

Tema 5.

Análisis y diseño orientado a objetos.

Diagramas de clases.

Contenidos:

1. Introducción al Lenguaje UML.
2. Elaboración de diagramas de clases
 - 2.1 -Objetos
 - 2.2 -Diagrama de Clases
 - Clases. Atributos, métodos y visibilidad.
 - Relaciones.
 - 2.3. Herramientas de diseño de diagramas.
3. Generación de código a partir del diagrama de clases.
4. Ingeniería inversa.

Introducción al diseño orientado a objetos

1. Introducción al UML

El lenguaje de modelado unificado (UML) es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software. Este lenguaje se puede utilizar para modelar sistemas de software, de hardware u organizaciones del mundo real. Para ello utiliza una serie de diagramas en los que se representan distintos puntos de vista de modelado.

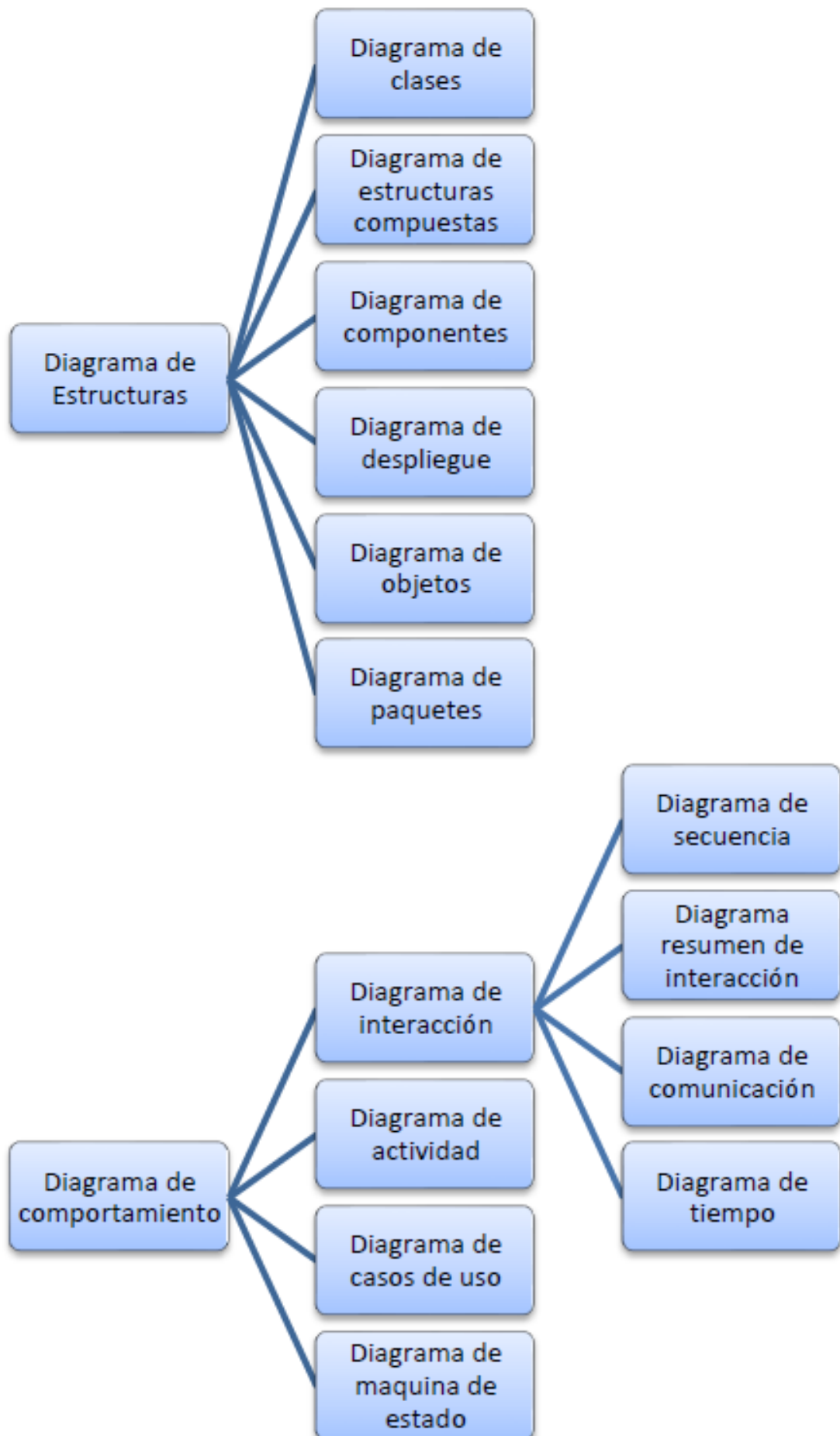
Podemos decir que UML es un lenguaje que se utiliza para documentar.

Existen dos grandes versiones de UML:

- **UML 1.x:** desde finales de los 90 se empezó a trabajar con el estándar UML. En los años sucesivos fueron apareciendo nuevas versiones que introducían mejoras o ampliaban a las anteriores.
- **UML 2.x:** en torno a 2005 se difundió una nueva versión de UML.

UML 2.0 define 13 tipos de diagramas, divididos en tres categorías: 6 tipos de diagramas representan la estructura estática de la aplicación o del sistema, 3 representan tipos generales de comportamiento y 4 representan diferentes aspectos de las interacciones:

- **Diagramas de estructura** (parte estática del modelo): incluyen el diagrama de clases, diagrama de objetos, diagrama de componentes, diagrama de estructura compuesta, diagrama de paquetes y diagrama de implementación o despliegue.
- **Diagramas de comportamiento** (parte dinámica del modelo): incluyen el diagrama de casos de uso, diagrama de actividad y diagrama de estado.
- **Diagramas de interacción:** todos los derivados del diagrama de comportamiento en general. Incluyen el diagrama de secuencia, diagrama de comunicación, diagrama de tiempos y diagrama de vista de interacción. Se centran en el flujo de control y de datos entre los elementos del sistema modelado.



2. Elaboración de diagramas de clases

2.1. Objetos

En esta primera parte, estudiaremos los conceptos básicos de la tecnología orientada a **objetos**: **los diagramas de clases**. Entendemos por **diagrama de clases** una representación gráfica y estática de la estructura general de un sistema, mostrando cada una de las **clases** y sus **interacciones** representadas en forma de bloques. Es decir, este diagrama sirve para **visualizar las relaciones entre las clases que componen el sistema**.

A lo largo de esta unidad usaremos herramientas para crear diagramas de clases y para generar código a partir de diagramas.

En el **diseño orientado a objetos** un sistema se entiende como un conjunto de objetos que tienen propiedades y comportamientos.

Un **objeto** consta de una estructura de datos y métodos u operaciones que manipulan dichos datos. Los datos definidos dentro del objeto son sus **atributos**. Los objetos se comunican unos con otros a través del paso de mensajes.

Por tanto los **objetos** son aquello que tiene **estado** (propiedades más valores), **comportamiento** (acciones y reacciones a mensajes) **e identidad** (propiedad que lo distingue de los demás objetos). La estructura y comportamiento de objetos similares tendrán estarán definidos en su clase común.

Una **clase** es una **plantilla para la creación de objetos**, cuando se crea un objeto se ha de especificar de qué clase es el objeto instanciado. Así una clase es un conjunto de objetos que comparten estructura y comportamiento común. Ejemplos:

Clase:
Coche

Clase Coche

arrancar, ir, parar, girar

color, velocidad, carburante

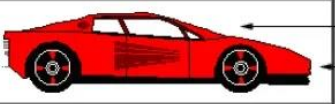
nombre de la clase

métodos (funciones)

atributos (datos)

♦ Objeto: Ferrari


coche.ferrari




nombre del objeto

métodos
arrancar, ir, parar, girar


datos
rojo, 280 km/h, lleno



Cortador de galletas.
(La Clase)



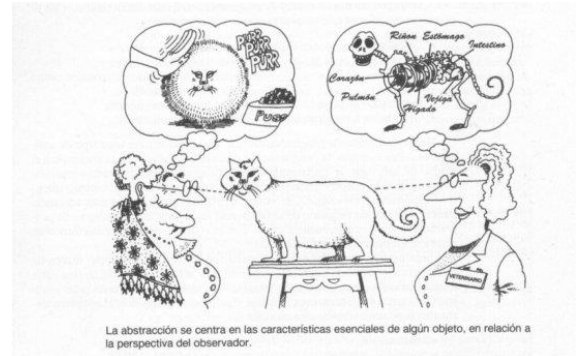
Las galletas
(Los objetos)

	Mundo Real	En OOP
<div>Clase Generalización de características (atributos y comportamientos)</div>	Perro Raza, Color, Edad, Corre,	Clase Define datos y métodos
<div>Objeto Instancia de una clase distinguible por sus características específicas</div>	Tino Pastor Alemán Marrón 7 meses Veloz	Objetos Ocupa espacio, se crea y se destruye

Las propiedades de los objetos son claves.

Los principios del modelo OO (orientado a objetos) son:

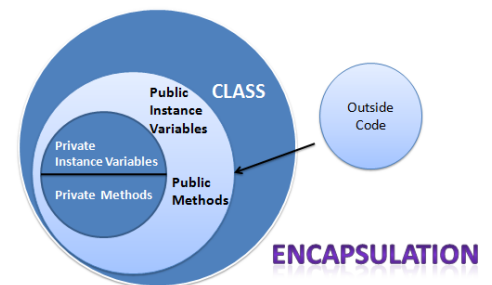
– **Abstracción:** características esenciales de un objeto, donde se capturan sus comportamientos. Se pretende conseguir una descripción formal. La abstracción es clave en el proceso y análisis del diseño OO, ya que podremos crear un conjunto de clases que modelen la realidad del problema a resolver.



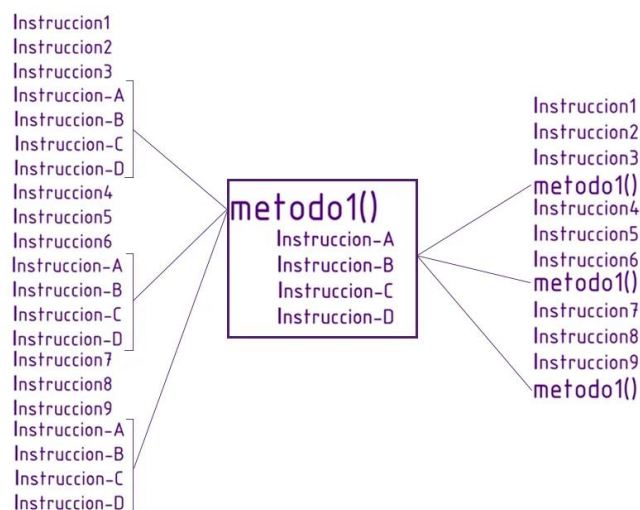
Por ejemplo, podemos crear una clase abstracta llamada "Vehicle" que tenga propiedades como "brand" y "model" y métodos como "start" y "stop". Esta clase abstracta no puede ser instanciada por sí misma, pero puede ser extendida por otras clases más específicas como "Car" o "Motorcycle", que pueden implementar los métodos y propiedades de la clase "Vehicle" de manera específica para cada tipo de vehículo.

La abstracción nos permite crear un modelo conceptual de un objeto o concepto y luego utilizar ese modelo para crear objetos más específicos que heredan las características y comportamientos del modelo conceptual. Esto nos permite reutilizar código y simplificar la creación de objetos complejos.

