nave,antenas,manos);
  c.establecerVidas(v);
  return c;}
```

La consulta `equals` retorna `true` si los valores de los atributos de los objetos que se comparan son iguales, esto implica que mantienen referencias a una misma nave.

Dada la siguiente clase tester:

```
class testVideoJuego{
public static void main (String s[]){
//Primer bloque
  NaveEspacial n1 = new NaveEspacial (100,45);
  NaveEspacial n2 = new NaveEspacial (100,65);
  Alien a1,a2,a3,a4;
  a1 = new Alien(n1,3,2);
  a2 = a1;
  a3 = a1.clone();
  a4 = new Alien(n2,2,3);
//Segundo bloque
  a4.copy(a1);
  System.out.println(a1==a2);
  System.out.println(a1==a3);
  System.out.println(a1==a4);
  System.out.println(a1.equals((a2)));
  System.out.println(a1.equals((a3)));
  System.out.println(a1.equals((a4)));
//Tercer bloque
  a1.recibeHerida();
  System.out.println(a1==a2);
  System.out.println(a1==a3);
  System.out.println(a1==a4);
  System.out.println(a1.equals((a2)));
  System.out.println(a1.equals((a3)));
  System.out.println(a1.equals((a4)));
//Cuarto bloque
  a1.recibeHerida();
  a4.copy(a1);
  System.out.println(a1.equals((a2)));
  System.out.println(a1.equals((a3)));
  System.out.println(a1.equals((a4)));
//Quinto bloque
NaveEspacial n3 = n1.clone();
Alien a5=a1.clone();}
}
```

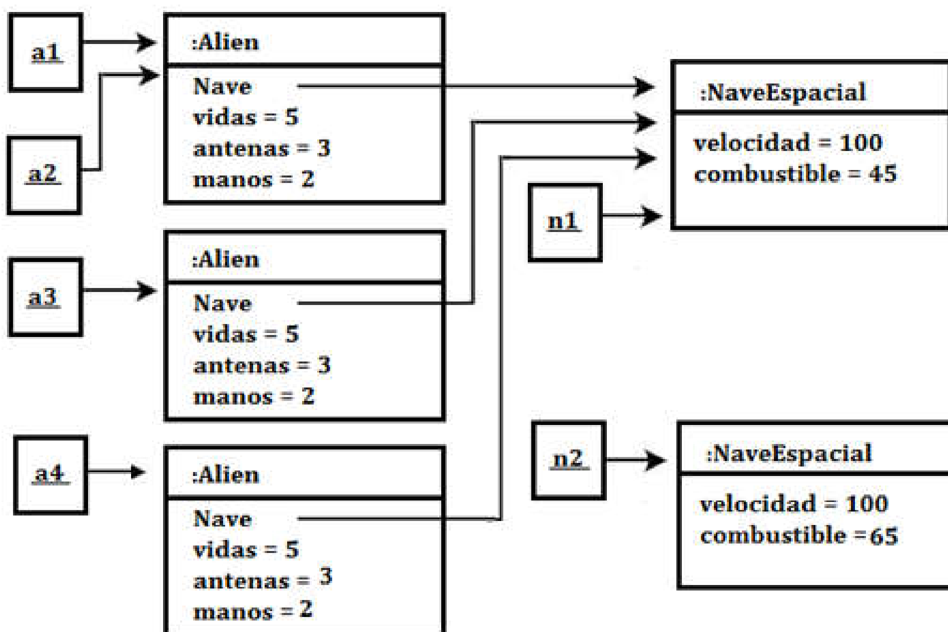
Después de la ejecución de las instrucciones que siguen al comentario “Primer bloque”, el diagrama de objetos es:



Cuando a continuación se ejecuta:

```
a4.copy(a1);
```

El diagrama de objetos se modifica como sigue:



De modo que el operador relacional en las instrucciones:

