

# Terminando

- [Terminando](#)
- [Taller con POO](#)
- [Identificación de las clases](#)
- [Jerarquía de clases](#)
- [Codificación](#)
- [Interfaz gráfica de usuario](#)
- [Calculadora con interfaz gráfica de usuario y ejecutable](#)
- [Pintemos en Java](#)
- [Bibliografía recomendada](#)
- [Código utilizado en los ejemplos](#)
- [Tarea](#)
- [Voluntario: Muestra tus proyectos](#)
- [Créditos](#)

# Terminando

El objetivo de este último módulo es facilitarte mas recursos. Por ello voy a realizar 1 ejemplo completo desde su fase de análisis hasta su fase de codificación tratando de explicarte aquellas partes mas relevantes. También voy a ver unas pinceladas sobre como crear una interfaz gráfica de usuario, facilitarte algo de bibliografía y, como siempre, dejarte todo el código completo y una tarea.

# Taller con POO

Antes de ponernos a hacer nada vamos a definir nuestro problema.

“ La cooperativa CATEDUSC nos ha solicitado la elaboración de un programa informático para la gestión de sus talleres. En estos talleres se arreglan vehículos. Estos vehículos pueden ser coches o motos.

De cada taller se quiere almacenar su nombre, dirección, propietario y un listado de clientes.

De los propietarios queremos almacenar su DNI, nombre, primer apellido y dirección.

De cada cliente queremos almacenar su DNI, nombre, primer apellido, teléfono y un listado de sus vehículos.

De los vehículos queremos almacenar su matrícula, marca y modelo. De los coches queremos almacenar su anchura y altura (en centímetros) y de las motos si lleva limitador o no.

Los propietarios, clientes y vehículos deberán implementar una funcionalidad que los haga identificarse.

CATEDUSC nos ha dejado claro que quiere que utilicemos el lenguaje de programación Java ya que sus técnicas se encargarán posteriormente del mantenimiento. Además quieren que trabajemos con programación orientada a objetos y que hagamos un buen uso de sus características. Por supuesto quieren que el código que se les entregue esté documentado tanto a nivel de documentación como a nivel interno y que se sigan los convenios de la POO en Java en cuanto a la creación de nombres. En caso de no cumplir lo anterior el contrato quedará rescindido y el trabajo no será abonado. Valorarán la creación de un menú textual para el manejo del software.

Ahora que tenemos definido nuestro programa es el momento de ponernos a pensar antes de empezar a codificar. Para ello en los siguientes capítulos empezaremos por definir las clases que hemos detectado y las relaciones existentes entre ellas. Una vez que tengamos definido lo anterior



será el momento de empezar a escribir el código fuente.

# Identificación de las clases

Mi consejo para los primeros programas en los que vayáis a trabajar con programación orientada a objetos es que os imprimáis el enunciado del problema y con un lápiz pongáis un recuadro sobre aquellos elementos que defináis como un objeto y subrayéis aquellos elementos que identifiquéis como atributos del mismo.

La cooperativa CATEDUSC nos ha solicitado la elaboración de un programa informático para la gestión de sus talleres. En estos talleres se arreglan vehículos. Estos vehículos pueden ser coches o motos.

De cada taller se quiere almacenar su nombre, dirección, propietario y un listado de clientes.

De los propietarios queremos almacenar su DNI, nombre, primer apellido y dirección.

De cada cliente queremos almacenar su DNI, nombre, primer apellido, teléfono y un listado de sus vehículos.

De los vehículos queremos almacenar su matrícula, marca y modelo. De los coches queremos almacenar su anchura y altura (en centímetros) y de las motos si lleva limitador o no.

Los propietarios, clientes y vehículos deberán implementar una funcionalidad que los haga identificarse.

