5. Implementación de aplicaciones con múltiples clases en Java

Como se vio en el capítulo 2 de este libro, sección 2.2, las aplicaciones de software bajo el enfoque de POO se definen en base a las relaciones establecidas entre un grupo de clases. En el capítulo 3 se abordó la implementación en Java de aplicaciones simples, en las que solo participaban una clase entidad y una clase controladora (a la que hemos llamado Sistema).

En este capítulo se abordará el desarrollo de aplicaciones un poco más complejas, donde intervienen varias clases entidad, una clase controladora y una o más clases que proveen la interfaz con el usuario, siguiendo la arquitectura Modelo-Vista-Controlador. Dichas aplicaciones considerarán las relaciones que se pueden establecer entre las clases y, por ende, entre los objetos pertenecientes a ellas. Concretamente: Asociación, Agregación/Composición y Herencia.

5.1. Asociación

Suponga que se desea establecer la relación entre personas y productos, debido a que las personas compran productos. Vea el diagrama de clases de la Figura 5.1.

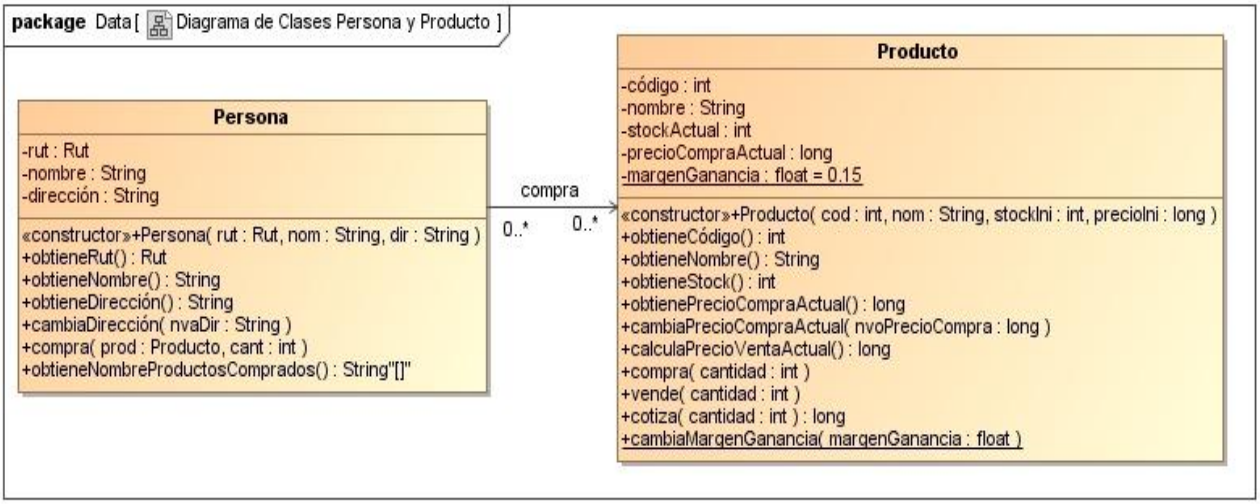

Figura 5.1. Asociación entre Persona y Producto.

En este caso, la asociación compra se podría implementar a través de las clases Persona y Producto, de manera que, dada una persona se pueda determinar qué productos compra y, dado un producto, se pueda determinar qué personas lo compran. Sin embargo, puede observarse que la relación es unidireccional, va sólo desde Persona a Producto. Esto implica que la relación debe implementarse sólo en la clase Persona.

Ejemplo 5.1, una definición general de la clase Persona y la clase Producto:

```
import java.util.LinkedList;  // Importa la clase LinkedList
public class Persona {
      // Atributos          // Suposición: Rut es una clase utilitaria ya definida
      private Rut rut;
      private String nombre;
      private String dirección;

      // Relación
      private LinkedList<Producto> compra = new LinkedList<Producto>();

    // Constructor
      public Persona(Rut rut, String nom, String dir ) {
        // Precondición: rut, nom, dir <> null, trim(nom), trim(dir) <> ""
        this.rut = rut;
        nombre = nom;
        dirección = dir;
      }

      // Métodos
      public Rut obtieneRut() {
        return rut;
      }

      public String obtieneNombre() {
        return nombre;
      }

      public String obtieneDirección() {
        return dirección;
      }

      public void cambiaDirección(String nvaDir) {
        // Precondición: nvaDir <> null, trim(nvaDir) <> ""
        Dirección = nvaDir,
      }

      public void compra(Producto prod, int cant) {
        // Precondición: prod<>null, cant>0
        if(!compra.contains(prod)) {
            compra.add(prod);
        }
            prod.vende(cant);
      }

      public String[] obtieneNombreProductosComprados() {
        String[] nomProdComprados = new String[compra.size()];
        for (int i=0; i<compra.size(); i++) {
            nomProdComprados[i] = compra.get(i).obtieneNombre();
        }
            return nomProdComprados;
      }
}   /* Fin clase Persona */
```

La variable de instancia "compra" almacenará las referencias a los objetos Producto que una cierta persona haya comprado (representado por el objeto respectivo).

Método que agrega el objeto prod al contenedor compra (LinkedList) si es la primera vez que este cliente compra este producto, además actualiza stock del producto prod.

Invoca el método venta del objeto prod, método definido en la clase Producto.

Método que recupera el nombre de todos los productos relacionados con esta persona (cliente).

```
public class Producto {
      // Atributos
      private int código;
      private String nombre;
      private int stockActual;
      private long precioCompraActual;
      private static float margenGanancia=0.15;

      //Constructor
      public Producto(int cod, String nom, int stockIni, long precioIni) {
        // Precondición: cod,stockIni,precioIni > 0, nom<>null, trim(nom)<>""
            código = cod;
            nombre = nom;
            stockActual = stockIni;
            precioCompraActual = precioIni;
      }

      // Métodos
      public int obtieneCódigo() {
        return código;
      }
      public String obtieneNombre() {
        return nombre;
      }

      public int obtieneStock () {
        return stockActual;
      }

      public int obtienePrecioCompraActual() {
        return precioCompraActual;
      }

      public void cambiaPrecioCompraActual(long nvoPrecioCompra) {
        // Precondición: nvoPrecioCompra > 0
        precioCompraActual = nvoPrecioCompra;
      }

      public long calculaPrecioVentaActual() {
        return precioCompraActual * (1 + margenGanancia);
      }

      public void compra (int cantidad) {
        // Precondición: cantidad >= stockActual
        stockActual += cantidad;
      }

      public void vende (int cantidad) {
        // Precondición: cantidad > 0
        stockActual -= cantidad;
      }
```

```
    public long cotiza (int cantidad) {
      // Precondición: cantidad > 0
      return  calculaPrecioVentaActual() * cantidad;
    }

    public static void cambiaMargenGanancia(float margenGanancia) {
      // Precondición: margenGanancia > 0
      this.margenGanancia = margenGanancia;
    }

}   /* Fin clase Producto */
```

> ***¡Importante!***
> Las relaciones del tipo asociación, se pueden implementar mediante arreglos o clases definidas en Java como ArrayList, Vector, LinkedList, etc.