– **Encapsulación:** oculta detalles que no aporta a sus características esenciales, es decir, separa la parte interna inaccesible para otros objetos, de la externa, que sí será accesible. Dicho de otra forma, **oculta los métodos y atributos a otros objetos pasando a ser privados.**



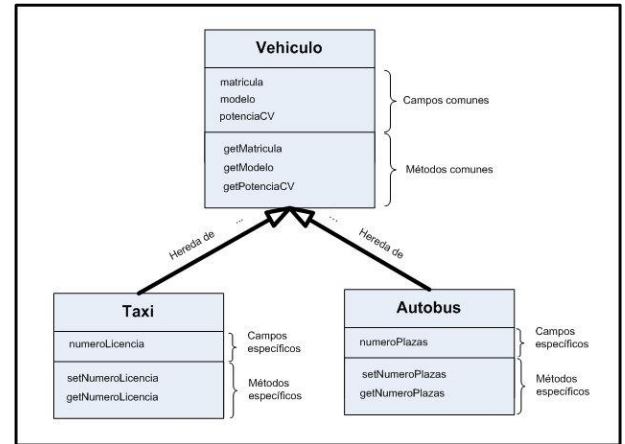
– **Modularidad:** es la capacidad de un sistema o aplicación para **dividirse en pequeños módulos independientes.**



Código con redundancias

Código sin redundancias

– **Jerarquía o herencia:** propiedad que permite que algunas clases tengan propiedades y características de una clase superior. La clase principal se llama clase padre o **superclase**. La nueva clase se denominará clase hija o **subclase**, que heredará todos los métodos y atributos de la superclase. Cada subclase estará formada con objetos más especializados.



– **Polimorfismo:** reunir con el mismo nombre comportamientos diferentes.

Cuando dos instancias u objetos, pertenecientes a distintas clases, pueden responder a la llamada a métodos del mismo nombre, cada uno de ellos con distinto comportamiento encapsulado.

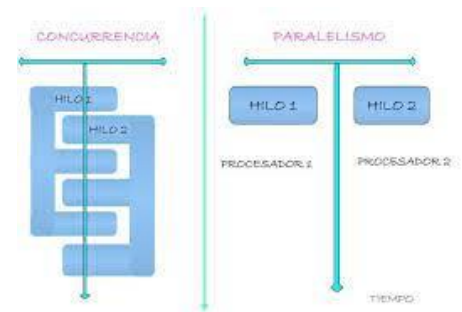
Ejemplo El método hablar puede ser sobrescrito por cualquier entidad (humano, perro, pato, gato).

<https://jarroba.com/polimorfismo-en-java-parte-i-con-ejemplos/>



– **Tipificación:** definición precisa de un objeto de forma que no pueda ser intercambiada.

– **Concurrencia:** propiedad que distingue un objeto activo de otro que no lo está.



– **Persistencia:** propiedad de un objeto a través de la cual su existencia trasciende en el tiempo y/o el espacio. (ej. Objetos de clases asociadas a Bases de datos)

Se llama “persistencia” de los objetos a su capacidad para guardarse y recuperarse desde un medio de almacenamiento. La persistencia en Base de Datos relacionales se suele implementar mediante el desarrollo de funcionalidad específica utilizando la tecnología JDBC o mediante frameworks que automatizan el proceso a partir de mapeos (conocidos como Object Relational Mapping, ORM) como es el caso de Hibernate.

Relational database (such as PostgreSQL or MySQL)

ID	FIRST_NAME	LAST_NAME	PHONE
1	John	Connor	+16105551234
2	Matt	Makai	+12025555689
3	Sarah	Smith	+19735554512
...

ORMs provide a bridge between relational database tables, relationships and fields and Python objects

Python objects

```
class Person:
    first_name = "John"
    last_name = "Connor"
    phone_number = "+16105551234"

class Person:
    first_name = "Matt"
    last_name = "Makai"
    phone_number = "+12025555689"

class Person:
    first_name = "Sarah"
    last_name = "Smith"
    phone_number = "+19735554512"
```

2.2. Diagramas de clases.

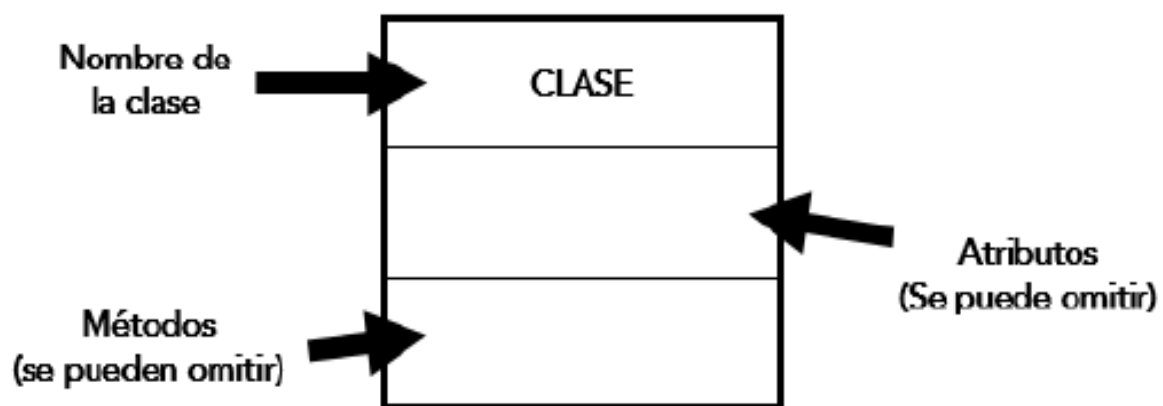
Como ya hemos comentado, un diagrama de clases es una representación gráfica y estática de la estructura general de un sistema, mostrando cada una de las clases y sus interacciones representadas en forma de bloques. Vamos a poder visualizar las relaciones que existen entre las clases.

Un **diagrama de clases** está compuesto por:

- **Clases** (atributos, métodos y visibilidad),
- **Relaciones** (asociación, herencia, agregación, composición, realización y dependencia)

2.2.1. Clases

Son la unidad básica que **contendrá toda la información referente a un objeto** (un objeto es una instancia de una clase). A través de ella se podrá modelar el entorno en estudio. **Una clase en UML podemos representarla con un rectángulo dividido en 3 partes:**

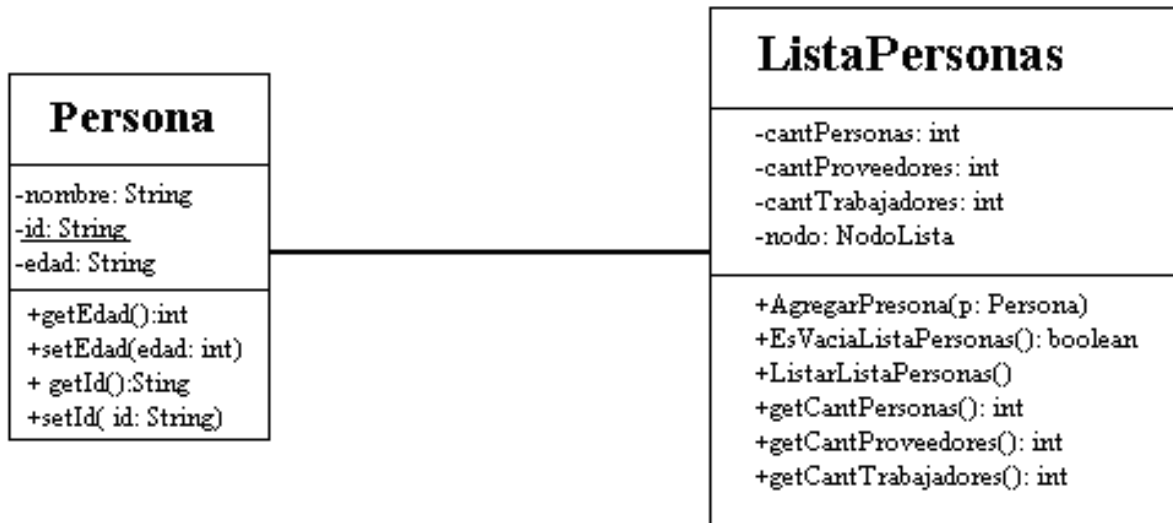


– Parte superior: nombre de la clase.

– Parte central: atributos, que caracterizan la clase (*private*, *protected*, *package* o *public*).

– Parte inferior: métodos u operaciones, forma de interactuar del objeto con el entorno (según visibilidad: *private*, *protected*, *package* o *public*).

Al representar la clase podemos eliminar los métodos y atributos. **La visibilidad de los métodos debe ser por defecto *public* y la de los atributos, *private*.**




• ATRIBUTOS

Representan las propiedades de la clase. Se pueden representar solo con el nombre o también incluyendo su nombre, tipo y valor por defecto. Al crear los atributos indicaremos el tipo de dato. En UML los más básicos son: *Integer*, *String* y *Boolean*.

Cuando creamos el atributo hemos de indicar también su visibilidad con el entorno, estando estrechamente relacionada con el encapsulamiento.

Se pueden distinguir los siguientes tipos:

- **public:** el atributo será público, visible tanto fuera como dentro de la clase. Se representa con el signo “+”.
- **private:** el atributo solo será accesible dentro de la clase (solo sus métodos). Se representa con signo “-”.
- **protected:** el atributo no será accesible desde fuera de la clase, pero sí por métodos de la clase y de subclases. Se representa con “#”.
- **package:** el atributo será visible en las clases del mismo paquete. Se representa con “~”. (Los paquetes son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia.)



		Mismo paquete		Otro paquete	
		Subclase	Otra	Subclase	Otra
-	private	<i>no</i>	<i>no</i>	<i>no</i>	<i>no</i>
#	protected	<i>sí</i>	<i>sí</i>	<i>sí</i>	<i>no</i>
+	public	<i>sí</i>	<i>sí</i>	<i>sí</i>	<i>sí</i>
~	<i>package</i>	<i>sí</i>	<i>sí</i>	<i>no</i>	<i>no</i>

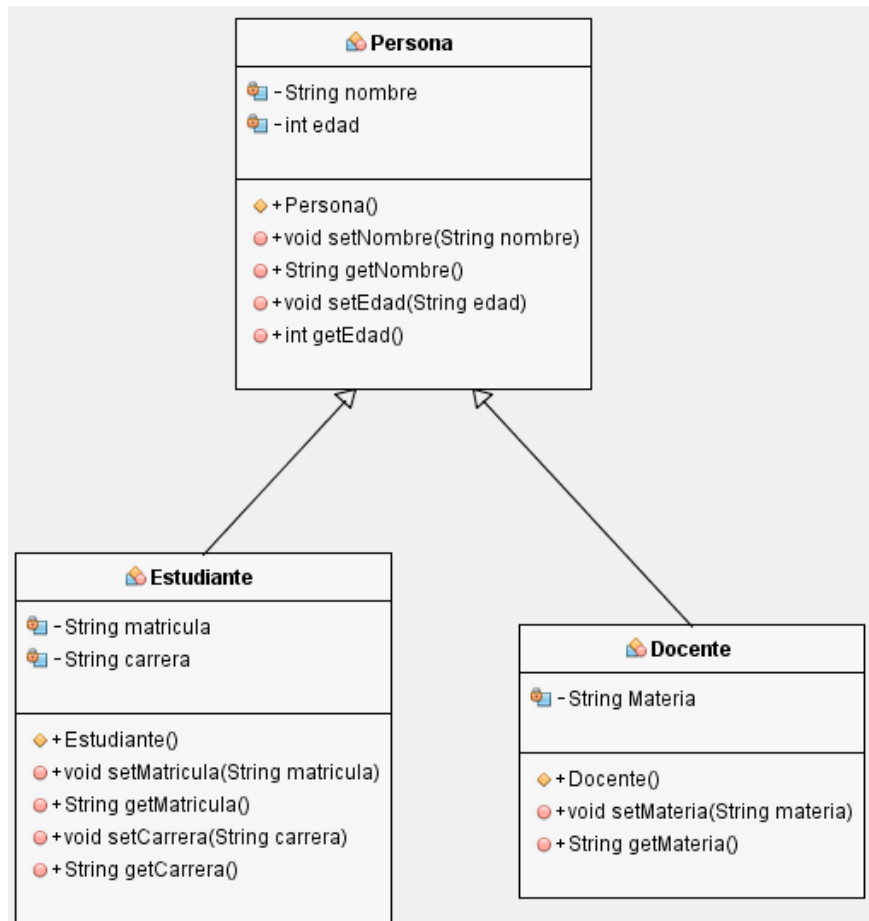
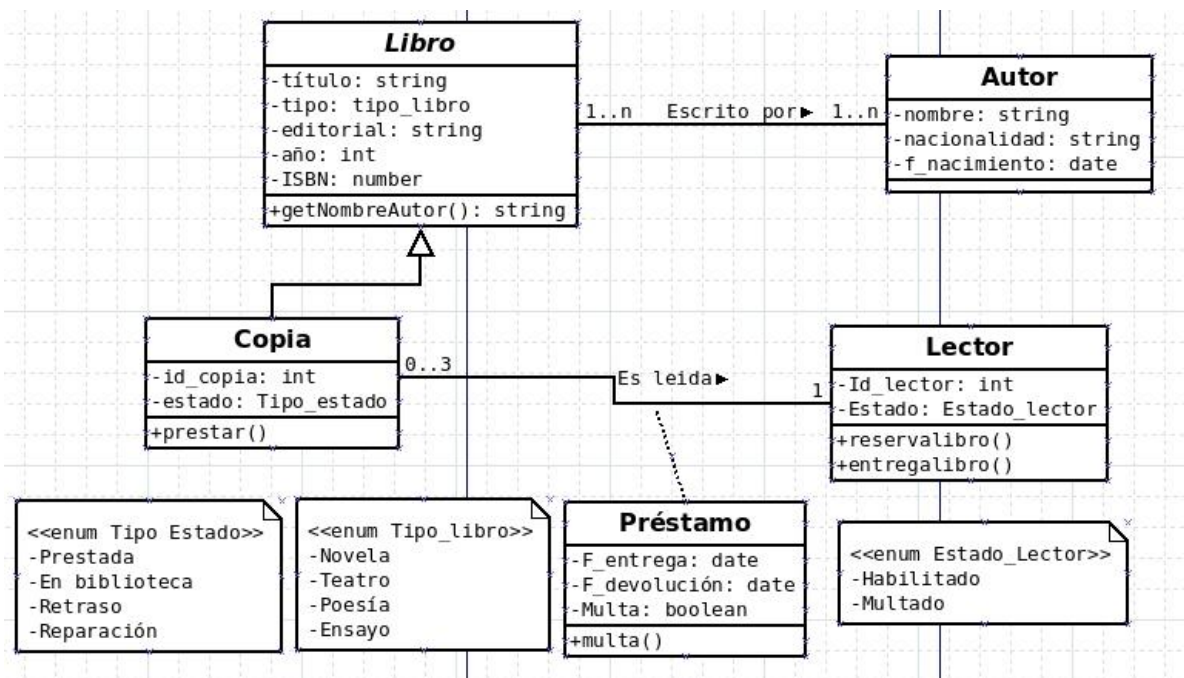
- **MÉTODOS (también llamados operaciones)**