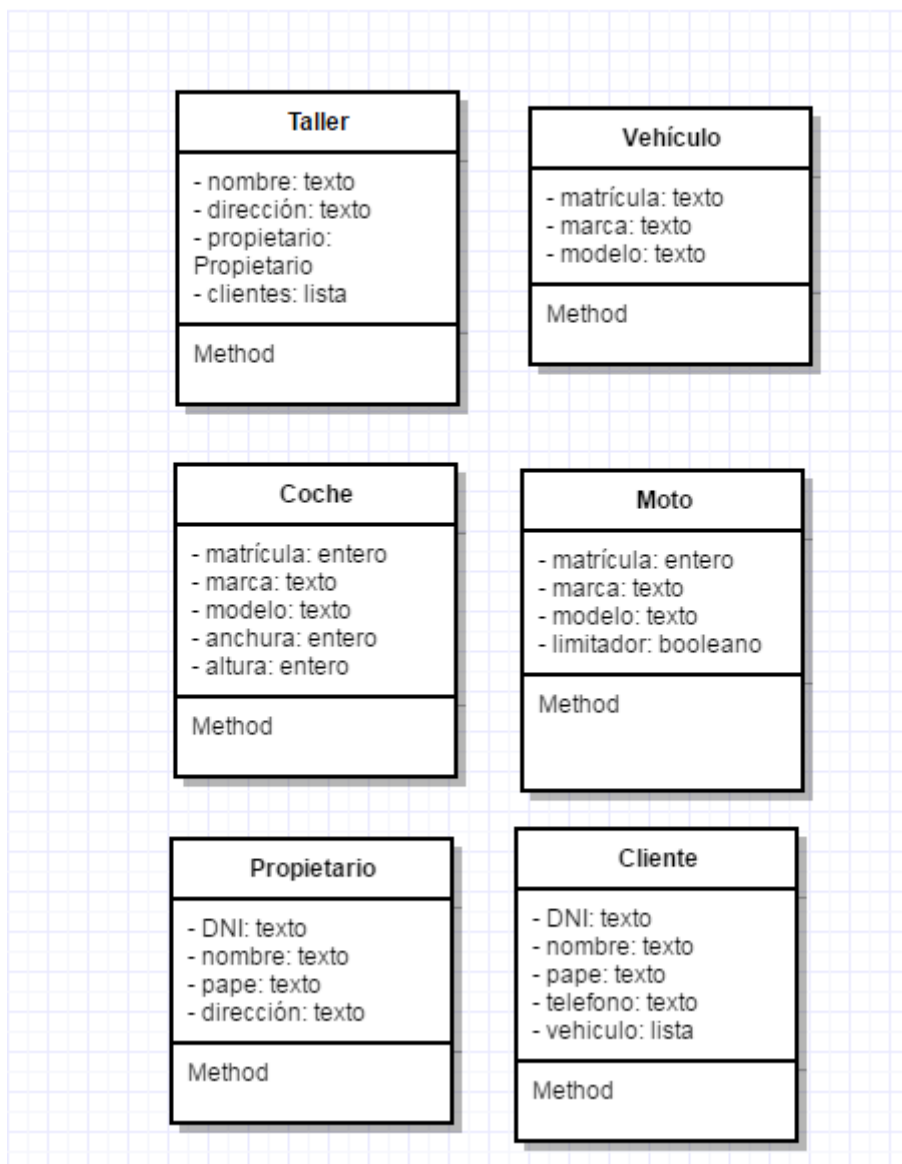
CATEDUSC nos ha dejado claro que quiere que utilicemos el lenguaje de programación Java ya que sus técnicas se encargarán posteriormente del mantenimiento. Además quieren que trabajemos con programación orientada a objetos y que hagamos un buen uso de sus características. Por supuesto quieren que el código que se les entregue esté documentado tanto a nivel de documentación como a nivel interno y que se sigan los convenios de la POO en Java en cuanto a la creación de nombres. En caso de no cumplir lo anterior el contrato quedará rescindido y el trabajo no será abonado. Valorarán la creación de un menú textual para el manejo del software.



En la imagen superior vemos que identifico las clases Taller, Vehículo, Coche, Moto, Propietario y Cliente y una serie de atributos para clase. He subrayado del mismo color los atributos que corresponden a cada clase, la cual he recuadrado en ese mismo color.

Una vez hemos identificado las clases y sus atributos lo ideal sería utilizar algún modo estandar de representarlo. Yo he optado por utilizar un [lenguaje unificado de modelado \(UML\)](#) para esta tarea y concretamente lo relativo a los [diagramas de clases](#) que es la tarea que nos ocupa. Con UML cada clase estará contenida en un recuadro que dividiré horizontalmente en 3 recuadros. En el recuadro superior escribiré el nombre de la clase, en el recuadro inferior los atributos de la clase y en el recuadro inferior los métodos de la clase. Vamos a ver como quedarían nuestras clases dibujadas de este modo.

Gliffy / \*untitled 🔒





En la imagen superior vemos el modo en que dibujaríamos las clases de nuestro problema. El campo para los métodos de momento lo he dejado vacío. En los atributos, antes de su nombre, he puesto el símbolo - para indicar que se trata de un atributo privado y cumplir así los criterios de encapsulación cuando cree los getters y setters correspondientes.