```
System.out.println(a1==a2);
System.out.println(a1==a3);
System.out.println(a1==a4);
```

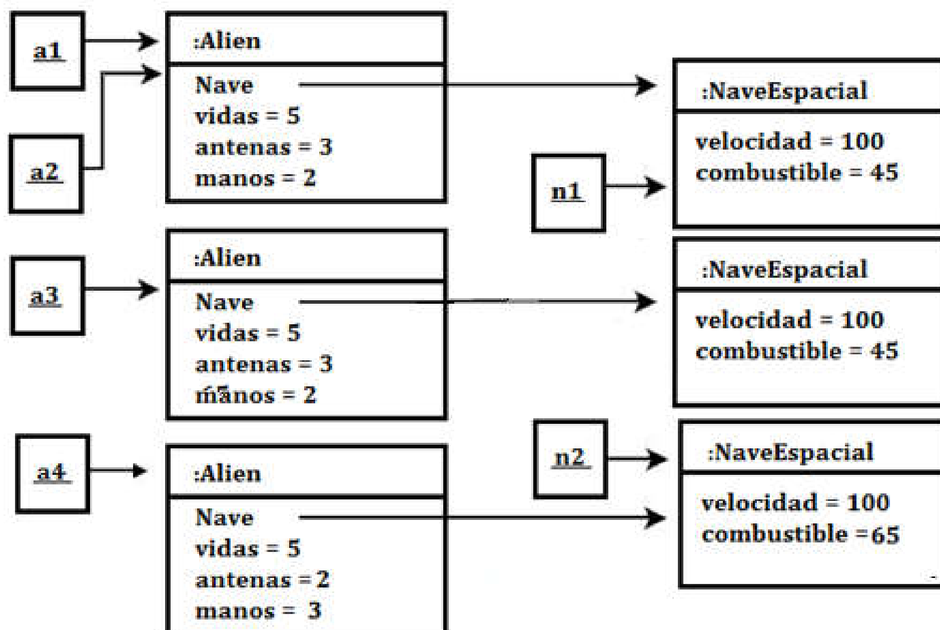
Computa `true`, `false` y `false`.

Ejercicio: Muestre la salida por consola del segmento completo de código y dibuje el diagrama de objetos inmediatamente antes de terminar la ejecución.

El diseñador del videojuego puede especificar que los métodos `copy`, `clone` y `equals` se implementen **en profundidad**. El código de **clone en profundidad** es:

```
public Alien clone(){
//Requiere Nave ligada
  NaveEspacial n = Nave.clone();
  int m = manos;
  int a = antenas;
  int v = vidas;
  Alien c = new Alien(n,a,m);
  c.establecerVidas(v);
  return c;}
```

En esta implementación, se crea un clone del alien que recibe el mensaje, vinculado a un clone de la nave asociada al alien que recibe el mensaje. Dada la misma clase tester, después de la ejecución del primer bloque, el diagrama de objetos es:



El objeto ligado a la variable `a3` mantiene una referencia a una nave que es un clone de la nave asociada al alien ligado a la variable `a1`.

La consulta `clone` definida en la clase `Alien` envía el mensaje `clone` al objeto ligado al atributo de instancia `Nave`. Las clases `Alien` y `NaveEspacial` brindan métodos con el mismo nombre. **Es la clase del objeto que recibe el mensaje la que determina cuál de los métodos debe ejecutarse** en cada caso**.**

Dada la secuencia de instrucciones del último bloque del código:

```
NaveEspacial n3 = n1.clone();
Alien a5 = a1.clone();
```

El primer mensaje `clone` se liga al método clone definido en la clase `NaveEspacial`. El segundo mensaje `clone` se liga al método clone definido en la clase `Alien`.

La implementación del comando **copy en profundidad** es:

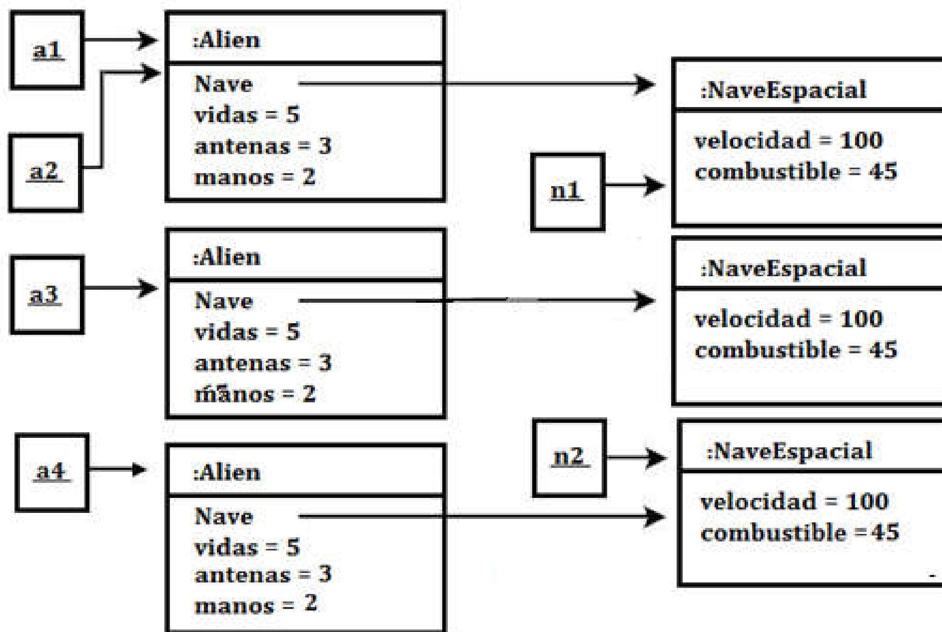
```
public void copy (Alien a){
//Requiere a ligada y la nave de a ligada
  Nave.copy(a.obtenerNave());
```