Es la implementación de un servicio de la clase que muestra un comportamiento común a todos los objetos. Los métodos nos definirán cómo interactuará la clase con su entorno, contienen el código que manipula el estado del objeto

Los métodos, al igual que los atributos, pueden ser:

- **public:** el atributo será público, visible tanto fuera como dentro de la clase. Se representa con el signo “+”.
- **private:** el atributo solo será accesible dentro de la clase (solo sus métodos). Se representa con el signo “-”.
- **protected:** el atributo no será accesible desde fuera de la clase, pero sí por métodos de la clase y de subclases. Se representa con “#”.
- **package:** el atributo será visible en las clases del mismo paquete. Se representa con “~”.

Actividad: Dados los siguientes diagramas indica cuáles son las clases, sus métodos y atributos y si son públicos, privados, protegidos o empaquetados.



2.2.2. Relaciones

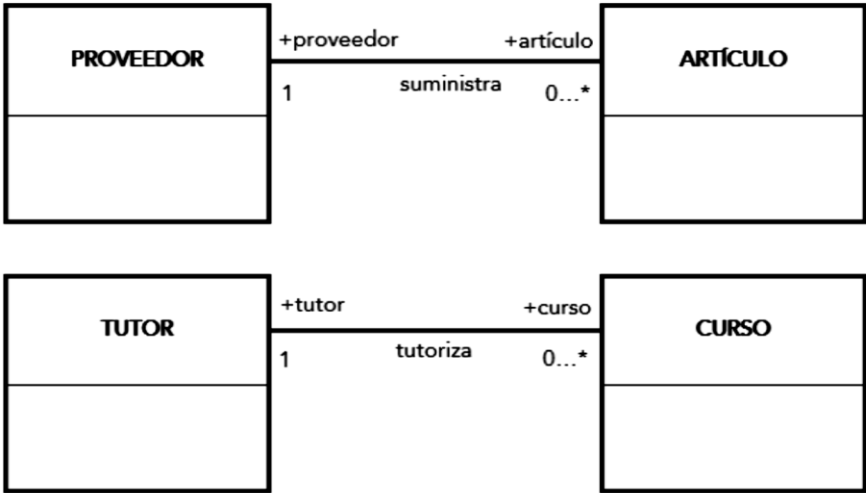
Los **objetos** estarán vinculados entre sí y se corresponden con asociaciones entre objetos. En UML, estos vínculos se describen a través de **asociaciones** al igual que los objetos se describen mediante clases.

Estas relaciones poseen un **nombre y una cardinalidad** llamada **multiplicidad**, que representa el número de instancias de una clase que se relaciona con las instancias de otra clase. La asociación es similar a la utilizada en el modelo Entidad/Relación.

Notación	Cardinalidad/Multiplicidad
0..1	Cero o una vez
1	Una y solo una vez
*	De cero a varias veces
1..*	De una a varias veces
M..N	Entre M y N veces
N	N veces

En cada extremo será posible indicar la multiplicidad mínima y máxima para indicar el intervalo de valores al que tendrá que pertenecer siempre la multiplicidad. Se usará la siguiente notación:

Ejemplos



Podremos distinguir los siguientes tipos de relaciones:

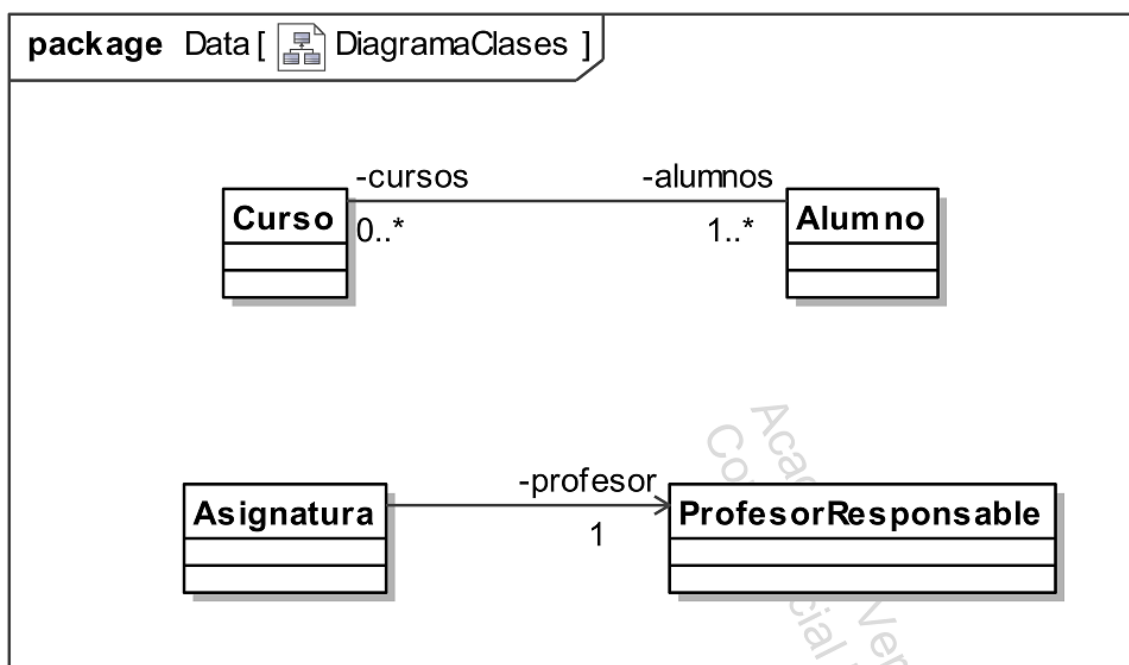
• ASOCIACIÓN

Podrá ser **unidireccional** o **bidireccional**, dependiendo de si una conoce la existencia de la otra o no. Cada clase cuenta con **un rol** que se indica en la parte superior o inferior de la línea que conecta a ambas clases y, además, este vínculo también tendrá un nombre representado con una línea que conecta a ambas clases.



En una asociación **bidireccional**, cada una de las clases tendrá un objeto que lo **relacione**. En cambio, en la asociación **unidireccional**, la clase destino no sabrá de la existencia de la clase origen, y **la clase origen contendrá un objeto o set de objetos de la clase destino**.

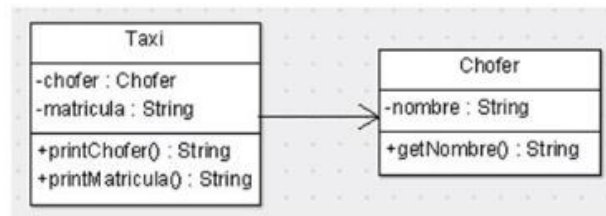
Dependiendo de la multiplicidad podemos pasar de un objeto de una clase a **uno o varios de la otra**. A este proceso se le llama **navegabilidad**. En la asociación unidireccional, la navegabilidad es solo en un sentido, desde el origen al destino, pero no al contrario.



Asociación

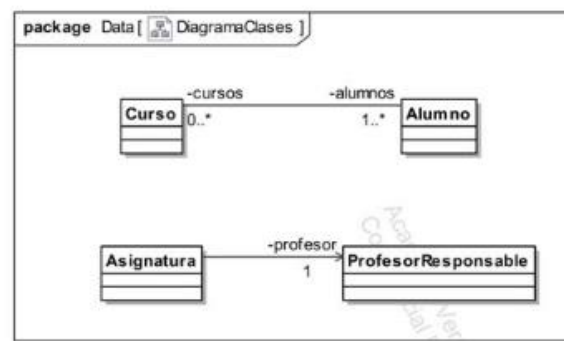
Una asociación implica que dos elementos del modelo tienen una relación.

Es una relación estructural entre las clases.

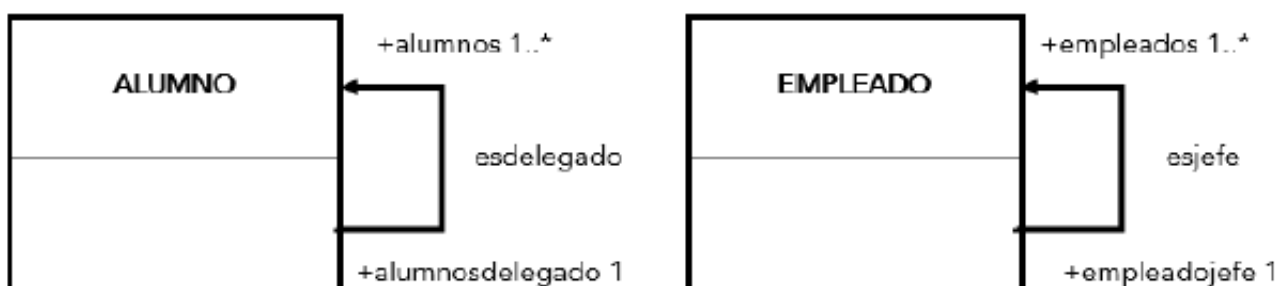


La primera es una **asociación bidireccional** que representa que un curso tiene desde 1 hasta varios alumnos y que un alumno puede estar en 0 o varios cursos.

La segunda es una **asociación unidireccional** que representa que una asignatura tiene un único profesor responsable.