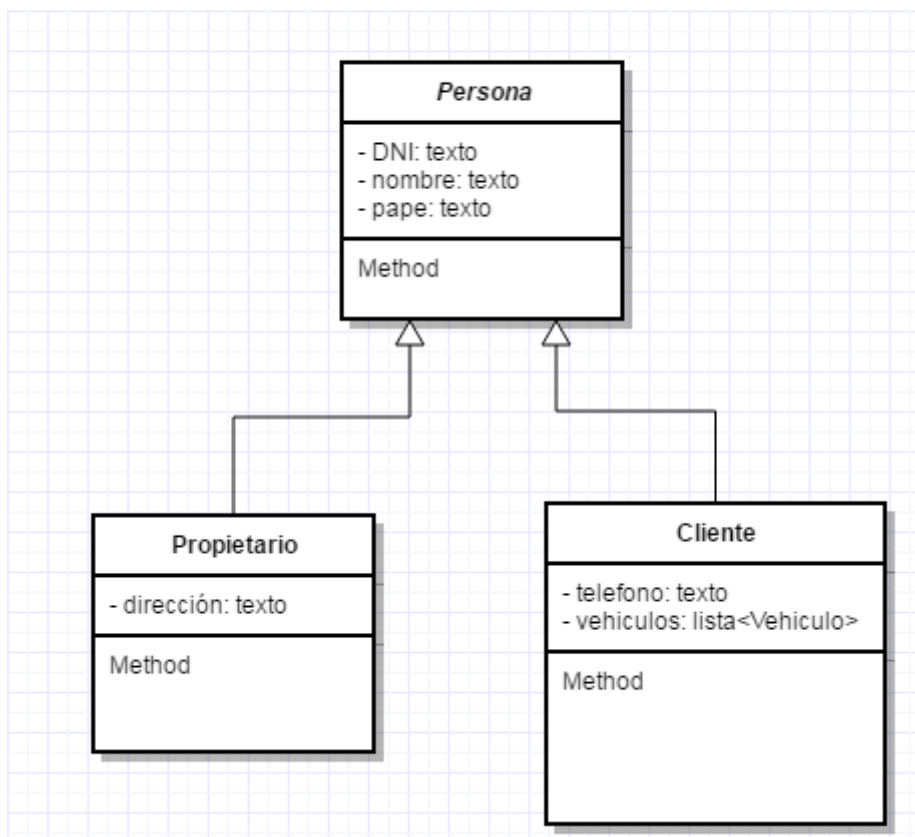
Podemos crear diagramas de clases con la aplicación web <https://www.gliffy.com/uses/uml-software/>

En el siguiente apartado vamos a redefinir este diagrama para ver si de algún modo existe herencia y las distintas relaciones entre las clases.

# Jerarquía de clases

Si analizamos el diagrama de clases que hemos realizado en el apartado anterior observamos que los clientes y los propietarios comparten parte de sus atributos por lo que podríamos crear una abstracción que llamásemos Persona que tuviese estos campos y que posteriormente Propietario y Cliente heredasen de ella. Vamos a ver como quedaría.

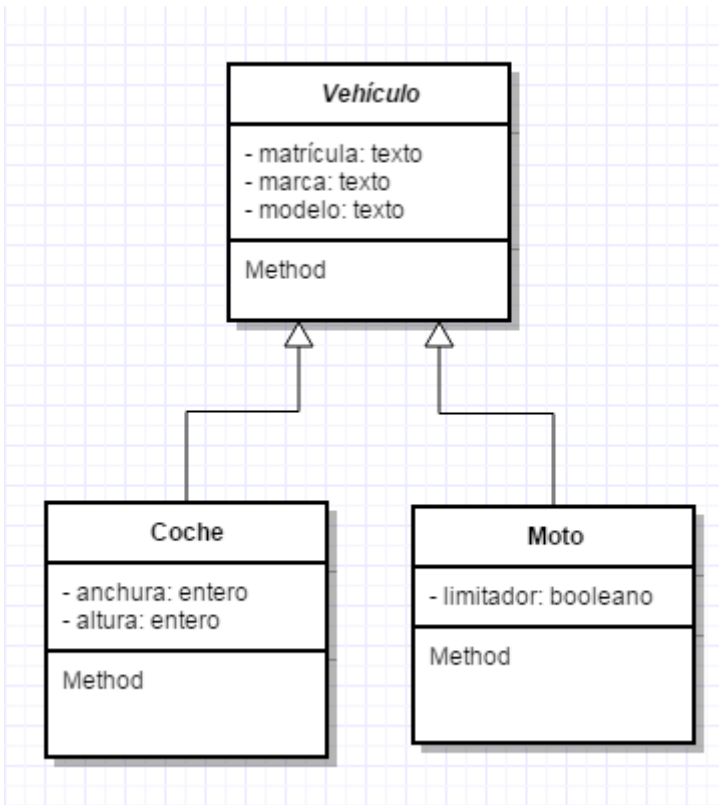
Gliffy / \*untitled 🔒



Fíjate que en la imagen anterior la clase Persona, al considerarla abstracta, escribimos su nombre con letra cursiva. Consideramos la clase Persona abstracta porque no vamos a tener que crear en ningún momento instancias de de ella.