```
  vidas = a.obtenerVidas();
  antenas = a.obtenerAntenas();
  manos = a.obtenerManos();}
```

Después de la ejecución de la primera instrucción del segundo bloque:

```
  a4.copy(a1);
```

Se copia en el estado interno de la nave asociada al alien que recibe el mensaje, el estado interno de la nave del alien `a`, recibido como parámetro forma. El diagrama de objetos es:



Es decir, la copia en profundidad modifica los valores de los atributos `vidas`, `antenas` y `manos`, y el estado interno de la nave asociada al alien que recibe el mensaje, pero no se modifica el valor del atributo instancia `Nave` de la clase `Alien`.

La implementación de **equals en profundidad** es:

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

La consulta `equals` definida en la clase `Alien` envía el mensaje `equals` al objeto ligado al atributo de instancia `Nave`. Nuevamente, es la clase del objeto que recibe el mensaje la que determina cuál de los métodos debe ejecutarse en cada caso.

Para esta implementación de la consulta `equals` y considerando el último diagrama de objetos, las instrucciones:

```
  System.out.println(a1.equals((a3));
  System.out.println(a1.equals((a4));
```

Muestran en consola:

```
true
true
```

Es posible que el diseñador especifique en el diagrama que una clase debe brindar, por ejemplo, el método `equals` implementado en forma superficial y también en profundidad. En este caso es necesario elegir un nombre diferente para cada consulta.

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

En este caso la clase `Alien` brinda dos métodos alternativos para decidir si dos objetos son equivalentes. El primero, más estricto, decide que dos aliens son equivalentes si están asociados a una misma nave. El segundo, considera que dos aliens son equivalentes si están asociados a naves equivalentes.

Clientes y proveedores de servicios

En el conjunto de clases que modela una aplicación, algunas clases son **clientes** de los servicios provistos por otras clases **proveedoras**. Con frecuencia una misma clase puede cumplir ambos roles, es decir, ser cliente y proveedora a la vez.

Una clase que **usa** los servicios provistos por otra clase, es cliente de la clase que provee dichos servicios. Una clase que **tiene** atributos de otra clase, es cliente de las clases a las que corresponden a esos atributos; estas últimas son proveedoras de servicios.

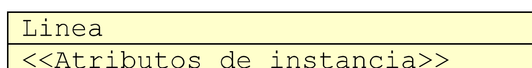
En particular una clase tester es una clase cliente de los servicios que brindan las clases que deben ser verificadas. Al implementar una clase tester se establece una dependencia entre esta clase y las que van a ser verificadas.

La clase cliente accede a la clase proveedora a través de su **interfaz**. La programación orientada a objetos propone **minimizar** la interfaz a través de la cual se comunican una clase cliente y una clase proveedora, de modo que se minimice también el impacto de los cambios.

Caso de Estudio: Plan 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. 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.

Las clases Linea y Plan están asociadas tal como modela el siguiente diagrama.