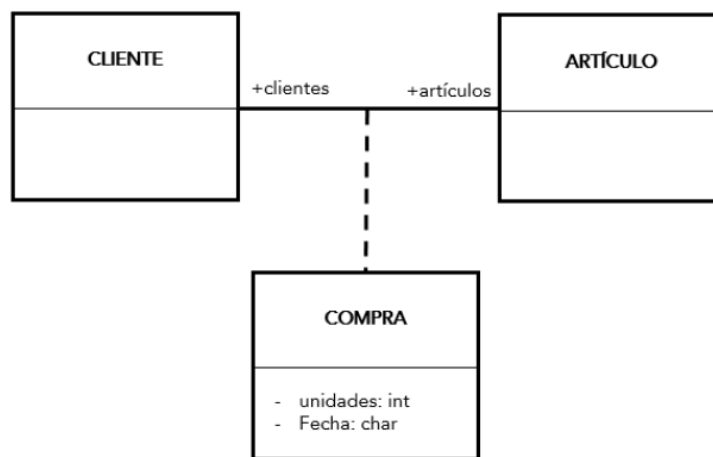


Algunas clases pueden asociarse consigo mismas creando así una **asociación reflexiva**. Estas asociaciones unen entre si instancias de una misma clase.



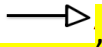
• CLASE ASOCIACIÓN

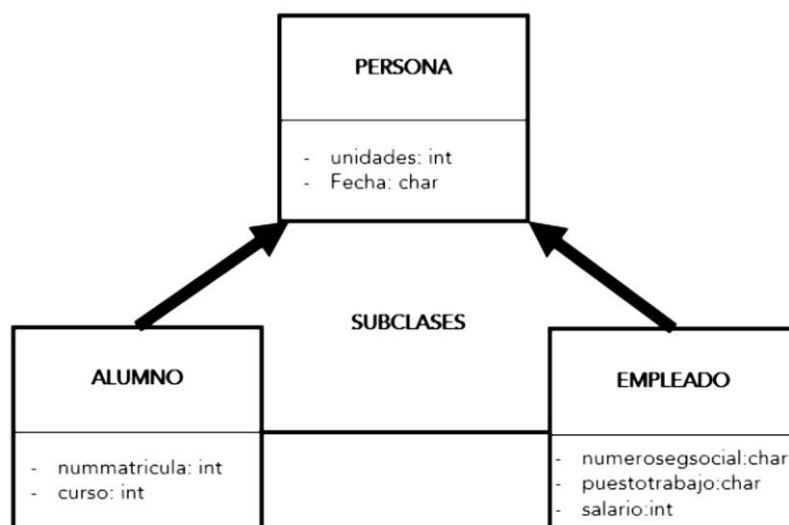
Hay asociaciones entre clases que podrán tener información necesaria para dicha relación, por lo que se creará una clase llamada **clase asociación**. Recibirá el estatus de clase y las instancias serán elementos de la asociación, al igual que podrán vincularse con otras clases y podrán tener atributos y operaciones. (similar a relaciones N:M en el modelo Entidad/Relación)



• HERENCIA (GENERALIZACIÓN y ESPECIALIZACIÓN)

Podremos **organizar las clases de forma jerárquica** y, a través de la herencia, seremos capaces **de compartir atributos y operaciones comunes con las demás clases a través de una superclase**. Las demás clases se conocen como subclases. La subclase hereda los atributos y métodos de la otra. La superclase generaliza a las subclases y las subclases especializan a la superclase.

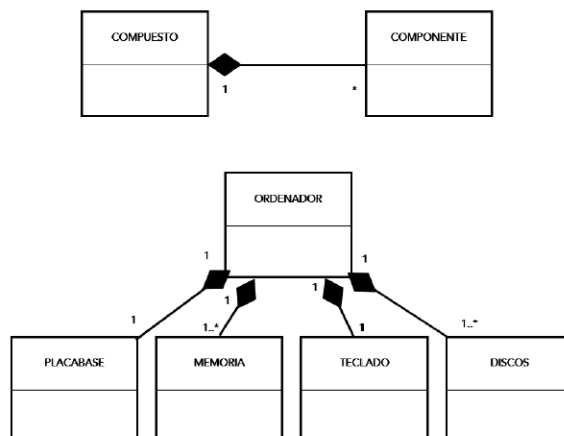
Para representar esta asociación se usará **una **, donde el extremo de la flecha apunta a la superclase.



• COMPOSICIÓN

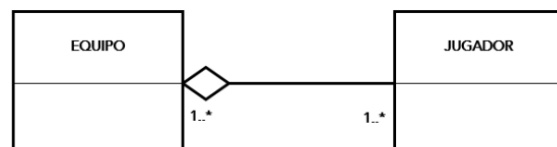
La asociación de composición consiste en que un objeto puede estar compuesto por otros objetos. Existirán dos formas: fuerte, conocida como **composición**; y débil, conocida como **agregación**.

En la asociación fuerte, los objetos están constituidos por componentes, es decir estos son una parte del objeto compuesto y no pueden ser compartidos entre varios objetos compuestos. La cardinalidad será uno y la supresión del objeto comporta la supresión de los componentes. Se representa con una línea con un rombo lleno



• AGREGACIÓN

Es la composición débil en la cual los componentes pueden ser compartidos por varios compuestos, y la destrucción de uno de ellos no implica la eliminación del resto. Se suele dar con más frecuencia que la composición. Al comienzo del proyecto podremos usar solamente agregación y, después, determinar cuál de ellas son composición. Su representación es una línea con un rombo vacío

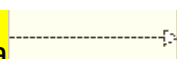


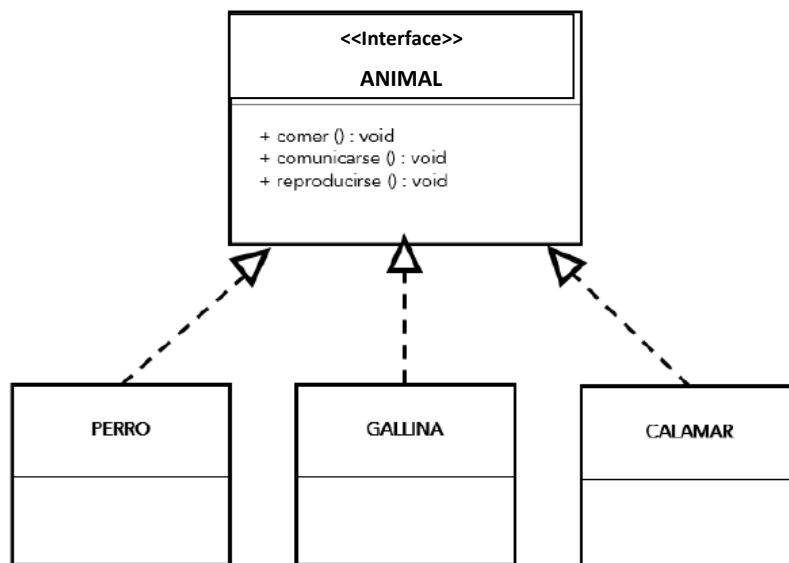
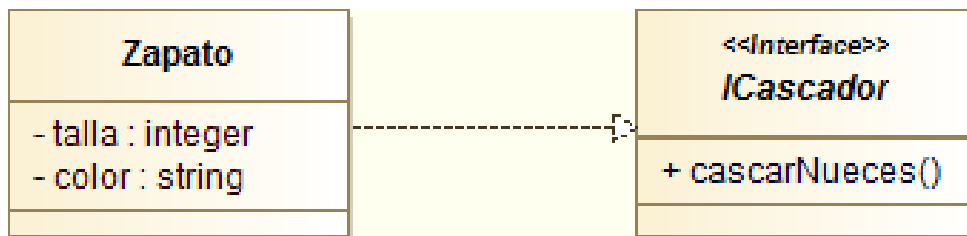
• REALIZACIÓN

Será la relación de herencia que existe entre la clase interfaz y una subclase que implementa esa interfaz. (En las interfaces se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describen la lógica del comportamiento de los métodos)

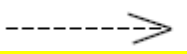
implements https://www.w3schools.com/java/java_interface.asp

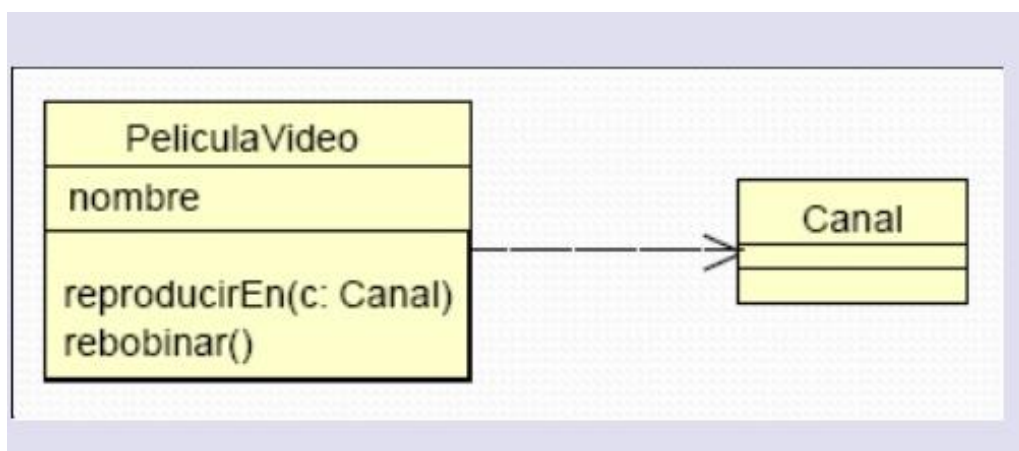
Se representa gráficamente con una flecha con línea discontinua

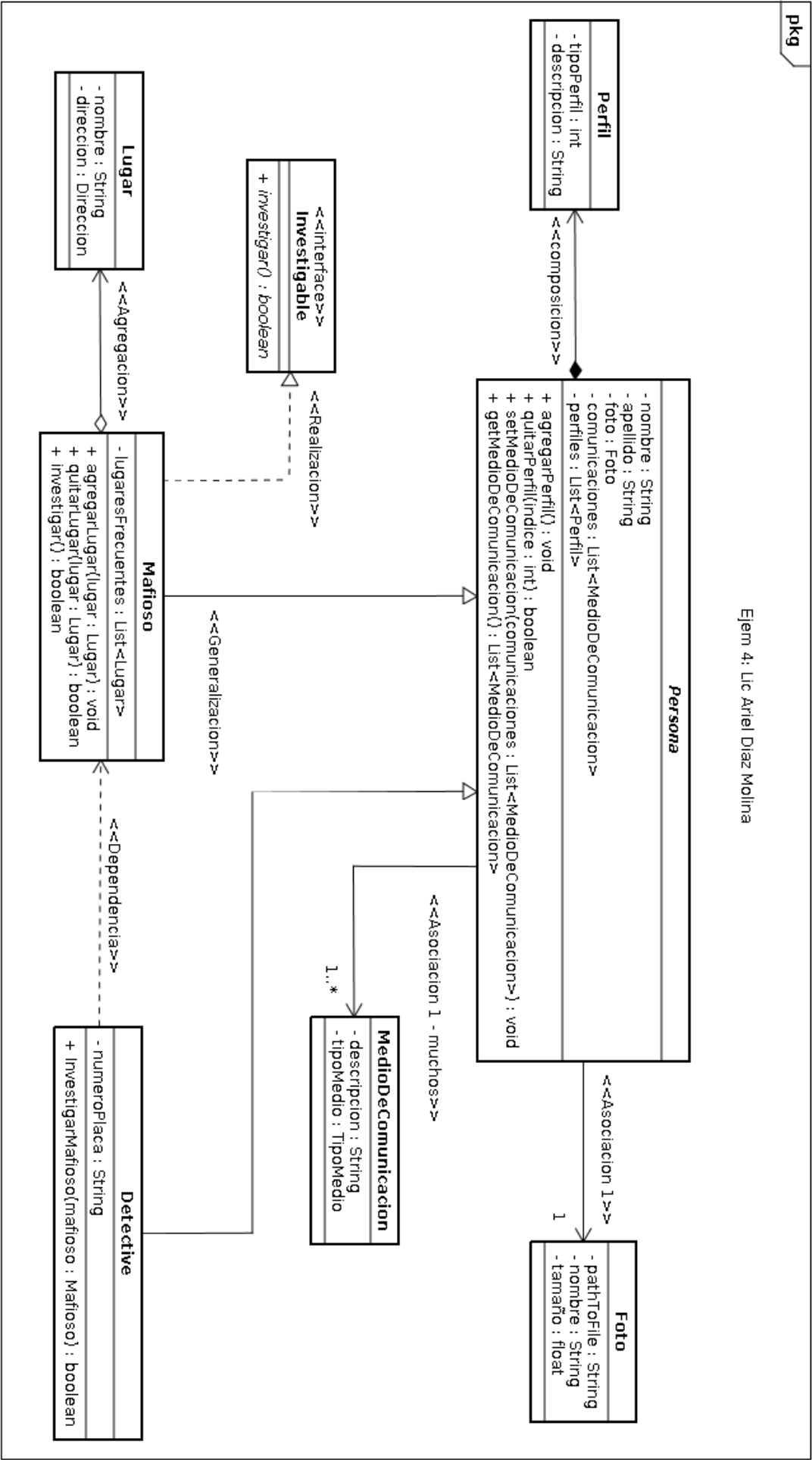




- DEPENDENCIA**

Relación que se establece cuando una clase utiliza el contenido de otra clase. Se representa con una flecha sin relleno discontinua  que irá desde la clase utilizadora a la utilizada. Un cambio en la clase utilizada puede afectar a la utilizadora, pero no al contrario.