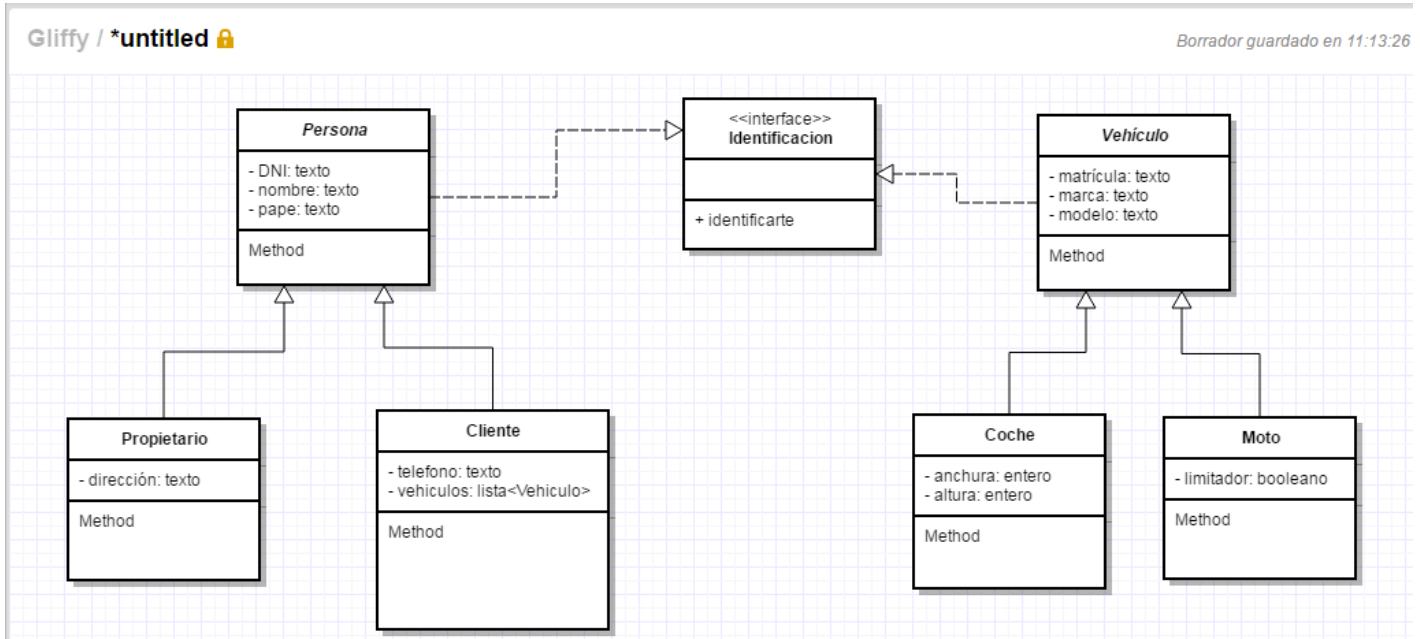
De modo análogo ocurre con Vehículo, Coche y Moto. A fin de cuentas comparten una serie de atributos, podemos "sacar factor común" haciendo que Coche y Moto herenden de Vehículo tal y como vemos en la imagen posterior.





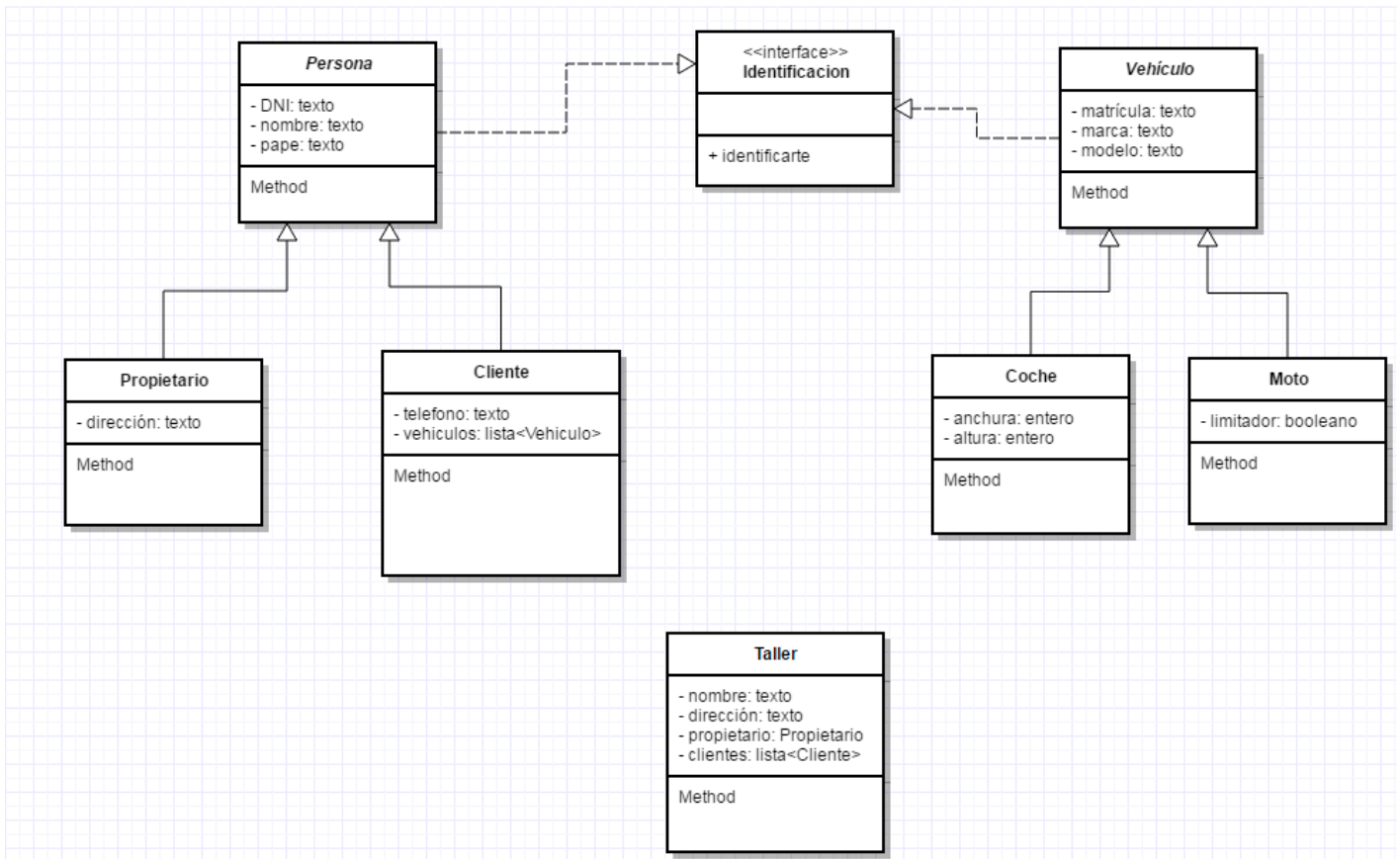
Igual que antes Vehículo es una clase abstracta y como tal escribimos su nombre en cursiva. Coche y Moto extenderán a Vehículo.