```
nro: String
plan : Plan
consumosSms,
consumosAComunidad,
consumosAMoviles : entero
-----------------------------
<<Constructor>>
Linea(nro:String)
<<Comandos>>
establecerPlan(p:Plan)
aumentarSms(c:entero)
aumentarACom(c:entero)
aumentarAMov(c:entero)
<<Consultas>>
obtenerNro():String
obtenerPlan():Plan
obtenerConsumosSms():entero
obtenerConsumosAComunidad():
entero
obtenerConsumosAMoviles():
entero
consumoCredito():entero
smsDisponibles():entero
creditoDisponible():entero
-----------------------------
Requiere que se  establezca el
plan antes de aumentar los
consumos o ejecutar las
consultas
```

```
codigo:entero
sms,credito:entero
costo:entero
-----------------------------
<<Constructor>>
Plan(c:entero)
<<Comandos>>
establecerSms(n:entero)
establecerCredito(n:entero)
establecerCosto(n:entero)
<<Consultas>>
obtenerSms():entero
obtenerCredito():entero
obtenerCosto():entero
-----------------------------
```

`consumoCredito()`: se computa como `consumosAComunidad+ consumosAMoviles*2.`
`smsDisponibles()`: se computa como el crédito que corresponde al plan, menos el consumo de crédito de la línea .

La implementación de `Linea` es:

```
class Linea {
//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;}
//Comandos
public void establecerPlan (Plan p){
  plan = p;}
public void aumentarSMS (int n){
 consumosSMS=+ n;}
public void aumentarAComunidad (int n){
 consumosSMS=+ n;}
public void aumentarAMoviles (int n){
 consumosSMS=+ n;}
//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 ;}
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();}
}
```

Cada objeto de clase `Linea` está asociado a una instancia de clase `Plan`. La clase `Linea` **tieneUn** atributo de clase **`Plan.`** La clase `Linea` también está asociada a la clase `String`.

Los métodos `establecerPlan` y `obtenerPlan` de la clase `Linea`, generan también una relación de dependencia entre `Plan` y `Linea`. La clase `Linea` es **cliente** de la clase **proveedora** `Plan`. La interfaz entre las clases `Linea` y `Plan` está formada por la signatura de los servicios públicos.

La clase cliente `Linea` puede implementarse conociendo únicamente la interfaz de la clase proveedora `Plan`. La clase `Plan` se implementa desconociendo quienes van a ser sus clientes.

El atributo `plan` en la clase `Linea` 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 entonces a un mismo objeto de clase `Plan`. La clase `Linea` asume que cuando un objeto reciba el mensaje `creditoDisponible()` su atributo de instancia `plan` estará ligado.
Ejercicio: Implemente la clase `Plan`*.*

Ejercicio: Implemente una clase `testTelefónica` *que* ***use*** *a las clases* `Linea`*,* `Plan` *y* `String`*, creando objetos de esa clase y enviándoles mensajes para verificar los servicios provistos por* `Linea`*.*

El contrato entre la clase Proveedora y la clase Cliente

Entre una clase cliente y una clase proveedora de servicios, se establece un **contrato** que determina las **responsabilidades** de cada una. Las condiciones del contrato se especifican en la etapa de **diseño** del sistema, en particular la **funcionalidad** que debe brindar cada servicio. En la **implementación** es necesario interpretar las responsabilidades y reflejarlas en el código. Parte de la **verificación** consiste en analizar si cada clase cumple con las responsabilidades establecidas en el contrato.

Cuando una clase no cumple con su contrato pueden producirse errores de ejecución o aplicación que es necesario detectar y depurar. También puede ocurrir que el contrato no se

haya especificado adecuadamente. En cualquier caso es fundamental diseñar una batería de casos de prueba que permitan detectar errores. El diseño de los casos de prueba puede hacerlo el diseñador del sistema o el responsable de la etapa de testeado.

Caso de Estudio: Facturas en Cuenta Corriente

*En un negocio se desea mantener información referida a las **facturas** imputadas a las cuentas corrientes de los **clientes**. De cada **factura** se mantiene el **número**, el **monto** y el **cliente**. De cada **cuenta corriente** se mantiene el **nombre** del cliente y el **saldo**. Cuando se crea un objeto de clase `Factura` el constructor inicializa todos los atributos de instancia de acuerdo a los parámetros y actualiza el saldo de la cuenta corriente del cliente con el monto de la factura.*

Factura
<<atributos de instancia>> nroFact : String montoFact : real clienteFact : Cta_Cte
<< Constructores>> Factura (nro:String,m:float,cli: Cta_Cte) <<Comandos>> establecerClienteFact(cli:Cta_Cte) establecerMontoFact(m:real) <<Consultas>> obtenerNroFact():String obtenerClienteFact():Cta_Cte obtenerMontoFact():real
<<Responsabilidades>> Requiere que nro y cli sean referencias ligadas.

Cta_Cte
<<atributos de instancia>> nombre : String saldo : float
<<Constructor>> Cta_Cte (nom:String) <<Comandos>> establecerSaldo(m:real) actualizarSaldo (m:real) <<Consultas>> obtenerNombre():String obtenerSaldo():real
<<Responsabilidades>> Requiere que nom sea una referencia ligada.

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

El comando `establecerClienteFact` de la clase `Factura` recibe como parámetro una variable de la `Cta_Cte`. La consulta `obtenerClienteFact` retorna una referencia a un objeto de la clase `Cta_Cte`. La asociación entre las clases `Factura` y `Cta_Cte` provoca también una relación de dependencia.

La clase `Factura` es la clase **cliente** de las clases `Cta_Cte` y `String`. La clase `Cta_Cte` es **proveedora** de servicios respecto a `Factura`, pero es también **cliente** de la clase `String`.

Las clases `Factura` y `Cta_Cte` son **proveedoras** de los servicios que van a ser usados por las clases que declaren variables de los tipos `Factura` y `Cta_Cte`. Estas clases deben asumir la responsabilidad de asegurar que una factura se crea ligada a un número y a una cuenta corriente. Análogamente se requiere que las clases cliente que envíen el mensaje `establecerClienteFact` a un objeto de clase `Factura`, lo hagan con una variable ligada como parámetro.

La clase `Factura` no controla que `nroFact` y `clienteFact` mantengan referencias ligadas, porque el contrato establece que esta no es su responsabilidad. En un sistema **robusto** cada clase puede establecer controles que prevengan errores provocados por otras clases, cuando estas no cumplen con sus responsabilidades. En este caso se utilizan los mecanismos provistos por el lenguaje para **manejar excepciones**.

En este diseño la clase `Factura` asume la responsabilidad de modificar el saldo de la cuenta corriente cuando se crea un objeto.

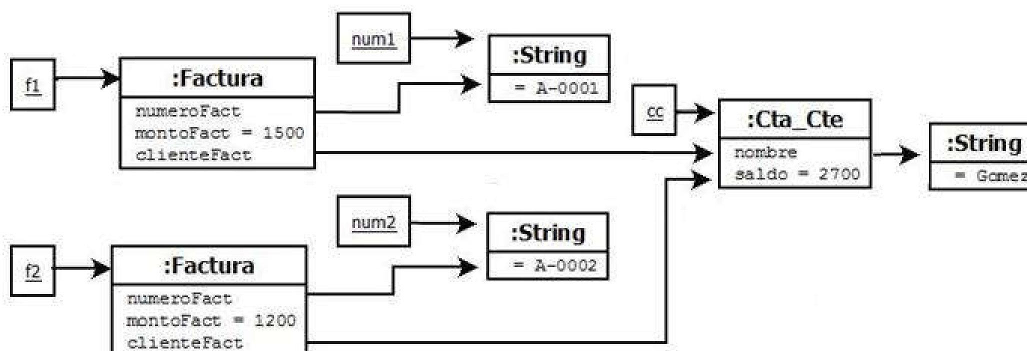
```
class Factura {
private String nroFact;
private float montoFact;
private Cta_Cte clienteFact;
public  Factura  (String n, float m,Cta_Cte cli){
/*Crea una factura, guarda número, monto y cliente, y actualiza el saldo de la cuenta corriente del cliente con el mismo monto. Requiere que n y cli estén ligadas */
 nroFact = n;
 montoFact = m;
 clienteFact = cli;
 clienteFact.actualizarSaldo (m);}
```

El constructor envía el mensaje `actualizarSaldo` al objeto ligado a `clienteFact`, asumiendo que es una referencia ligada, porque así lo establece el requerimiento en el diagrama.

Después de la tercera asignación, las variables `cli` y `clienteFact` están ligadas a un mismo objeto. Observemos que aunque el valor de la variable `cli` no cambia, sí se modifica el estado interno de la variable referenciada por `cli`. Decimos que la ejecución del constructor tiene un **efecto colateral**, no solo inicializa el estado interno del objeto que se crea sino que modifica también el estado interno de uno de los objetos que se recibe como parámetro.