5.2. Agregación y/o Composición

Las relaciones de agregación y/o composición se implementan del mismo modo que las relaciones de asociación. Veamos los ejemplos que a continuación se presentan.

Ejemplo 5.2, implementación de las clases presentes en la Figura 5.2.

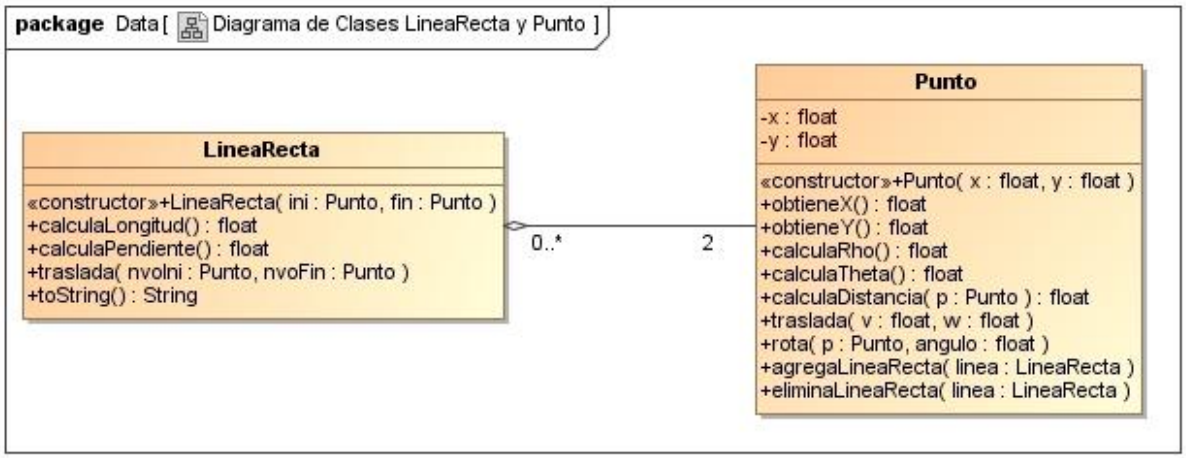

Figura 5.2. Relación de agregación entre LineaRecta y Punto.

En este caso se señala que una línea recta tiene dos puntos (inicio y fin). Una definición para la clase LineaRecta sería:

```
public class LineaRecta {
      // Sin atributos

      // Relaciones
      private Punto puntoInicio;
      private Punto puntoFin;

      // Constructor
      public LineaRecta(Punto ini, Punto fin) {
        // Precondición: ini, fin<>null, ini<>fin
        puntoInicio = ini;
        puntoFin = fin;
        ini.agregaLineaRecta(this);
        fin.agregaLineaRecta(this);
      }

      // Métodos
      public float calculaLongitud() {
        return puntoInicio.calculaDistancia(puntoFin);
      }

      public float calculaPendiente() {
        // Determine el contenido de este método
      }

      public void traslada(Punto nvoIni, Punto nvoFin) {
        // Precondición: nvoIni<>null, nvoFin<>null, nvoIni<>nvoFin
        float distanciaANvoIniX = nvoIni.obtieneX() - puntoInicio.obtieneX();
        float distanciaANvoIniY = nvoIni.obtieneY() - puntoInicio.obtieneY();
        float distanciaANvoFinX = nvoFin.obtieneX() - puntoFin.obtieneX();
        float distanciaANvoFinY = nvoFin.obtieneY() - puntoFin.obtieneY();

        puntoInicio.traslada(distanciaANvoIniX, distanciaANvoIniY);
        puntoFin.traslada(distanciaANvoFinX, distanciaANvoFinY);
      }

      public String toString() {
        return "{ (" + puntoInicio.obtieneX() + "," + puntoInicio.obtieneY() + "); (" +
                puntoFin.obtieneX() + "," + puntoFin.obtieneY() + ") }";
      }

} /* Fin clase LineaRecta */
```

A la línea recta que se crea se le asocian los dos puntos que la definen.

A cada punto se le asocia la recta que se está creando.

La figura 5.2 muestra que los objetos de la clase Punto requieren conocer los objetos LineaRecta en que participan, por lo tanto, una definición para la clase Punto sería:

```
import java.util.LinkedList;
public class Punto {

    // Atributos
    private float x;
    private float y;

    // Relación
    private LinkedList<LineaRecta> misLineasRectas = new LinkedList<LineaRecta>();

    // Constructor
    public Punto(float a, float b) {
        x = a;
        y = b;
    }

    // Métodos
    public float obtieneX() {
      return x;
    }

    public float obtieneY() {
      return y;
    }

    public float calculaRho() {
        // Calcula distancia al origen
        return (float)Math.sqrt(x*x + y*y);
    }

    public float calculaTheta() {
        // Retorna ángulo del punto this respecto de la horizontal
        return (float)Math.atan2(y,x);
    }

    public float calculaDistancia(Punto p) {
      // Precondición: p <> null
        // Retorna la distancia entre el punto this y p
        return (float)Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)* (y-p.y));
    }

    public void traslada(float a, float b) {
        // Traslada el punto una distancia "a" en el eje X, y una distancia "b" en el eje Y
        x = x + a;
        y = y + b;
    }
```