En el enunciado de nuestro problema se dice "Los propietarios, clientes y vehículos deberán implementar una funcionalidad que los haga identificarse.", esto debería ser un método pero ¿en que clase lo coloco? Es algo que tanto las personas como los vehículos deben implentar ¿pongo el método una vez en cada clase? La solución a nuestras dudas es crear una interface que vamos a llamar Identificación y haremos que tanto Persona como Vehículo implementen esta clase. Vamos a ver como dibujaríamos esta interface y la relación entre ella, Persona y Vehículo.



Observa que la línea entre las clases es en este caso discontinua mientras que cuando expresamos herencia es una línea continua.

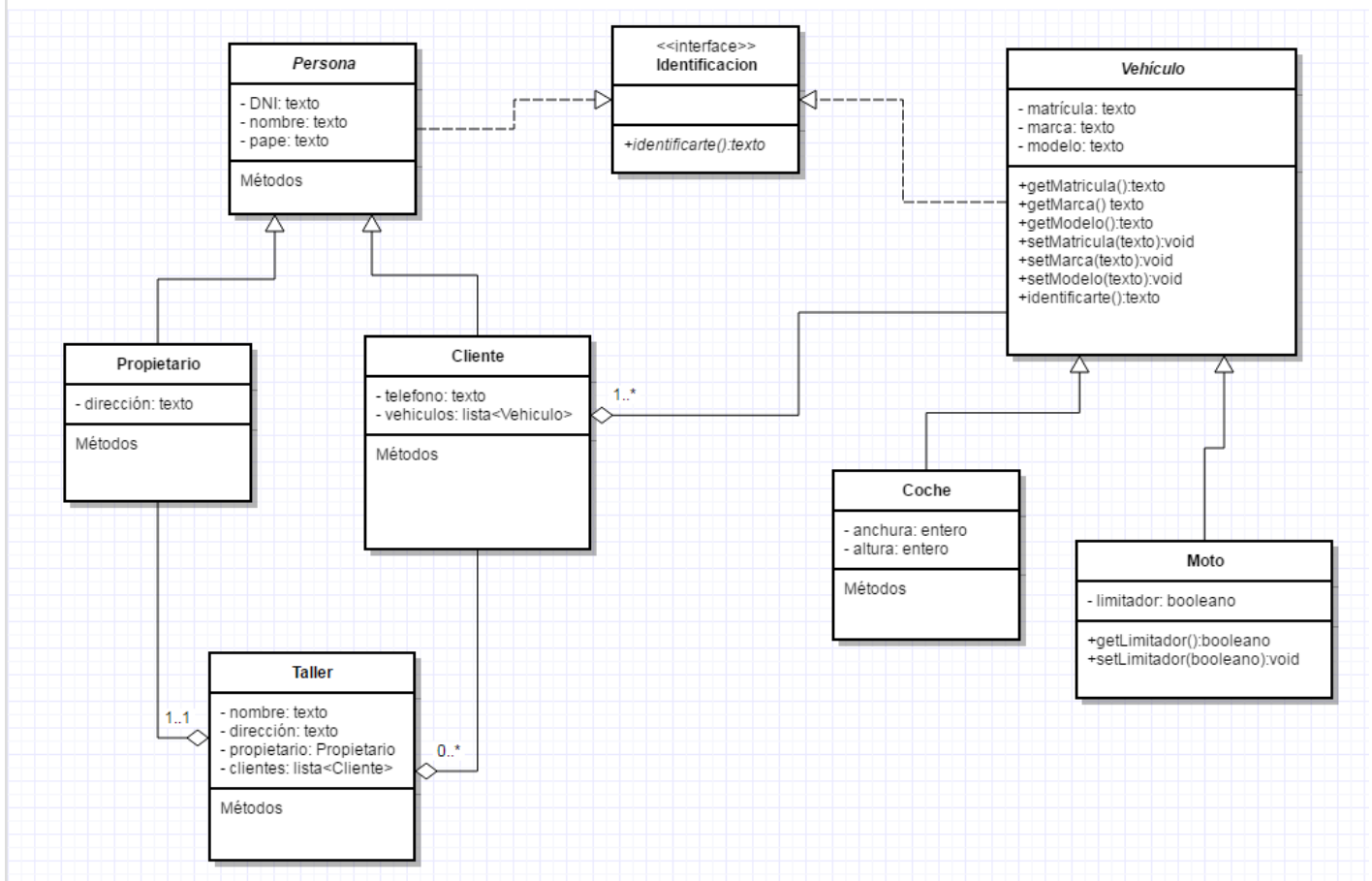
Si comparamos nuestro esquema actual con nuestro primer esquema observaremos que nos falta la clase Taller así que vamos a añadirla a nuestro esquema.