```
class Ventas {
…
public static void test(){
String num1 = new String("A-0001");
String num2 = new String("A-0002");
Cta_Cte cc = new Cta_Cte ("Gomez");
…
Factura f1 = new Factura (num1,1500,cc);
Factura f2 = new Factura (num2,1200,cc);}
…}
```

La clase `Ventas` es cliente de la clase `Factura` y es responsable de asegurar que el tercer parámetro del constructor es una variable ligada, pero no de actualizar `saldo`. El diagrama de objetos después de la ejecución del segmento anterior es:



Consideremos el siguiente diseño alternativo:

*En un negocio se desea mantener información referida a las **facturas** imputadas a las cuentas corrientes de los **clientes**. De cada **<u>factura</u>** se mantiene el **número**, el **monto** y el **cliente**. De*

cada ***cuenta corriente*** *se mantiene el* ***nombre*** *del cliente y el* ***saldo****. Cuando se crea una factura, inmediatamente después se actualiza el saldo de la cuenta corriente.*

En esta alternativa de diseño cada vez que se crea un objeto de clase `Factura` la responsabilidad de actualizar el saldo es de la clase cliente. El código para la segunda alternativa de diseño de `Factura` es:

```
class Factura {
private String nroFact;
private float montoFact;
private Cta_Cte clienteFact;
public  Factura  (String n, float m,Cliente cli){
/*Crea una factura, guarda número, monto  y cliente. Requiere que n y
cli ligadas */
 nroFact = n;
 montoFact = m;
 clienteFact = cli;}
…
}
```

Nuevamente la clase `Ventas` **usa** los servicios provistos por `Factura`, pero además asume la responsabilidad de actualizar el saldo de la cuenta corriente del cliente.

```
class Ventas {
…
public static void test(){
  String num1 = new String("A-0001");
  String num2 = new String("A-0002");
  Cta_Cte cc = new Cta_Cte ("Gomez");
  Factura f1 = new Factura (num1,1500,cc);
  cc.actualizarSaldo (1500);
  Factura f2 = new Factura (num2,1200,cc);
  cc.actualizarSaldo (1200);}
…}
```

El diagrama de objetos al completarse la ejecución del método test, es el mismo cualquiera sea la alternativa elegida.

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. El responsable del testeado busca errores de aplicación en base al contrato.

Cualquiera sea la decisión de diseño el código parcial de la clase `Cta_Cte` es:

```
class Cta_Cte {
//Atributos de Instancia
private String nombre;
private float saldo;
//Constructores
public Cta_Cte (String n){
  nombre =n;}
//Comandos
public void actualizarSaldo (float s) {
 saldo += s;}
…
}
```

Observemos que 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. 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 cliente, va a producirse un **error de aplicación**, que pasa desapercibido para el compilador.

Problemas Propuestos

1. En una Biblioteca se registran los datos que permiten hacer un seguimiento de los préstamos de distintos tipos de ítems bibliográficos, en particular libros:

Prestamo	Item	Fecha
<<Atributos de instancia>> libro:Item socio:String fechaPrestamo:Fecha dias:entero fechaDevolucion:Fecha	<<Atributos de instancia>> nombre:String autor:String editorial:String	<<Atributos de instancia> dia: entero mes: entero anio: entero <<Constructor>> Fecha(d,m,a:entero)
<<Constructor>> Prestamo(l:Item, fp:Fecha, d:entero) <<Comandos>> establecerFDevolucion (fd:Fecha) <<Consultas>> obtenerLibro(): Libro obtenerFPrestamo(): Fecha obtenerFDevolucion() :Fecha estaAtrasado(f:Fecha) : boolean penalizacion(): Fecha	<<Constructor>> Item(n,a, e:String) <<Comandos>> establecerNombre n:String) establecerAutor(a :String) establecerEditori al(e:String) <<Consultas>> obtenerNombre():S tring obtenerAutor():St ring obtenerEditorial( ):String	<<Comandos>> establecerDia (d:entero) establecerMes (m:entero) establecerAnio (a:entero) <<Consultas>> obtenerDia(): entero obtenerMes(): entero obtenerAnio(): entero esValida(d,m,a:entero) esAnterior(f:Fecha):boole no sumaDias(d:entero):Fecha

La clase `Item` *brinda los comandos y consultas triviales. El constructor requiere los tres parámetros ligados.*

En la clase `Fecha`*:*

El constructor requiere que `d`*,* `m` *y* `a` *representen una fecha válida.*

`esAnterior(f:Fecha)` *retorna verdadero si la fecha que recibe el mensaje es menor que el parámetro* `f`*.*

`sumaDias(d:entero)` *retorna la fecha que resulta de sumar* `d` *días a la fecha que recibe el mensaje.*

En la clase `Prestamo`*:*

El constructor requiere l y fp ligados y d mayor que 0.

`estaAtrasado(f:Fecha)` *recibe como parámetro la fecha actual y retorna verdadero si el libro no se devolvió y ya debería haberse devuelto de acuerdo a la fecha de préstamo y la cantidad de días.*

`penalizacion` *calcula la cantidad de días de penalización y devuelve la fecha hasta la que corresponde aplicar la penalización, a partir de la fecha de devolución, que tiene que estar ligada. La penalización es de 3 días si se atrasó menos de una semana, 5 días si se atrasó menos de 30 días y 10 días si se atrasó 30 días o más. Si el libro tiene categoría A los días de penalización se duplican. Requiere que la fecha de devolución esté ligada.*

a. *Implemente el diagrama completo*
b. *Implemente una clase tester para cada especificación.*

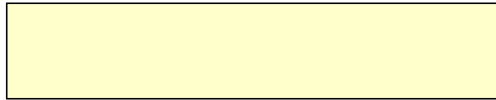
2. En un campeonato de hockey por cada partido ganado se obtienen 3 puntos y por cada empate se logra 1. Cada equipo tiene un nombre, un capitán, una cantidad de partidos ganados, otra de empatados y otra de perdidos, una cantidad de goles a favor y otra de goles en contra. El capitán es un jugador que tiene un nombre, un número de camiseta, un número que representa la posición en la que juega, la cantidad de partidos jugados y la cantidad de goles convertidos en el campeonato.

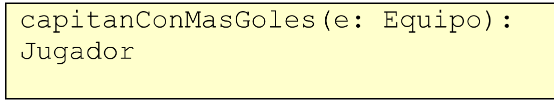
Para un jugador se desea calcular el promedio de goles por partido y dado otro jugador, cuál es el que hizo más goles. Para un equipo se desea calcular los partidos jugados y los puntos obtenidos. Además para otro equipo dado es necesario decidir cuál es el equipo con mayor puntaje y cuál es el capitán con más goles. Si dos equipos tienen en los mismos puntos, se devuelve el que tiene mayor cantidad de goles a favor y si también hay coincidencia se consideran los goles en contra. Si hay coincidencia se devuelve uno cualquiera.

El siguiente diagrama modela las clases Jugador y Equipo:

```
Jugador
-------------------------------------
<<Atributos de instancia>>
nombre: String
nroCamiseta: entero
posicion: entero
golesConvertidos: entero
partidosJugados: entero
-------------------------------------
<<Constructor>>
Jugador(nom:String)
<<Comandos>>
aumentarGoles(n:entero)
aumentarUnPartido()
<<Consultas>>
promedioGolesXPart(): entero
jugConMasGoles(j: Jugador):
Jugador
masGoles(j:Jugador): boolean
toString(): String
```