Almacena todas las "líneas rectas" a las que pertenece un "punto".

```
    public void rota(Punto p, float angulo) {
      // Precondición: p <> null
          // Completar
    }

    public void agregaLineaRecta(LineaRecta linea) {
      // Precondición: linea<>null, linea ya tiene asociado este punto
      if (!misLineasRectas.contains(linea)) {
          misLineasRectas.add(linea);
      }
    }

    public void eliminaLineaRecta(LineaRecta linea) {
      // Precondición: linea<>null
      if (misLineasRectas.contains(linea)) {
          misLineasRectas.remove(linea);
      }
    }

}   /* Fin clase Punto */
```

Ejemplo 5.3:

Si sólo se desea implementar la relación entre los objetos LineaRecta y Punto de la Figura 5.2, esto es, sólo se requiere obtener qué puntos forman parte de una recta (y no en qué rectas está contenido un punto), entonces se deben realizar los siguientes cambios a la solución anterior:

- Desaparece, de la clase Punto, la variable de instancia que implementa la relación con LineaRecta, es decir, desaparece:

```
private LinkedList<LineaRecta> misLineasRectas = new LinkedList<LineaRecta>();
```

- Desaparecen los métodos agregaLineaRecta y eliminaLineaRecta de la clase Punto.
- El constructor de la clase LineaRecta cambia a:

```
public LineaRecta(Punto ini, Punto fin) {
    // Precondición: ini<>null, fin<>null, ini<>fin
    puntoInicio = ini;
    puntoFin = fin;
}
```

5.3. Herencia

La relación de herencia es un tanto distinta a las anteriores y, por lo tanto, los lenguajes de programación proveen mecanismos diferentes para implementarla, tal es el caso de Java también.

Para comprender la implementación de esta relación es necesario estar familiarizado con algunos conceptos importantes que fueron tratados anteriormente, pero que serán presentados nuevamente, pues cobran importancia en este momento. Dichos conceptos serán abordados considerando el ejemplo presentado en la Figura 5.3.

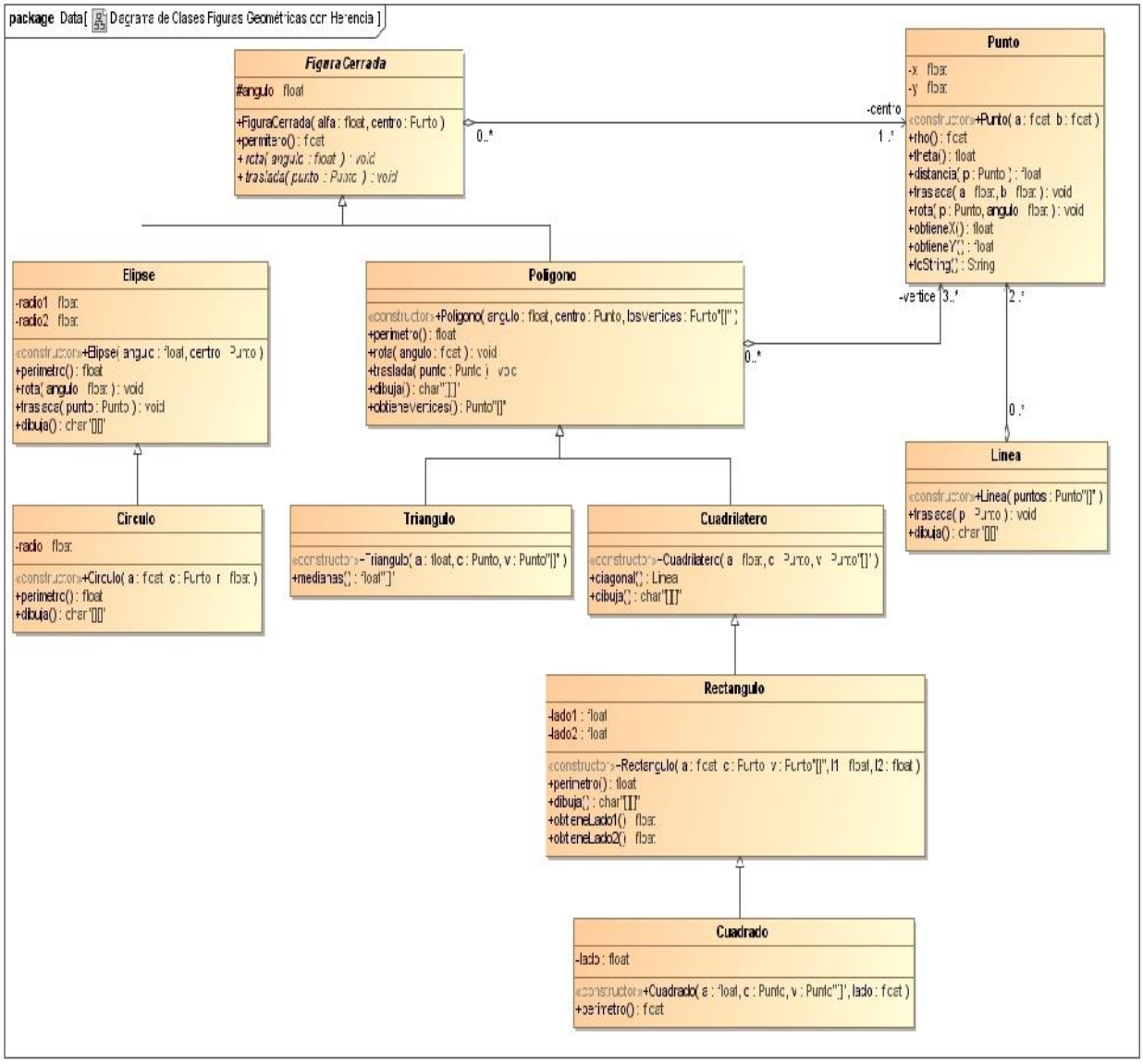

Figura 5.3. Relación de herencia entre Cuadrilatero, Rectangulo y Cuadrado.

5.3.1. Superclases y Subclases

Se dice que una clase es superclase de otra si la primera es padre de la segunda, siendo esta última una subclase de la primera. Por ejemplo, en la Figura 5.3 la clase Cuadrado es subclase de Rectangulo, y ésta es superclase de Cuadrado, pero es una subclase de Cuadrilatero.