Casi hemos terminado. De algún modo tenemos que expresar la composición entre clases y la cardinalidad existente. Es decir, si nos fijamos en la clase Cliente vemos que está compuesta por una lista de Vehículos, tenemos que expresarlo de algún modo, además, cada cliente deberá tener al menos 1 vehículo. Si nos fijamos en Taller vemos que está compuesto tanto por Propietario (cada taller tiene 1 único propietario) como por una lista de clientes que serán 0 o mas. Vamos a ver como representar esta composición y cardinalidad.

Gliffy / \*untitled

Borrador guardado en 11:34:07



Con esto tendríamos terminado nuestro esquema de clases donde podemos observar la jerarquía de clases existente y las relaciones que existen entre ellas. Faltaría añadir en cada clase los métodos que tiene donde pone Métodos. Únicamente los he añadido en las clases Identificacion, Vehículo y Moto.

Con esto únicamente nos quedaría codificar nuestro esquema, pero esto lo haremos en el siguiente capítulo.

# Codificación

Una vez que tenemos esquematizada la solución de nuestro programa es el momento de codificarla. En esta sección voy a añadir imágenes de las partes que considero mas relevantes para solucionar nuestro problema. En el apartado "Código utilizado en los ejemplos" podréis descargar el código completo del proyecto.

```
1 package catedusc.taller.interfaces;
2
3 /**
4  * @author Pablo Ruiz Soria
5  */
6 public interface Identificacion {
7     public String identificarte();
8 }
```

El código anterior es la implementación de la interface Identificación. En ella vemos que aparece la firma de la función identificarte. Como ya sabemos a estas alturas todas las clases que implementen esta interface estarán obligadas a implementar esta función.

Vamos a contnuar con la clase abstracta Persona que implementa la interface anterior.



```

1  package catedusc.taller.humano;
2
3  import catedusc.taller.interfaces.Identificacion;
4  import catedusc.taller.util.Comprobador;
5
6  /**
7   * @author Pablo Ruiz Soria
8   */
9  public abstract class Persona implements Identificacion {
10     private String dni;
11     private String nombre;
12     private String pape;
13
14     public Persona(String dni, String nombre, String pape) throws Exception{
15         if(Comprobador.esValidoDNI(dni) ){
16             this.dni = dni;
17         }else{
18             throw new Exception("DNI inválido");
19         }
20         this.nombre = nombre;
21         this.pape = pape;
22     }
23

```

Como vemos en esta clase indicamos que es una clase pública y abstracta que implementa la interface Identificacion. En esta clase tenemos 3 atributos que siguen los principios de la encapsulación y en la imagen vemos que la clase tiene un único constructor que obliga a definir una Persona a través de 3 argumentos. Como ya sabemos, al tratarse de una clase abstracta no podremos crear objetos de esta clase. La clase Vehículo sería similar a esta clase que acabamos de analizar.