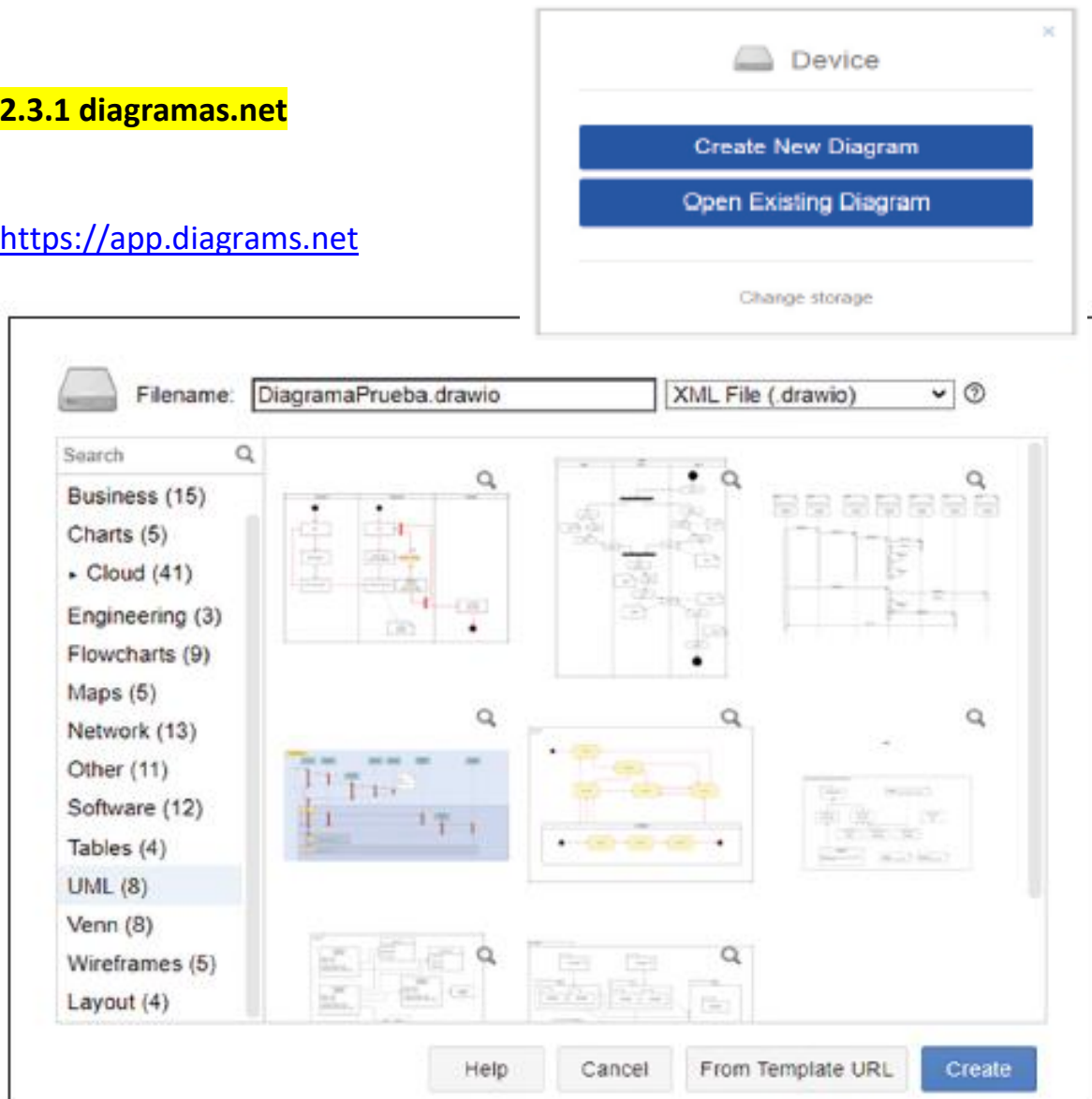


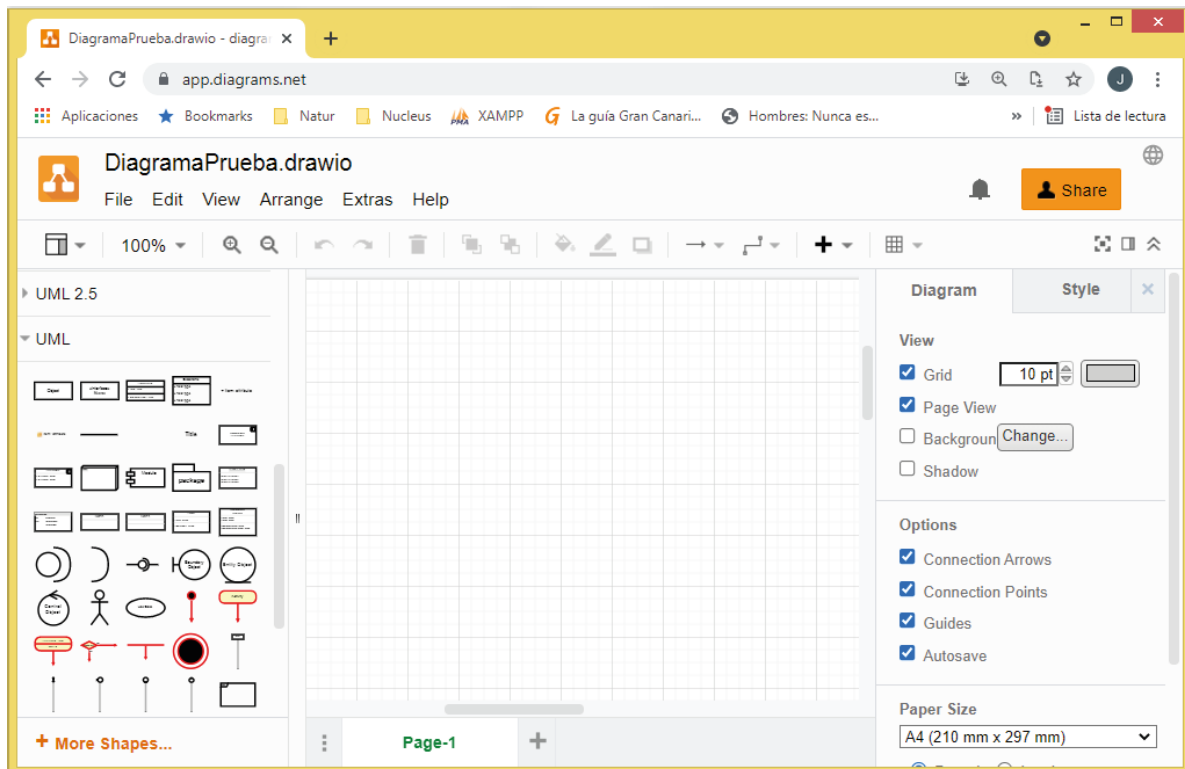
2.3. Herramientas para el diseño de diagramas

Hay muchas herramientas que soportan lenguaje UML. Tenemos que **tener claro para que la vamos a utilizar** por ejemplo para generar código java o simplemente para dibujar modelos y añadirlos a nuestra aplicación.

2.3.1 diagramas.net

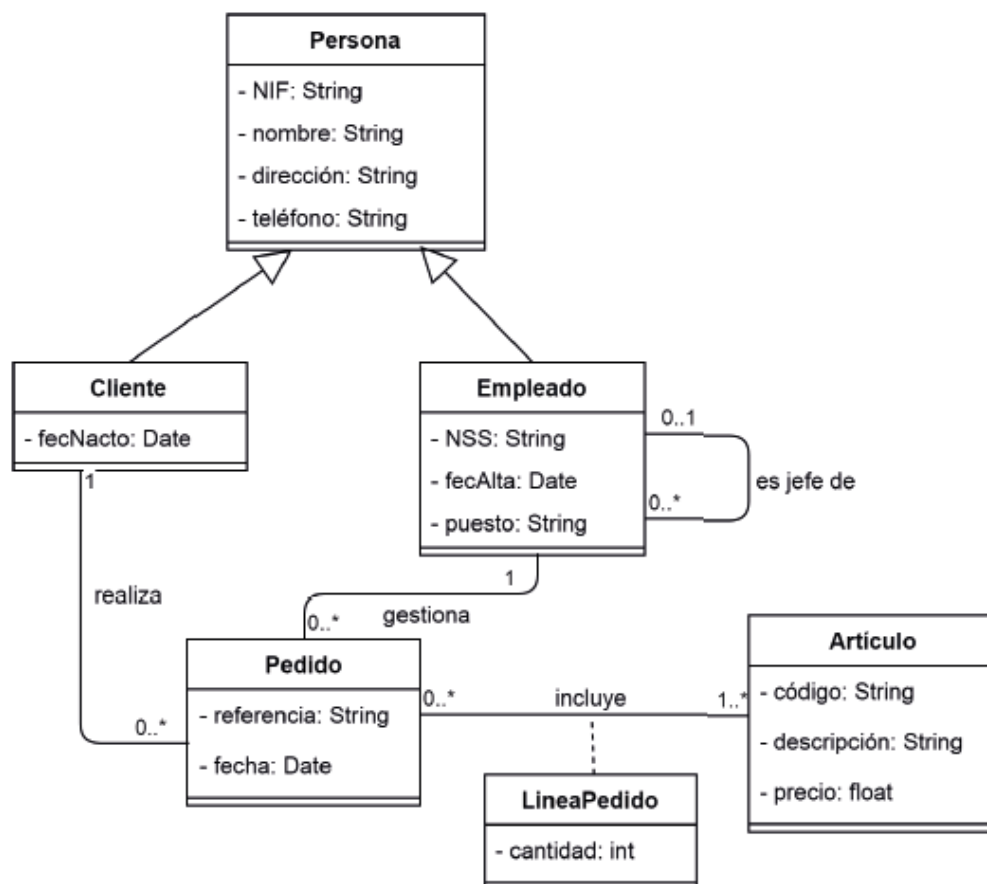
<https://app.diagrams.net>





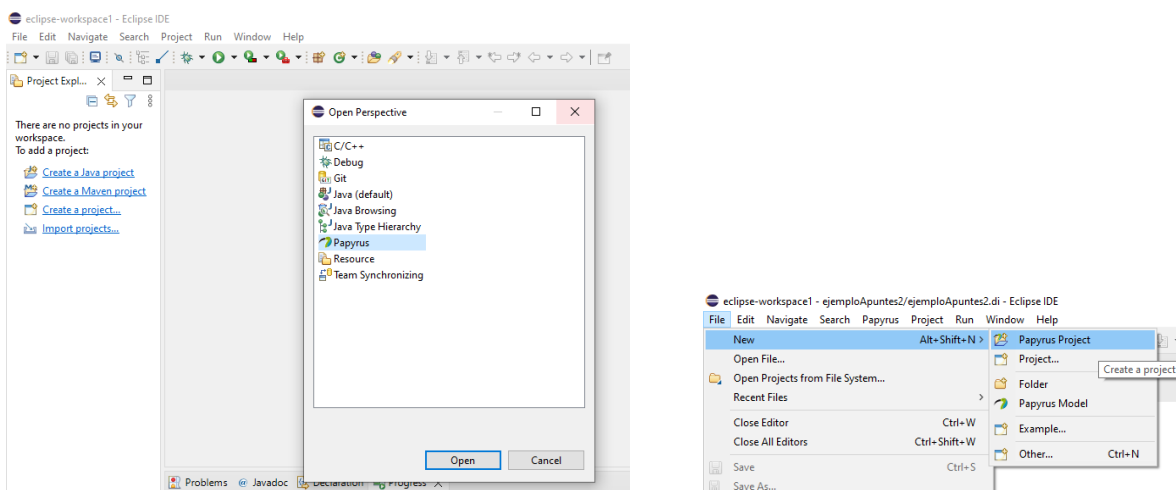
Pantalla principal de diagrams.net, donde se distinguen tres áreas de izquierda a derecha: área de paletas, área de edición y área de propiedades.