En Java, para indicar que una clase se hereda de otra (o es subclase de otra) se utiliza la expresión **extends** al iniciar la definición de la subclase, ello se ilustrará definiendo las clases Rectangulo y Cuadrado de la Figura 5.3.

Ejemplo 5.4, una definición de las clases Rectangulo y Cuadrado:

```
public class Rectangulo extends Cuadrilatero {
      // Atributos
      private float lado1;
      private float lado2;

      // Constructor
      public Rectangulo(float a, float b) {
            // Precondición: a>0, b>0
            lado1 = a;
            lado2 = b;
      }

      // Métodos
      public float perimetro() {...}

      public char[][] dibuja() {...}

}  /* Fin clase Rectángulo */
```

Indica que la clase Rectangulo hereda atributos y métodos de la clase Cuadrilatero y, por su intermedio, de sus superclases.

```
public class Cuadrado extends Rectangulo {

      // Sin Atributos

      // Constructor
      public Cuadrado (float a) {...}

      // Métodos
      public float perimetro() {...}

}  /* Fin clase Cuadrado */
```

Hereda de la clase Rectangulo y de todas las superclases de éste, es decir, además de rectángulo, hereda de:

- Cuadrilatero
- Poligono
- FiguraCerrada

Ver Figura 5.3.

5.3.2. Miembros Protected

Hasta ahora hemos programado utilizando dos tipos de acceso a miembros de una clase (atributo o método), éstos son: privado (private) y público (public). Pero, existe un tercer tipo de acceso, que es intermedio a los anteriores, es el llamado acceso protegido (protected), cuyo símbolo en UML es # y tiene sentido en relaciones de herencia. Cuando un miembro de una clase, por ejemplo un atributo, se define con acceso protegido significa que dicho atributo podrá ser accedido por la clase donde se define y por todas sus subclases, pero será inaccesible para otras clases. En la Figura 5.3 se define como protected el atributo angulo de la clase FiguraCerrada, en Java esta especificación se escribe como:

```
protected float angulo;
```

Lo anterior implica que este atributo sólo podrá ser accedido por FiguraCerrada y sus subclases; por lo tanto, las clases Linea y Punto no podrían acceder a él, pues para estas clases el atributo angulo se comporta como si hubiese sido definido privado.

5.3.3. Relaciones entre objetos de subclases y objetos de superclases

Debido a la herencia, si un objeto es una ocurrencia de una cierta subclase, entonces el objeto también es una ocurrencia de la superclase correspondiente. Por ejemplo, si se tiene un objeto **c** de la clase Cuadrado, **c** también es un Cuadrilatero, un Poligono y una FiguraCerrada.

Sin embargo, se debe tener claro que un objeto que es una ocurrencia de una superclase no implica que sea un objeto de una subclase particular de esa superclase. En el ejemplo de la Figura 5.3, si un objeto **p** es una ocurrencia de la clase Poligono, no implica que dicho objeto sea un Cuadrado, de hecho podría ser una instancia de Triangulo.

5.3.4. Polimorfismo

La propiedad señalada en la sección anterior permite tener objetos de distintas subclases en una colección que se define del tipo de la superclase común y, de este modo, invocar, **durante ejecución**, los métodos correctos dependiendo del tipo concreto del objeto. Esto se conoce como polimorfismo.

El polimorfismo resulta posible de implementar en los lenguajes de programación como Java cuando estos cuentan con una característica conocida como binding dinámico, es decir, cuando son capaces de asignarles tipos (o clases) concretos a las variables de manera tardía, no durante compilación, sino durante ejecución.

Ejemplo 5.5:

Se podría definir un arreglo de rectángulos, cuadrados y triángulos de la siguiente forma:

```
Poligono [] figuras = new Poligono[10];
```

y luego crear objetos de las distintas clases y almacenarlos en el arreglo:

```
figuras[0] = new Cuadrado(...);
figuras[1] = new Triangulo(...):
figuras[2] = new Poligono(...);
```

Esta asignación permite invocar el método perimetro, sin tener preocupación de la clase a la que pertenece el objeto, sabiendo que se encuentra definido tal método para todo objeto de la clase Poligono y, por lo tanto, de sus subclases. Es válido, en consecuencia, usar la siguiente instrucción:

```
for (int i=0; i<10; i++)
    figuras[i].perimetro();
```

En ejecución se invocará el método apropiado gracias al ***Polimorfismo***.

Antes de pasar al siguiente concepto, veamos un ejemplo más simple de herencia presentado en la Figura 5.4. Como se puede observar, existe una clase llamada MaterialBibliográfico que agrupa distintos tipos de materiales bibliográficos, en el caso de nuestro ejemplo videos y libros. A este proceso de definir clases más generales a partir de clases más especializadas se le llama generalización. En este caso, Video y Libro son las clases especializadas, y MaterialBibliográfico es la clase general.

Como todo material bibliográfico debe tener un título y un año de publicación o edición asociado, se crea una clase MaterialBibliografico que contempla estos atributos (comunes para cualquier material bibliográfico). Luego, las clases Libro y Video heredarán dichos atributos y agregarán sus propios y específicos atributos (al ser más especializadas).