Vamos a continuar con el código de la clase Propietario que hereda de la clase anterior y se verá obligada a implementar el método de la interface Identificacion.



```

1  package catedusc.taller.humano;
2
3  /**
4   * @author Pablo Ruiz Soria
5   */
6  public class Propietario extends Persona {
7      private String direccion;
8
9      public Propietario(String dni, String nombre, String pape, String direccion)
10         super(dni, nombre, pape);
11         this.direccion = direccion;
12     }
13
14     /**
15      * @return the direccion
16      */
17     public String getDireccion() {
18         return direccion;
19     }
20
21     /**
22      * @param direccion the direccion to set
23      */
24     public void setDireccion(String direccion) {
25         this.direccion = direccion;
26     }
27
28     @Override
29     public String identificarte() {
30         return "Soy un propietario. " + super.identificarte() + " y mi dirección
31             + this.getDireccion();
32     }
33 }

```

Vemos en la línea 6 que se trata de una clase pública que extiende a la clase Persona (que a su vez implementaba a Identificación). Al heredar de Persona dispondremos de acceso a aquellos atributos y métodos no privados. Si nos fijamos en el constructor vemos que hemos de llamar a super (línea 10) para que se llame al constructor de la superclase de Propietario (Persona). Observamos también en las líneas 28 a 32 la implementación del método de la interface. Del mismo modo que construimos la clase Propietario procederíamos con la clase Cliente.

Una vez hemos creado todas nuestras clases únicamente necesitaríamos disponer de una clase con un método main que nos permitiese interactuar con las clases que hemos definido para poder crear objetos a partir de ellas. En el apartado "Código utilizado en los ejemplos" he añadido 2 clases que nos permiten lanzar la aplicación, la clase Inicio.java contiene una interfaz de tipo texto y la clase igu contiene una interfaz gráfica de usuario limitada. El código se encuentra documentado para facilitar su comprensión.



En el siguiente apartado vamos a ver como crear esta interfaz gráfica de usuario a la que hago referencia en el párrafo anterior.



# Interfaz gráfica de usuario

Hasta el momento siempre que hemos introducido algún dato lo hemos hecho a través del código de nuestros programas pero lo habitual es que los programas tengan una [interfaz gráfica de usuario](#) (IGU, GUI en inglés) con la que nosotros interactuemos y de ese modo introduzcamos u obtengamos datos. Por ello he preparado el siguiente videotutorial en el que muestro como crear una IGU básica por si queremos incorporarla a nuestros programas.

[https://www.youtube.com/embed/jQwbk5A3Lxc?list=PL1ubUMkNBAR\\_98ka0Q393l3fpzSjo3FGq](https://www.youtube.com/embed/jQwbk5A3Lxc?list=PL1ubUMkNBAR_98ka0Q393l3fpzSjo3FGq)

# Calculadora con interfaz gráfica de usuario y ejecutable

En este apartado vamos a ver como elaborar una calculadora que tenga una interfaz gráfica de usuario y como generar un "fichero ejecutable" que nos permita ejecutar nuestros proyectos en cualquier sistema operativo que disponga de la JRE. Os dejo el vídeo a continuación:

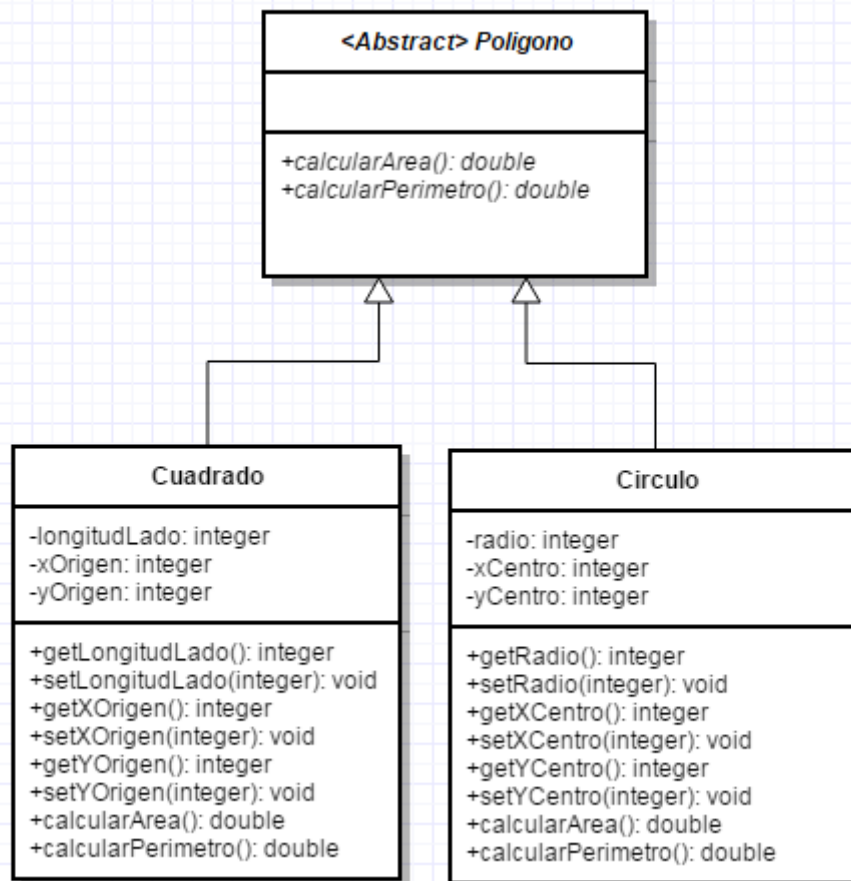
[https://www.youtube.com/embed/9xKSFonCYB8?list=PL1ubUMkNBAR\\_98ka0Q393l3fpzSjo3FGq](https://www.youtube.com/embed/9xKSFonCYB8?list=PL1ubUMkNBAR_98ka0Q393l3fpzSjo3FGq)