Diagrama de clases creado con diagrams.net que refleja el personal y los clientes de una empresa y los pedidos que estos hacen de los artículos que vende la empresa

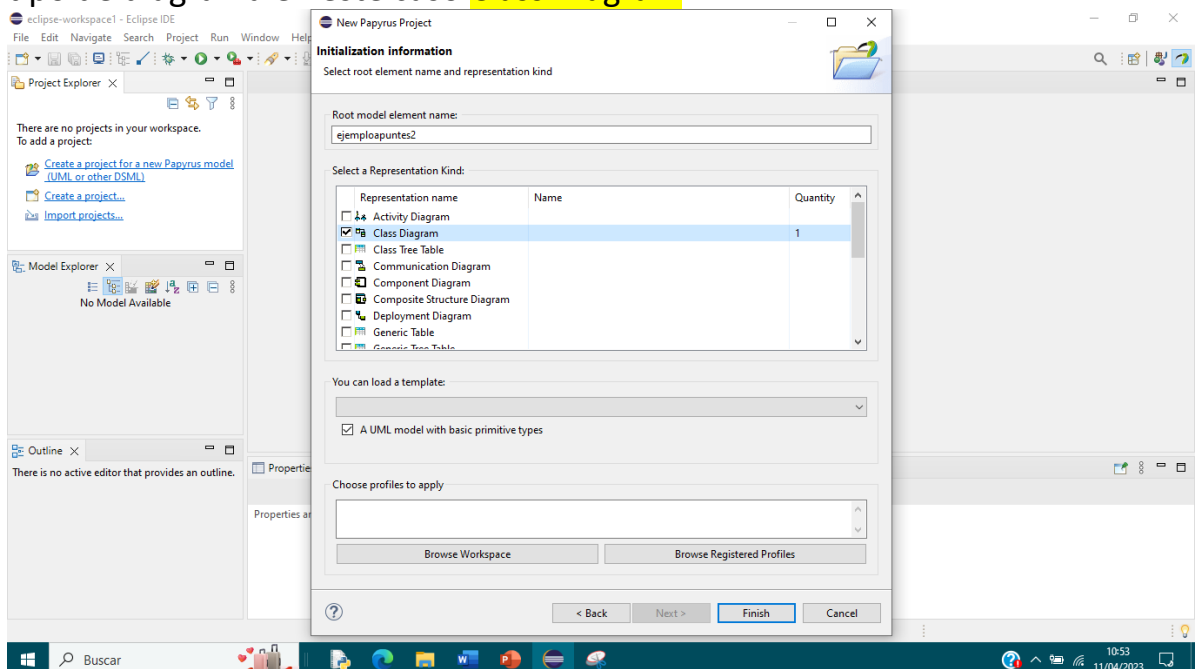


2.3.2. UML con Papyrus de Eclipse

- ❑ Para ver la vista Papyrus en Eclipse: Window → Perspective → Open Perspective → Other → <Papyrus>
- ❑ Si no aparece ir al menú *Help* y seleccionamos *Eclipse Market Place*. En *Find* escribimos *Papyrus* (papyrus software designer) e instalarlo.
- ❑ Para crear un proyecto: File → New → Papyrus Project

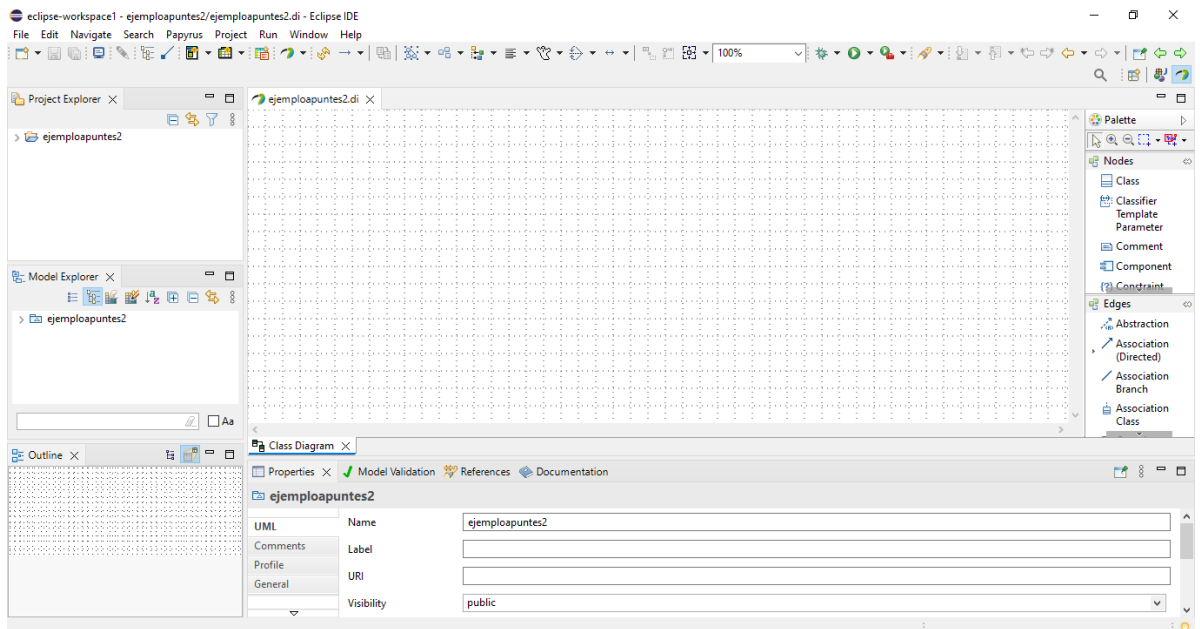


- ❑ Le ponemos **nombre** al proyecto y pulsamos **next**, hemos de elegir el tipo de diagrama en este caso **Class Diagram**.



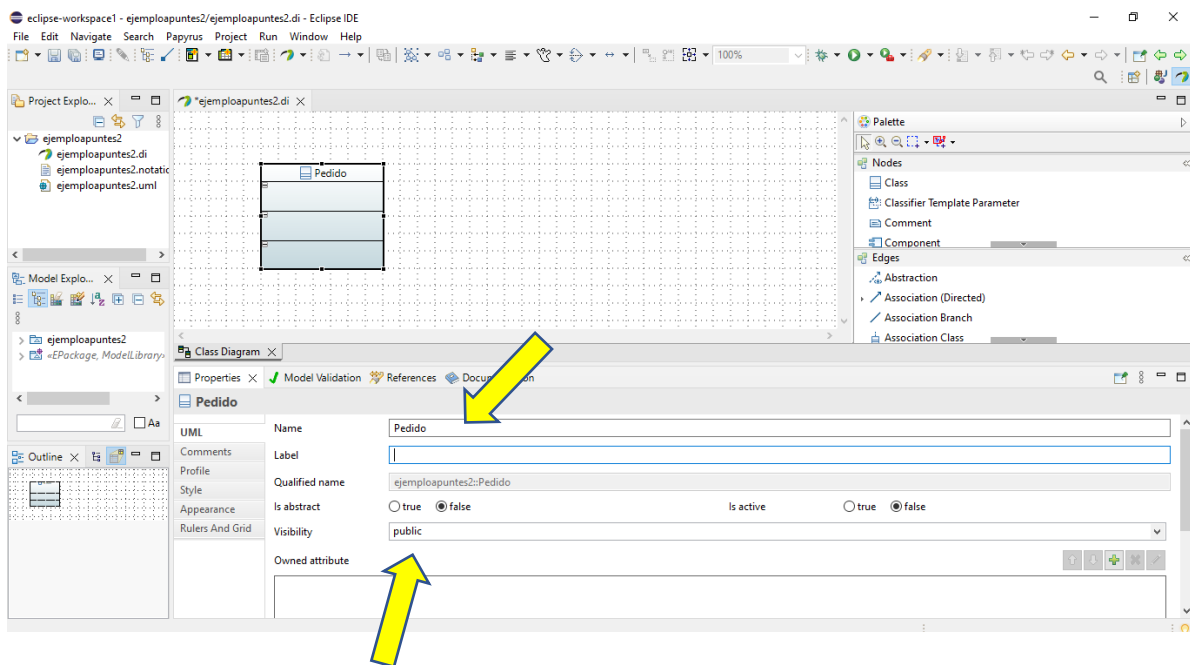
Tema 5. ED

Análisis y diseño orientado a objetos. Diagramas de clases.

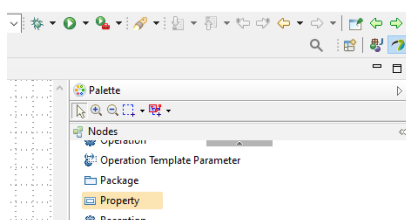


• CREACIÓN DE DIAGRAMAS DE CLASE

- ❑ Cogemos la Class de la paleta y pulsamos en la cuadrícula, vamos abajo a propiedades y le añadimos el nombre, si es pública o privada, etc.