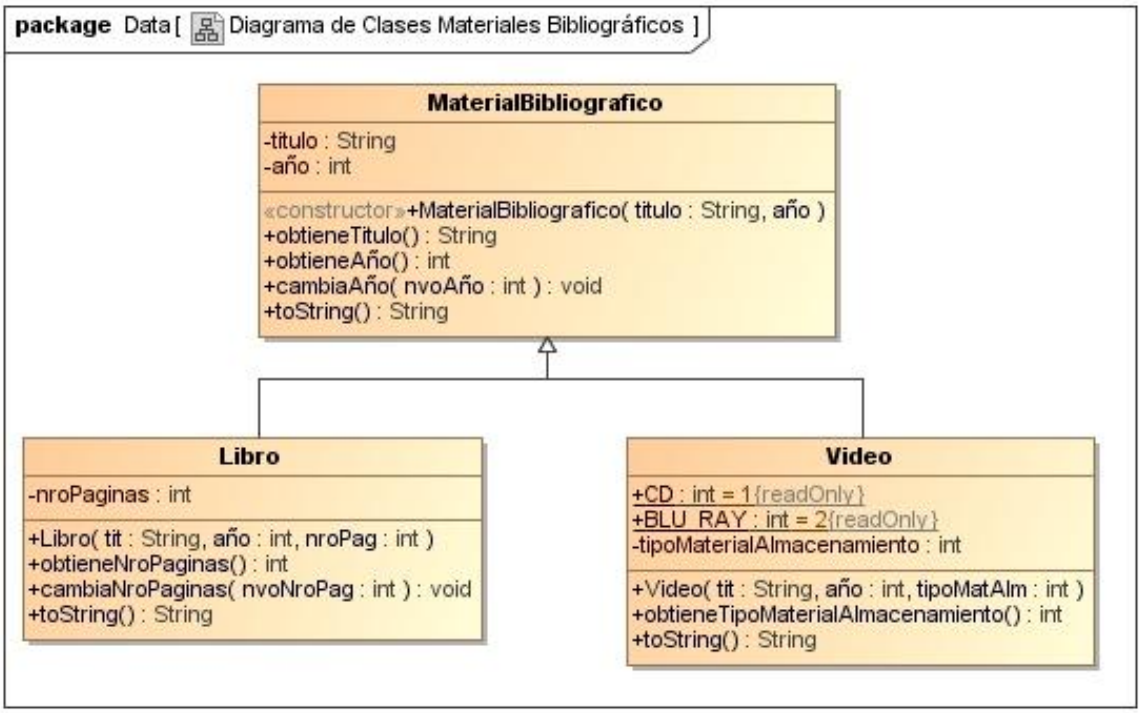

Figura 5.4. Relación de herencia entre MaterialBibliografico, Libro y Video.

Ejemplo 5.6, implementación de la relación de herencia entre MaterialBibliografico, Libro y Video.

```
public class MaterialBibliografico {
      // Atributos
      public String titulo;
      public int año;

      // Constructor
      public MaterialBibliografico(String titulo, int año) {
        // Precondicion: titulo<>null, length(trim(titulo))>0, año>0
        this.titulo = titulo;
        this.año = año;
      }

      // Métodos
      public String obtieneTitulo() {
        return titulo;
      }

      public int obtieneAño() {
        return año;
      }

      public void cambiaAño(int nvoAño) {
        año = nvoAño;
      }

      public String toString() {
        return titulo + "," + año;
      }
}  // Fin clase MaterialBibliografico


public class Libro extends MaterialBibliografico {

        // Atributos
        private int nroPaginas;

        // Constructor
        public Libro(String tit, int año, int nroPag){}
            // Precondicion: tit<>null, length(trim(tit))>0, año>0, nroPag>0

            super(tit,año);

            nroPaginas = nroPag;
        }

            // Métodos
        public String obtieneNroPaginas(){
            return nroPaginas;
        }
```

Indica que hereda los atributos y métodos de la clase MaterialBibliografico, aunque no pueda acceder a los atributos por ser privados.

Atributo que agrega la clase Libro.

Invocación al constructor de la superclase MaterialBibliografico.

Nuevo método, propio de la subclase Libro.

```
    public void cambiaNroPaginas(nvoNroPag) {
        nroPaginas = nvoNroPag;
    }

    public String toString() {

        return (super.toString() +  "," + nroPaginas);
    }

}  // Fin clase Libro


public class Video extends MaterialBibliografico {

    // Constantes
    public final static int CD=1;
    public final static int BLU_RAY=2;

    // Atributos
    private int tipoMaterialAlmacenamiento;

    // Constructor
    public Video(String tit, int año, int tipoMatAlm){}
        // Precondicion: tit<>null, length(trim(tit))>0, año>0, tipoMatAlm>0

        super(tit,año);

        tipoMaterialAlmacenamiento = tipoMatAlm;
    }

        // Métodos
    public String obtieneTipoMaterialAlmacenamiento() {
        return tipoMaterialAlmacenamiento;
    }

    public String toString() {

        return (super.toString() + "," + tipoMaterialAlmacenamiento);
    }

}  // Fin clase Video
```

Método redefinido.

Método original de la clase MaterialBibliografico.

Atributo que agrega la subclase Video.

Invocación al constructor de la superclase (MaterialBibliografico).

Nuevo método, propio de la subclase Video.

Método redefinido por la clase Video.

Método original de la clase MaterialBibliografico.

5.4. Clases Abstractas

Con el concepto de herencia aparece el concepto de clase abstracta. Una clase es abstracta cuando, siendo una superclase, constituye más bien un concepto creado durante el proceso de análisis del problema y, por lo tanto, supone incorrecto crear objetos de dicha clase tal como se explicó en el punto **¡Error! No se encuentra el origen de la referencia.**. Sólo tiene como fin definir una plantilla que cada subclase tomará y concretará, ya sea, agregando atributos, agregando y/o redefiniendo métodos.

En el ejemplo presentado en la Figura 5.3, la clase FiguraCerrada se ha definido como una clase abstracta (ello se evidencia mediante el nombre de la clase en letra cursiva), esto es, **no** se podrán crear objetos de esta clase, pero **sí** de sus subclases.

Crear un objeto no es lo mismo que declarar un objeto, luego en una clase que use estas figuras podría aparecer lo siguiente:

```
FiguraCerrada [] figuras = new FiguraCerrada[10];
figuras[0] =  new Cuadrado(...)
```

pero **NO** podría aparecer:

```
figuras[1] = new FiguraCerrada(...)
```

En Java, una clase abstracta se define usando la palabra **abstract.**

Ejemplo 5.7, una definición de FiguraCerrada:

```
public abstract class FiguraCerrada {

    // Atributos
    protected float angulo;

    // Asociaciones
    protected  Punto centro;

    // Constructor
    public FiguraCerrada(float alfa, Punto centro) {...}


    // Métodos
   public float perímetro() {
         return 0;
   }

    public abstract void rota(float angulo);

    public abstract void traslada(Punto punto);

    public abstract char[][] dibuja();

}  /* Fin clase FiguraCerrada */
```

Atributos que pueden ser accedidos directamente por las subclases de *FiguraCerrada*.

Una clase abstracta puede tener constructor, pero no debe ser usado para crear objetos de la clase.

Método concreto que puede ser redefinido en las subclases, cuando corresponda.