# Pintemos en Java

En este vídeo vamos a trabajar con las características de la POO (clases abstractas, herencia, polimorfismo) y vamos a hacerlo a través de una interfaz gráfica de usuario. Además, en esta interfaz gráfica de usuario, vamos a dibujar los objetos con los que vamos a trabajar. El enunciado de nuestro problema dice:

“ Para trabajar la unidad didáctica de geometría vamos a desarrollar una aplicación informática que nos permita trabajar con polígonos, concretamente queremos trabajar con cuadrados y círculos. De cualquier polígono queremos poder calcular su área y perímetro. De los cuadrados queremos almacenar su longitud de lado y el punto (x e y) que lo define en su posición superior izquierda. De los círculos queremos almacenar su centro (x e y) y su radio. En la aplicación deberemos de poder crear cualquiera de estos 2 elementos, listarlos (viendo el área de los cuadrados y el perímetro de los círculos) y dibujar todos los polígonos que tengamos. Todo hecho deberemos hacerlo haciendo un buen uso de las características de la POO. Por simplicidad, no es necesario controlar las excepciones ni que los valores de los lados son positivos.

El diagrama de clases resultante del enunciado anterior sería el siguiente:



Vamos a ver como implementarlo en el siguiente vídeo:

<https://www.youtube.com/embed/o330-623xrl>

# Bibliografía recomendada

Soy consciente de que quizás el curso no haya resuelto todas tus dudas acerca de la programación orientada a objetos y el lenguaje de programación Java puesto que se pretende que el mismo sea una introducción, por ello no querría dejar pasar la ocasión de recomendarte un par de libros que creo pueden resultarte de utilidad además de la gran cantidad de información existente en internet.

El primero de los libros se titula "Piensa en Java" escrito por Bruce Eckel. Aunque la última edición data de 2006 sigue siendo un libro de referencia para quienes quieren iniciarse en la programación orientada a objetos y el lenguaje de programación Java. En su [página oficial \(inglés\)](#) podemos encontrar mas información sobre el mismo.

El segundo libro es un libro de programación avanzada orientado a la ingeniería del software. El libro es interesante ya que nos proporciona una buena visión sobre buenas prácticas y nos plantea reflexiones que nos harán mejorar en el modo en que escribimos nuestro código haciéndonos mas eficientes. Su título es "Código limpio" y está escrito por Robert C. Martin. Esta es su [página web](#) para la edición en castellano.

# Código utilizado en los ejemplos

[Módulo 4 Taller](#) (zip - 0.04 MB).

# Tarea

La cooperativa de ebanistas de CATEDU nos ha indicado que necesita una aplicación informática para la gestión de su ebanistería. En esta ebanistería fabrican muebles. De los muebles quieren guardarse sus dimensiones (ancho, alto y profundo) en centímetros como números enteros. Los tipos de muebles que fabrican son mesas de oficina, mesas de taller y estanterías. De todas las mesas se quiere conocer si tienen cajonera o no pero de las mesas de oficina se quiere conocer el material con el que están hechas mientras que de las mesas de taller se quiere almacenar su grado de resistencia (un número del 1 al 10). De las estanterías se quiere guardar el número de baldas.

A partir del enunciado anterior tu tarea consiste en:

- Crear un diagrama de clases en el que se reflejen las clases de nuestro problema así como los atributos de las mismas y las relaciones de herencia y composición existentes en caso de haberlas
- Implementar con Java el diagrama que has realizado

# Voluntario: Muestra tus proyectos

<https://padlet.com/embed/fljmvkdbwmh6>

Hecho con Padlet



# Créditos

## Autoría

- {{ book.author }}

---

Cualquier observación o detección de error en [soporte.catedu.es](https://soporte.catedu.es)

Los contenidos se distribuyen bajo licencia **Creative Commons** tipo **BY-NC-SA** excepto en los párrafos que se indique lo contrario.



**GOBIERNO  
DE ARAGON**

Departamento de Educación,  
Cultura y Deporte

**CATEDU**   
CENTRO ARAGONÉS de TECNOLOGÍAS para la EDUCACIÓN