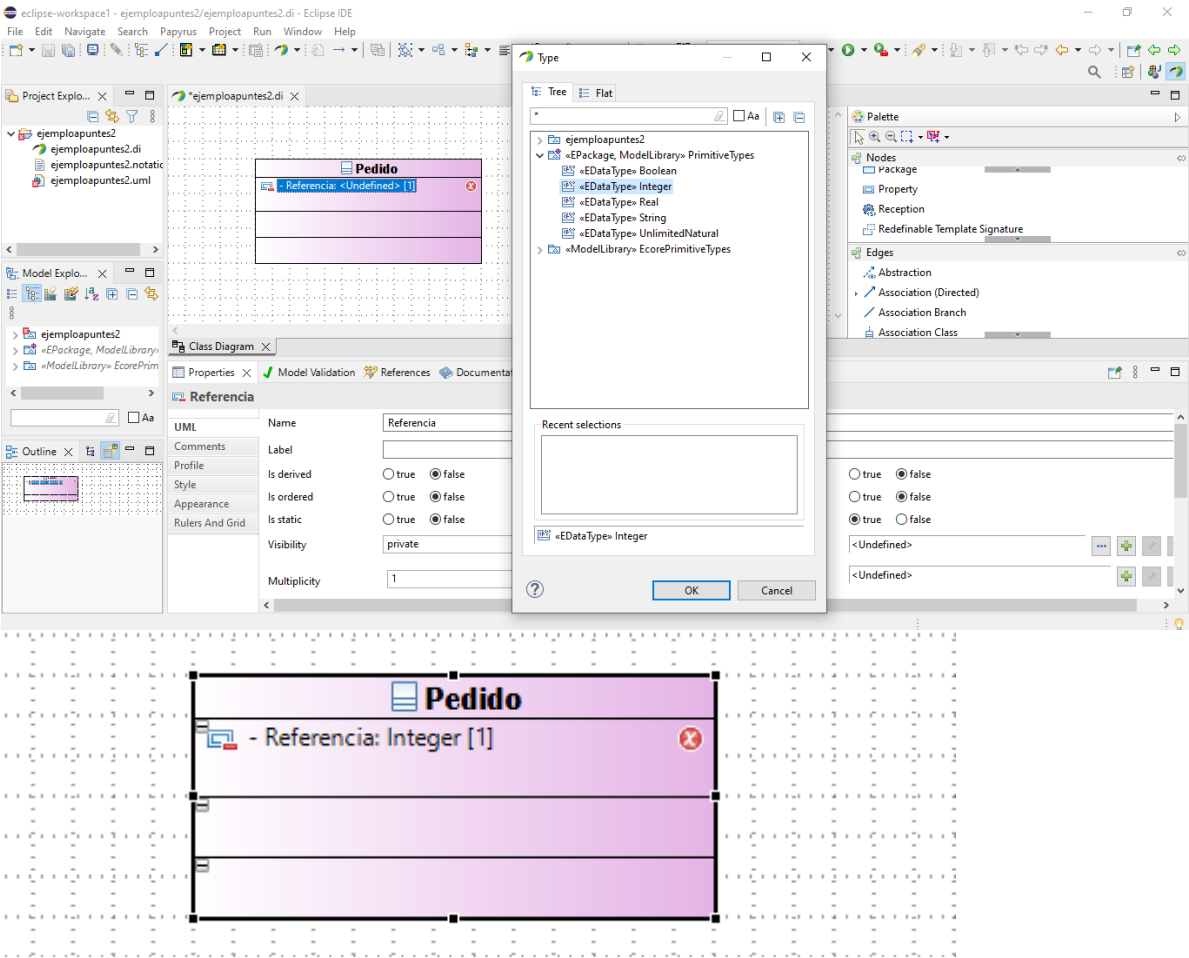
- ❑ Para añadir un **atributo** debemos seleccionar de la Paleta (**Property**)



, una vez hecho vamos abajo a propiedades y le añadimos sus propiedades. Nombre, tipo de dato y visibilidad son las más importantes

Tema 5. ED

Análisis y diseño orientado a objetos. Diagramas de clases.



- ❑ Para crear las otras dos clases vamos a **AssociationClass** (N:M esto nos crea directamente la clase para la relación)
- ❑ Y continuaríamos para crear el resto del diagrama.

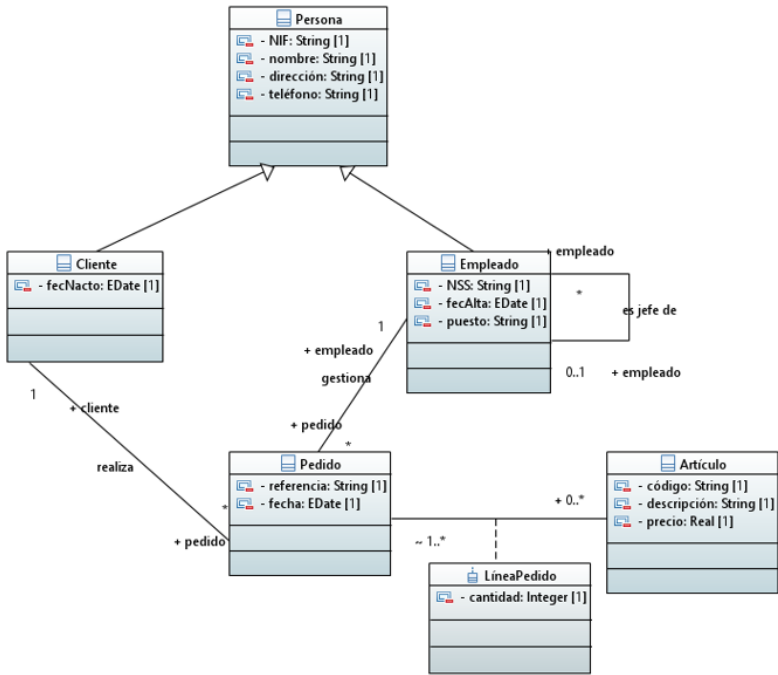
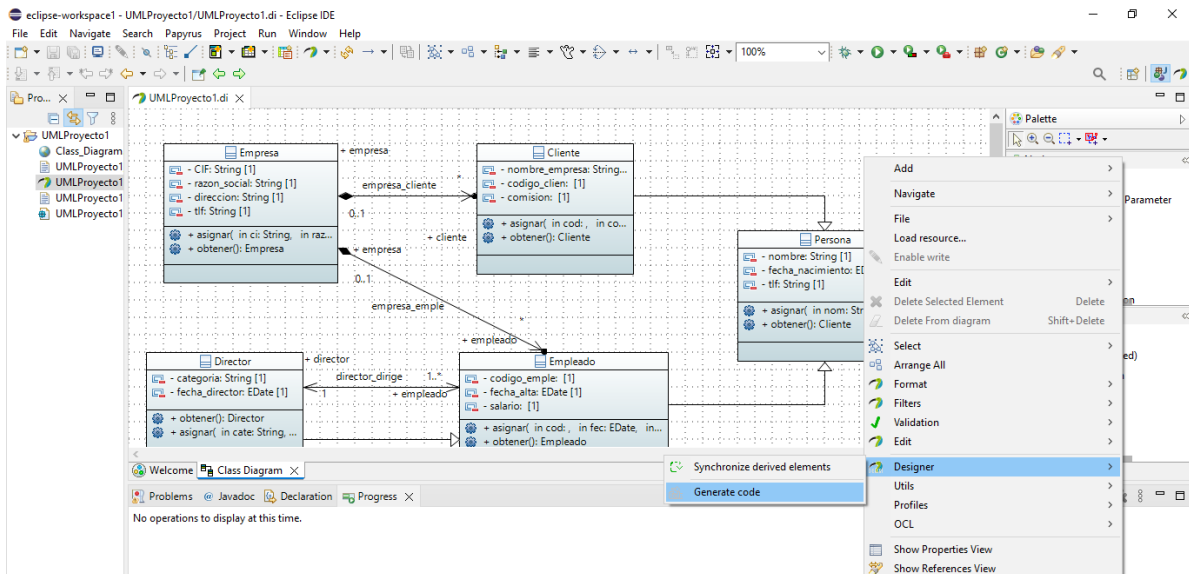


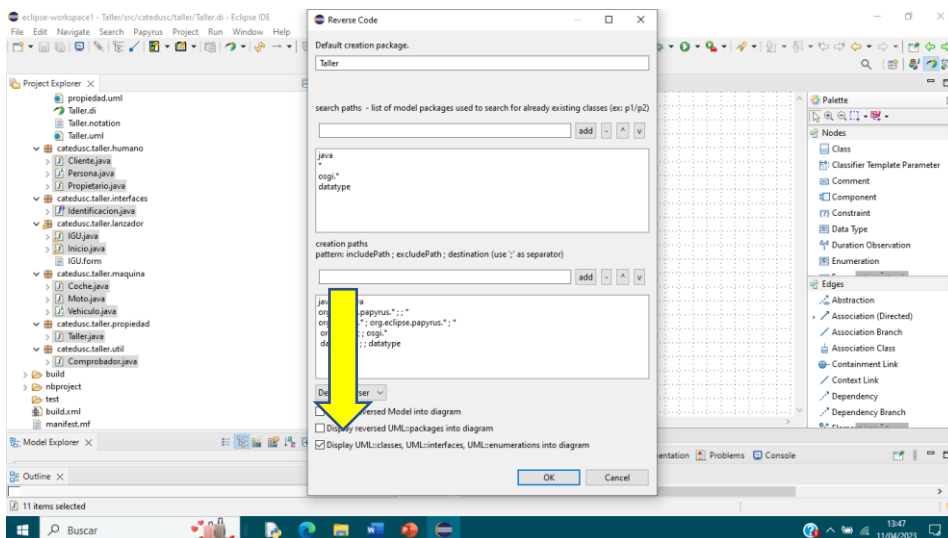
Figura 5.50. Diagrama de clases creado con Papyrus en Eclipse que refleja los empleados y clientes de una empresa y los pedidos que estos hacen de artículos que vende la empresa.

- ❑ Una vez que lo tengo todo, pulsando en cualquier parte de la cuadrícula vamos a **designer-generate code**.



Ahora veamos un ejemplo de ingeniería inversa.

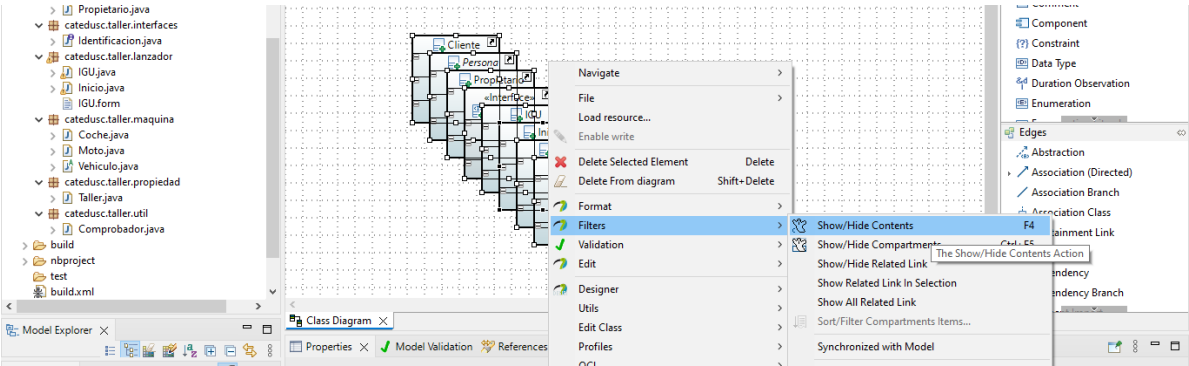
Tenemos el código y queremos generar el diagrama de clases. Solo tenemos que importar el proyecto. Seleccionar el paquete y buscar **new-papyrus model**. Una vez abierto el lienzo en blanco tendremos que arrastrar todas las clases .java y marcar la última opción.



Por último tendremos que reordenar y mostrar los atributos, métodos y relaciones que por defecto aparecen ocultos

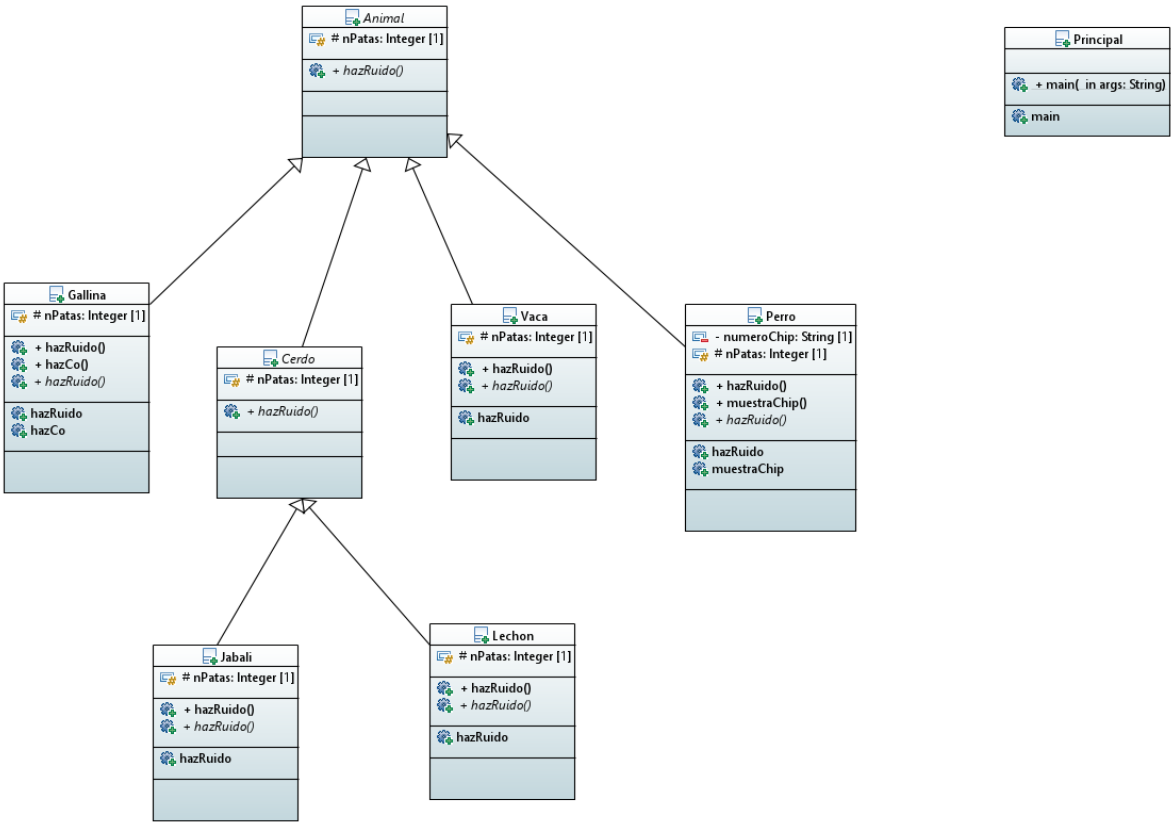
Tema 5. ED

Análisis y diseño orientado a objetos. Diagramas de clases.