Métodos abstractos. Deben definirse, obligatoriamente, en las subclases no abstractas.

Una clase abstracta puede tener **métodos abstractos** y/o **concretos**. Un **método abstracto** es aquél cuya definición no se da, pues dependerá de la subclase particular de que se trate, esto quiere decir que la definición se deja a las subclases. Se agrega como método de la superclase abstracta para dejar expresado que tal método es común a todas sus subclases y, por tanto, lo heredarán, pero tienen la responsabilidad de concretarlo (definirlo con las instrucciones adecuadas).

¡Importante!
Una clase abstracta puede tener definido un constructor, sin embargo, tal constructor será invocado sólo por las subclases, en sus propios constructores, durante el proceso de creación de un objeto de las subclases.

Ejemplo 5.8, una implementación más completa de las clases de la Figura 5.3:

```
public class Punto {
    // Atributos
    private int x, y;

    // Constructor
    public Punto(int a, int b) {
      x = a;
      y = b;
    }

    // Métodos
    public float rho() {
      // Calcula distancia al origen
      return (float)Math.sqrt(x*x + y*y);
    }

    public float theta() {
      // Retorna ángulo del punto this respecto de la horizontal
      return (float)Math.atan2(y,x);
    }

    public int distancia(Punto p) {
      // Retorna la distancia entre el punto this y p
      return (int)Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)* (y-p.y));
    }

    public void traslada(int a, int b) {
      // Traslada el punto una distancia "a" en el eje x
      // y una distancia "b" en el eje y
      x = x + a;
      y = y + b;
    }

    public void rota(Punto p, float angulo) { /* …pendiente */ }
```

```
    public int obtieneX() {
      return x;
    }

    public int obtieneY() {
      return y;
    }

    public String toString() {
      return "("+x+","+y+")";
    }

} /* Fin clase Punto */


import java.util.LinkedList;
public class Linea {

    // Asociaciones
    protected LinkedList<Punto> misPuntos;

    // Constructor
    public Linea(Puntos[] p) {
          misPuntos = new LinkedList<Punto>(p.length);
          for (int i=1; i<=p.length; i++)
              if (p[i-1] != null)
                  misPuntos.add(p[i-1]);
    }

    // Métodos
    public void traslada(Punto p) { /* ...pendiente… */ }

    public char[][] dibuja() { /* ...pendiente… */ }

}  /* Fin clase Linea */



public abstract class FiguraCerrada {

    // Atributos
    protected float angulo;

    // Agregación
    protected  Punto centro;

    // Constructor
    public FiguraCerrada(float alfa, Punto centro) {
      angulo = alfa;
      this.centro = centro;
    }
```

```
    // Métodos
    public float perímetro() {
        return 0;
    }

    public abstract void rota(float angulo);

    public abstract void traslada(Punto punto);

    public abstract char[][] dibuja(); /* …Pendiente definición completa de la cabecera… */

}  /* Fin clase FiguraCerrada */
```

A la clase Polígono se le añadirá un método que devuelva todos los vértices que lo conforman, dicho método se llamará obtieneVertices. A continuación la definición de esta clase.

```
import java.util.LinkedList;

public class Poligono extends FiguraCerrada {

    // Agregación
    protected LinkedList<Punto> misVertices;

    // Constructor
    public Poligono(float angulo, Punto centro, Punto[] losVertices) {
        super(angulo,centro);
        misVertices = new LinkedList<Punto>(losVertices.length);
        for (int i=1; i<=losVertices.length; i++)
            if (losVertices[i-1]!=null)
                misVertices.add(losVertices[i-1]);
    }

    // Metodos
    public float perímetro() { /* …pendiente… */ }

    public void rota(float angulo) {  /* ...pendiente… */ }

    public void traslada(Punto punto) {
        // Precondición: punto<>null
        centro = punto;
      }
```

La clase Poligono puede acceder al atributo centro heredado desde la clase FiguraCerrada ya que se definió protected.

```
    public char[][] dibuja() {  /* Se define el método abstracto dibuja */
        char[][] matrizPantalla = new char[23][80];

        for (int i=0; i<23; i++)
            for (int j=0; j<80; j++)
                matrizPantalla[i][j] = ' ';
        for (int i=0; i<misVertices.size(); i++) {
            Punto p = misVertices.get(i);
            if (p.obtieneX()>=0 && p.obtieneX()<=39 && p.obtieneY()>=0 && p.obtieneY()<=23)
                matrizPantalla[22-p.obtieneY()][p.obtieneX()*4] = 'x';
        }
        return matrizPantalla;
    }

    public Punto[] obtieneVertices() {
        return misVertices.toArray(new Punto[0]);
    }

} /* Fin clase Poligono */
```

Método de la clase linkedList que retorna todos los objetos Punto almacenados en la colección **misVertices** como un arreglo.

```
public class Cuadrilatero extends Poligono {

    //Constructor
    public Cuadrilatero(float angulo, Punto centro, Punto[] puntos) {
        super(angulo,centro,puntos);
    }

    // Métodos
    public Linea diagonal() { /* ...pendiente */ }

    public char[][] dibuja() { /* …pendiente */ }

} /* Fin clase Cuadrilatero */


public class Rectangulo extends Cuadrilatero {
    // Atributos
    private int lado1, lado2;

    // Constructor
    public Rectangulo(float angulo, Punto centro, Punto[] puntos, int lado1, int lado2) {
        super(angulo,centro,puntos);
        this.lado1 = lado1;
        this.lado2 = lado2;
    }
```

```
    // Métodos
    public int perimetro() {
        return 2*lado1 + 2*lado2;
    }

    public char[][] dibuja() { /* …pendiente */ }

    public float obtieneLado1() {
        return lado1;
    }

    public float obtieneLado2() {
        return lado2;
    }

} /* Fin clase Rectangulo */


public class Cuadrado extends Rectangulo {

    // Atributos
    private int lado;

    // Constructor
    public Cuadrado (float angulo, Punto centro, Punto [] puntos, int lado) {
        super(angulo,centro,puntos,lado,lado);
    }

    // Métodos
    public int perimetro() {
        return 4 * obtieneLado1();
    }

} /* Fin clase Cuadrado */
```

Ejemplo 5.8:

La clase MaterialBibligrafico presente en la Figura 5.4 debería definirse abstracta pues los libros y videos son materiales bibliográficos reales, en cambio material bibliográfico no existe como tal, sólo es una conceptualización que generaliza a las clases Libro y Video. Por lo tanto, una definición más adecuada de esta clase se muestra en la Figura 5.5.