```
Equipo
-------------------------------------
<<Atributos de instancia>>
nombre: String
capitan: Jugador
pG,pE,pP: entero
gFavor, gContra: entero
-------------------------------------
<<Constructor>>
Equipo(nom:String, cap: Jugador)
<<Comandos>>
incrementarPG(jugoElCap: boolean)
incrementarPE(jugoElCap: boolean)
incrementarPP(jugoElCap:boolean)
incrementarGfavor (total, delCap:
entero)
incrementarGContra(total: entero)
<<Consultas>>
partidos(): entero
puntos(): entero
mejorPuntaje(e: Equipos): Equipo
```





a. Implemente las clases modeladas en el diagrama agregando los métodos triviales para obtener y establecer atributos y considerando la siguiente especificación:

masGoles() devuelve true si el jugador que recibe el mensaje tiene más goles que el jugador que corresponde al parámetro.

incrementarPG(jugoElCap: boolean) , incrementarPE(jugoElCap: boolean),

incrementarPP(jugoElCap: boolean) *Aumentan en 1 los partidos del equipo y si corresponde envía un mensaje al capitán para que incremente en 1 sus partidos.*

incrementarGFavor(total, delCap: entero): *Aumenta los goles convertidos por el equipo y si corresponde envía un mensaje al capitán para actualizar sus goles.*

incrementarGContra(total: entero): *Aumenta los goles en contra del equipo*

b. Considerando el mismo diseño para la clase Jugador implemente la case Equipo de acuerdo a la siguiente especificación:

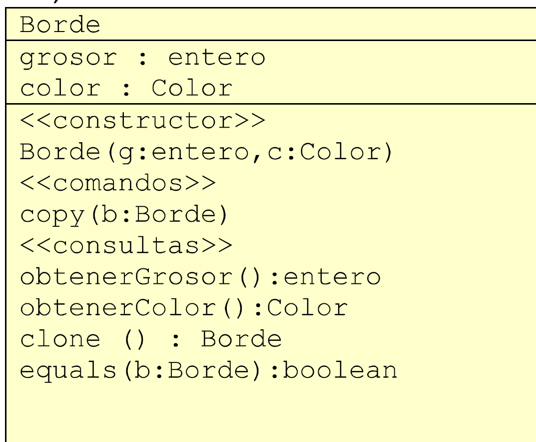
incrementarPG() incrementarPE() incrementarPP(): *Aumenta en 1 los partidos del equipo. Requiere que la clase cliente aumente, si corresponde la cantidad de partidos jugados por el capitán.*

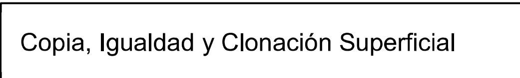
incrementarGFavor(total: entero): *Aumenta los goles del equipo. Requiere que la clase cliente aumente, si corresponde la cantidad de goles convertidos por el capitán.*

incrementarGContra(total: entero): *Aumenta los goles en contra del equipo.*

c. Implemente una clase tester para cada especificación.

Dada la clase Color implementada en prácticos anteriores y el siguiente diagrama para la clase Borde, asociada a la clase Color:





a. Implemente la clase Borde y verifique para un conjunto de casos de prueba fijos.

b. Dado el siguiente segmento de código:

```
Color C1,C2;
Borde B1,B2;

C1 = new Color(110, 110, 110);
```

```
C2 = new Color(110, 110, 110);
B1 = new Borde(1,C1);
B2 = new Borde(1,C2);

System.out.println( C1 == C2);
System.out.println( B1 == B2);
System.out.println( C1.equals(C2));
System.out.println(B1.equals(B2));
```

i. *Dibuje el diagrama de objetos al terminar el bloque de asignaciones.*
ii. *Muestre la salida*

c. *Dado el siguiente segmento de código:*

```
Color C1,C2;
Borde B1,B2,B3,B4;

C1 = new Color(110, 110, 110);
C2 = new Color(110, 110, 110);
B1 = new Borde(1,C1);
B2 = new Borde(1,C2);
B3 = B2.clone();
B4 = new Borde (B2.obtenerGrosor(),B2.obtenerColor());
B1.copy(B2);

System.out.println( B2 == B1);
System.out.println( B2 == B3);
System.out.println( B2 == B4);
System.out.println(B2.equals(B1));
System.out.println(B2.equals(B3));
System.out.println(B2.equals(B4));
System.out.println(B2.obtenerGrosor() == B1.obtenerGrosor() &
                   B2.obtenerColor() == B1.obtenerColor());
System.out.println(B2.obtenerGrosor() == B3.obtenerGrosor() &
                   B2.obtenerColor() == B3.obtenerColor());
System.out.println(B2.obtenerGrosor() == B4.obtenerGrosor() &
                   B2.obtenerColor() == B4.obtenerColor());
System.out.println(B2.obtenerGrosor() == B1.obtenerGrosor() &
                   B2.obtenerColor().equals(B1.obtenerColor()));
System.out.println(B2.obtenerGrosor() == B3.obtenerGrosor() &
                   B2.obtenerColor().equals(B3.obtenerColor()));
System.out.println(B2.obtenerGrosor() == B4.obtenerGrosor() &
                   B2.obtenerColor().equals(B4.obtenerColor()));
```

i. *Dibuje el diagrama de objetos al terminar el bloque de asignaciones.*
ii. *Muestre la salida*

d. *Dibuje el diagrama de objetos y muestre la salida del ejercicio anterior considerando que la clase Borde implementa:*

i. *clonación en profundidad, igualdad superficial , copia superficial*
ii. *clonación superficial, igualdad en profundidad, copia superficial*
iii. *clonación en profundidad, igualdad en profundidad, copia superficial*
iv. *clonación en profundidad, igualdad en profundidad, copia en profundidad*