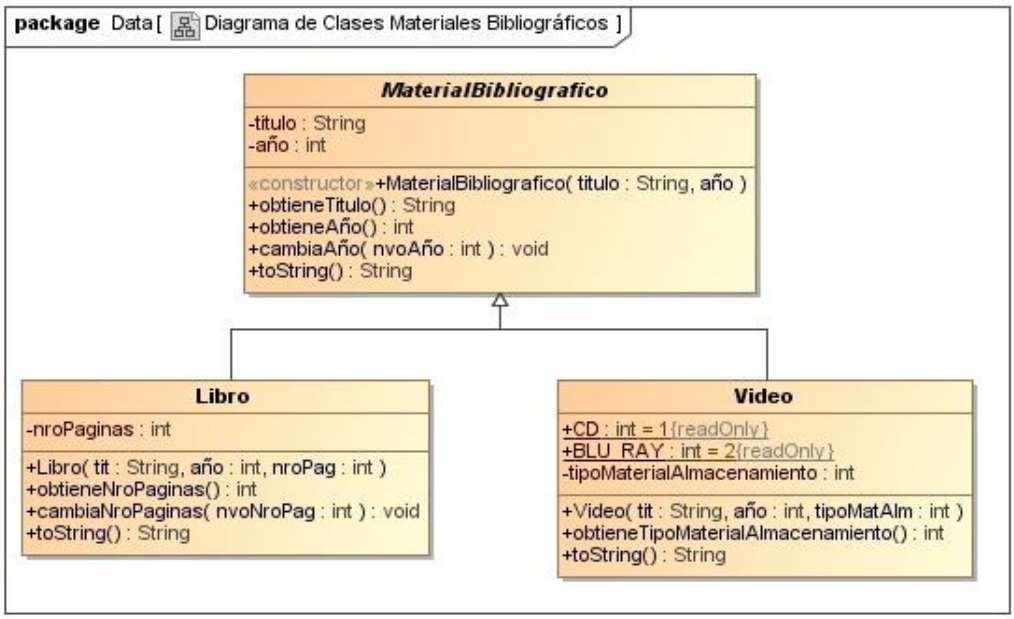

Figura 5.5. Relación de herencia mejor definida entre MaterialBibliografico, Libro y Video.

La implementación en Java de la clase MaterialBibliográfico sería ahora:

```
public abstract class MaterialBibliografico {
      // Atributos
      public String titulo;
      public int año;

      // Constructor
      public MaterialBibliografico(String titulo, int año) {
        // Precondicion: titulo<>null, length(trim(titulo))>0, año>0
        this.titulo = titulo;
        this.año = año;
      }

      // Métodos
      public String obtieneTitulo() {
        return titulo;
      }
      public int obtieneAño() {
        return año;
      }

      public void cambiaAño(int nvoAño) {
        año = nvoAño;
      }
      public String toString() {
        return titulo + "," + año;
      }

}  // Fin clase MaterialBibliografico
```

La anterior es una clase abstracta que no contiene métodos abstractos, sino sólo métodos concretos. Con esta definición se impide la creación de objetos de esta clase, permitiendo la creación de objetos Libro y Video únicamente, cuyas definiciones fueron presentadas en el ejemplo 5.7.

5.5. Arquitectura Modelo Vista Controlador

En primer lugar, una arquitectura de software es el proceso de diseñar la organización global de un sistema de software, incluyendo:

- Dividir el software en subsistemas.
- Decidir cómo interactuarán estos subsistemas, su dinámica y los datos que compartirán.
- Determinar sus interfaces.
    - La arquitectura es la base del diseño, de modo que todos los ingenieros de software necesitan entenderla.
    - La arquitectura a menudo restringirá la eficiencia, reusabilidad y mantenibilidad del sistema completo.

Una arquitectura de software se utiliza para permitir a todos comprender mejor el sistema, trabajar en partes del sistema en forma aislada sin necesidad e interactuar constantemente con los demás, preparar el sistema para que pueda ser extendido en el futuro y facilitar la mantención y el reúso del mismo. Para que un sistema pueda ser extendido y fácilmente mantenido se requiere que sea estable, esto es, que se puedan agregar nuevas funcionalidades o características con solo unos pocos cambios. La arquitectura de software busca alcanzar este propósito.

La arquitectura Modelo Vista Controlador (MVC) contempla 3 capas y se usa para ayudar a separar la interfaz de usuario de las otras partes de una aplicación de software.

El *modelo* contiene las clases (llamadas también clases entidad) cuyas instancias son consultadas y manipuladas a través de los métodos que estas proveen. La *vista* contiene objetos usados para dar la apariencia adecuada a los datos del modelo en la interfaz de usuario y obtener datos de entrada. El *controlador* contiene los objetos que controlan y manejan la interacción de usuario entre la vista y el modelo.

Dado lo anterior, esta arquitectura es usada para separar el modelo de la vista.

Ejemplo 5.9

A continuación se define una clase Sistema que ejemplifica la creación y uso de objetos de las clases Libro y Video. En este caso, laclase Sistema implementa las capas: vista y controlador.

```
public class Sistema {

    public static void main(String[] args)  {

        // Declaración de variables
        MaterialBibliografico[] materiales = new MaterialBibliografico[2];

        // Creación de objetos
        Libro lib = new Libro("Redes de Computadores",2010);
```

```
        materiales[1] = new Video("Paradigmas",2011,Video.CD);

        materiales[0] = lib;
        // Obtiene titulo de los materiales bibliográficos existentes
        for (int i=0; i<2; i++) {
            System.out.println("\n\nEl titulo del material bibliográfico " + (i+1) + " es: " +
                        materiales[i].obtieneTitulo());
            }

        // Muestra los libros y videos completos
        System.out.println("\n\nLos datos de los materiales bibliográficos existentes son: ");
        for (int i=0; i<2; i++) {
            System.out.println((i+1) + ": " + materiales[i]);
        }
    }
}  // Fin clase Sistema
```

Ejemplo 5.10

A continuación se presenta una clase llamada Sistema que crea objetos Punto y un Cuadrado e invoca algunos de sus métodos con el fin de demostrar el funcionamiento de las clases antes definidas. Sistema implementa las capas vista y controlador.

```
public class Sistema {
    public static void main (String []args) {
        // Declaración de variables
        Punto[] puntos = new Punto[4];
        Punto centroDelCuadrado = new Punto(4,5);
        Cuadrado c;
        float angulo=(float)2.5;
        int lado=5;

        puntos[0] = new Punto(1,7);
        puntos[1] = new Punto(6,7);
        puntos[2] = new Punto(6,2);
        puntos[3] = new Punto(1,2);
        c = new Cuadrado(angulo,centroDelCuadrado,puntos,lado);

        char[][] pantalla = c.dibuja();
        for (int i=0; i<pantalla.length; i++) {
            for (int j=0; j<pantalla[0].length; j++) {
                System.out.print(pantalla[i][j]);
            }
            System.out.println();
        }

        Thread.sleep(2000);

        // Sobreescribe los puntos almacenados en el arreglo puntos
```

Método estático de la clase Thread, hace que el programa se detenga el número de milisegundos que se indica.

```
        Punto otroPunto = new Punto(0,0);
        for (int i=0; i<puntos.length; i++) {
            puntos[i]= otroPunto;
        }

        // Despliega el contenido actual del arreglo puntos
        System.out.print("\n\nPuntos actualmente almacenados en el arreglo de puntos: ");
        for (int i=0; i<4; i++)
            System.out.print(puntos[i]+" ");

        // Recupera y despliega los vértices del cuadrado c
        puntos = c.obtieneVertices();

        // Despliega el contenido actual del arreglo puntos
        System.out.print("\n\nLos vértices del cuadrado creado, ahora almacenados en 'puntos': ");
        for (int i=0; i<puntos.length; i++)
            System.out.print(puntos[i] + " ");

    }

} /* Fin clase Sistema */
```

En la Figura 5.7 se observa la impelementación de la arquitectura MVC, donde la clase IUCriadero forma la capa **Vista**, la clase Criadero la capa **Controlador** y las clases entidad Perro y Raza constituyen la capa **Modelo**.

La implementación de las clases es la que sigue.

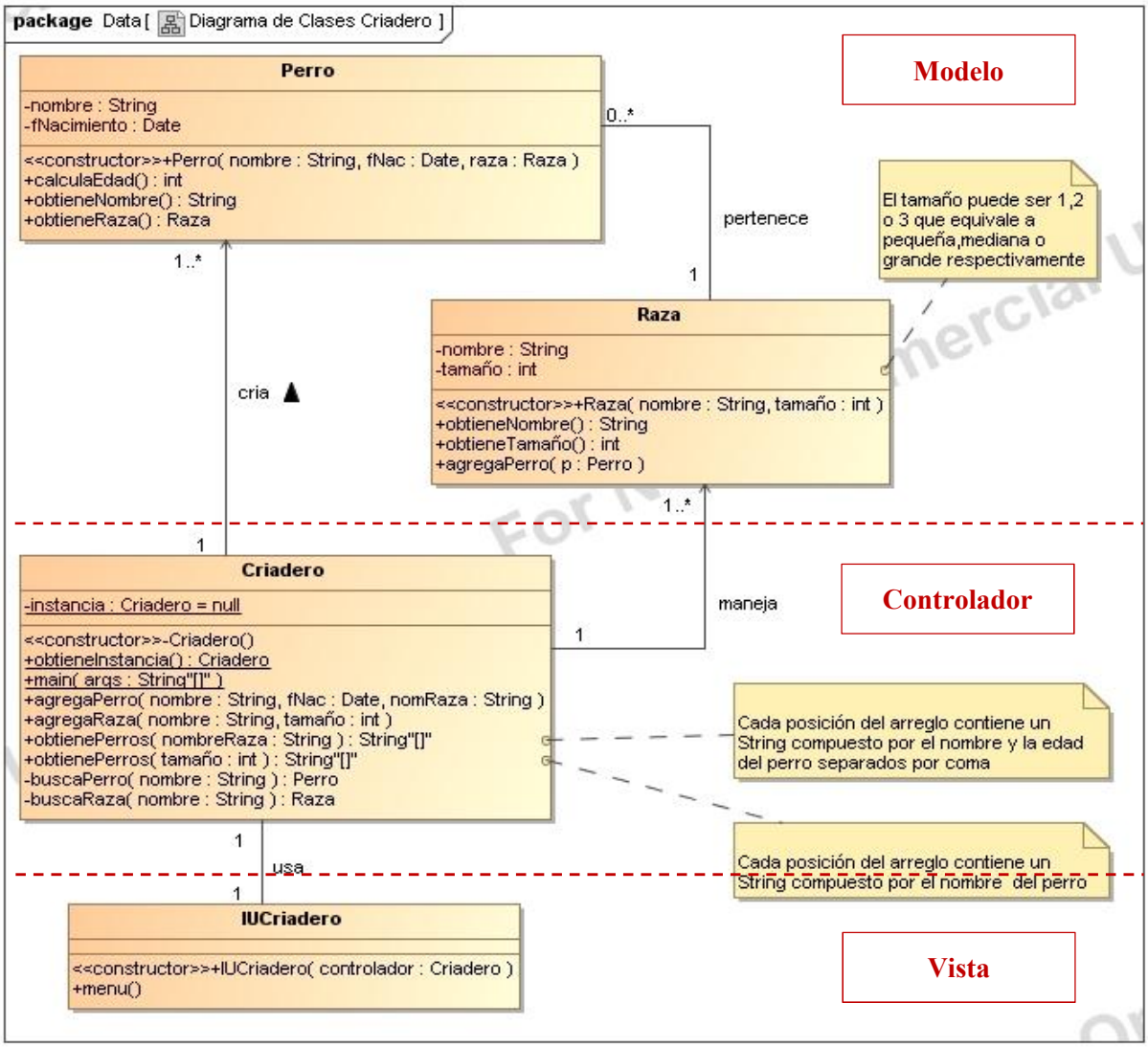

Figura 5.7. Diagrama de clases de un Sistema Criadero de Perros.