Si queremos guardar el diagrama como imagen basta con darle al botón derecho y pulsar **File>Save as image file**.

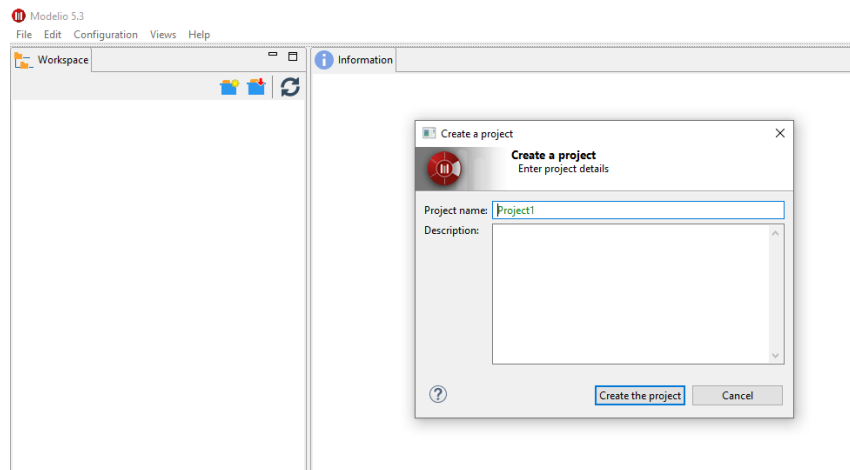
Ejercicio: importa el programa 02granja que te dejo en Moodle y sigue los pasos descritos anteriormente hasta llegar a la siguiente imagen.



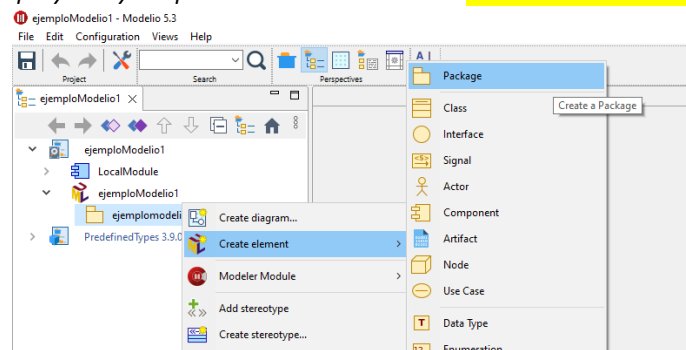
2.3.3. UML con Modelio

<https://www.modelio.org/downloads/download-modelio.html>

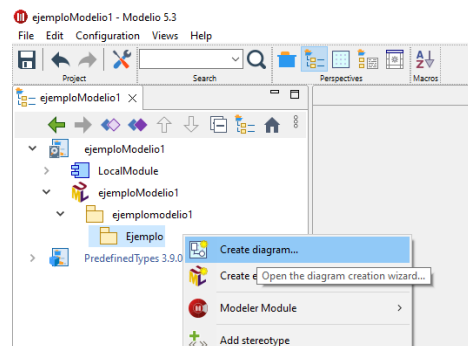
<https://github.com/ModelioOpenSource/Modelio/releases/tag/v5.3.1>



- Para crear un diagrama en Modelio, una vez creado un proyecto, hay que seleccionar la carpeta correspondiente al proyecto y la opción del menú contextual **"Create element > Package"**.



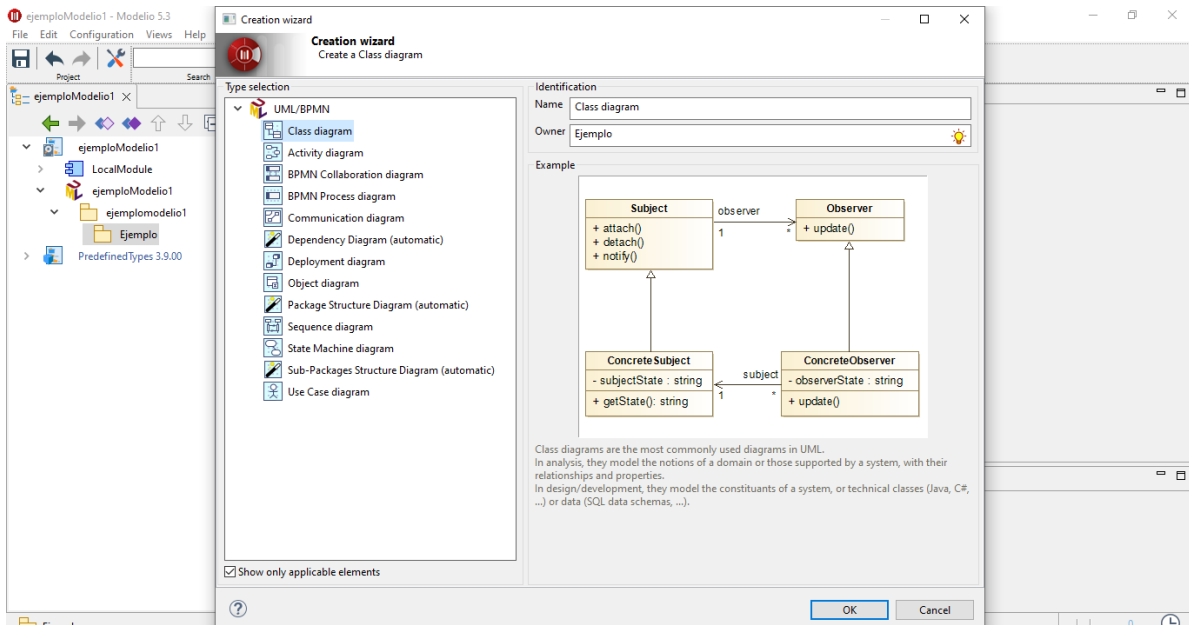
- Para crear un diagrama en Modelio, tras crear un paquete, hay que seleccionar el paquete y elegir la opción del menú contextual **"Create diagram"**



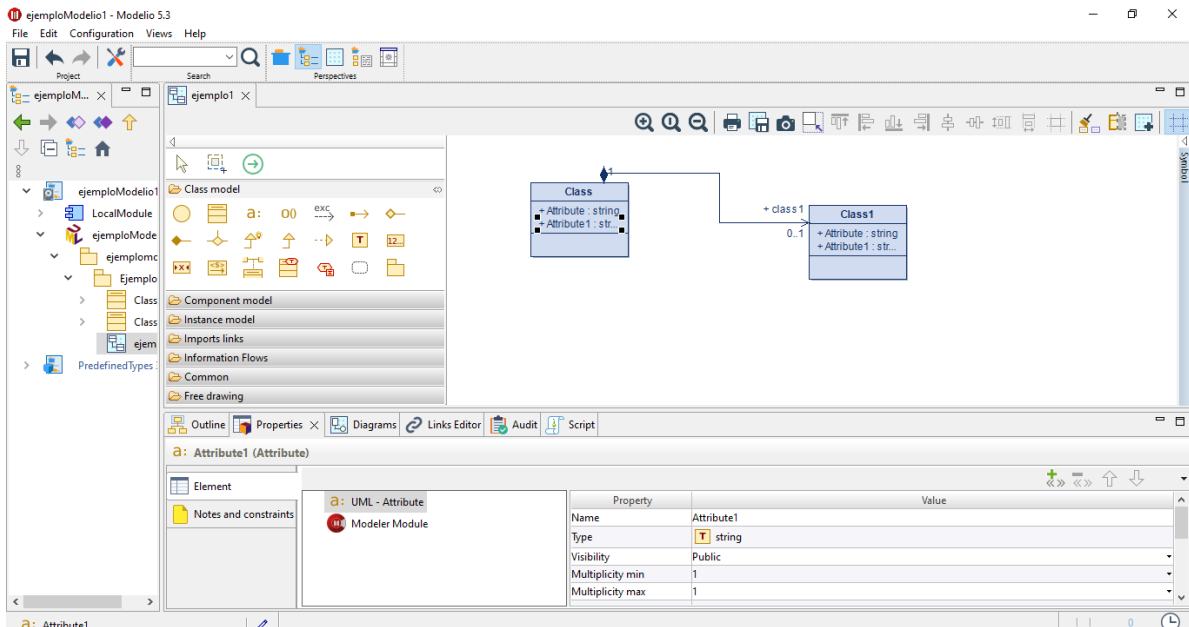
- Ventana que se muestra antes de proceder a la creación de un diagrama en la que se selecciona el tipo de diagrama que se desea crear y se le da un nombre.

Tema 5. ED

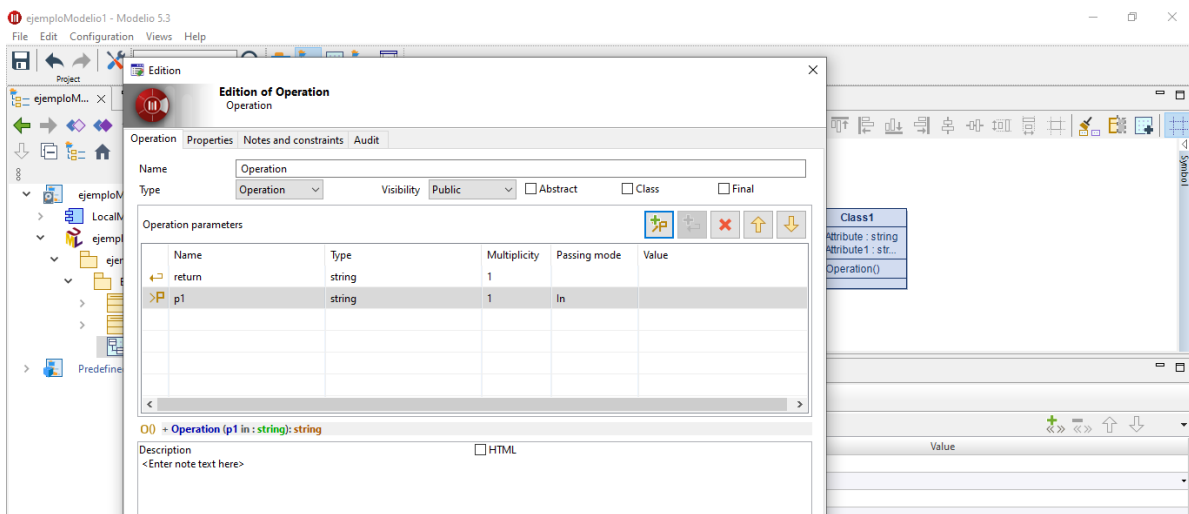
Análisis y diseño orientado a objetos. Diagramas de clases.



- Se van eligiendo las clases, atributos, métodos y relaciones. En la pestaña **"Properties"** se pueden editar las propiedades de cada elemento.



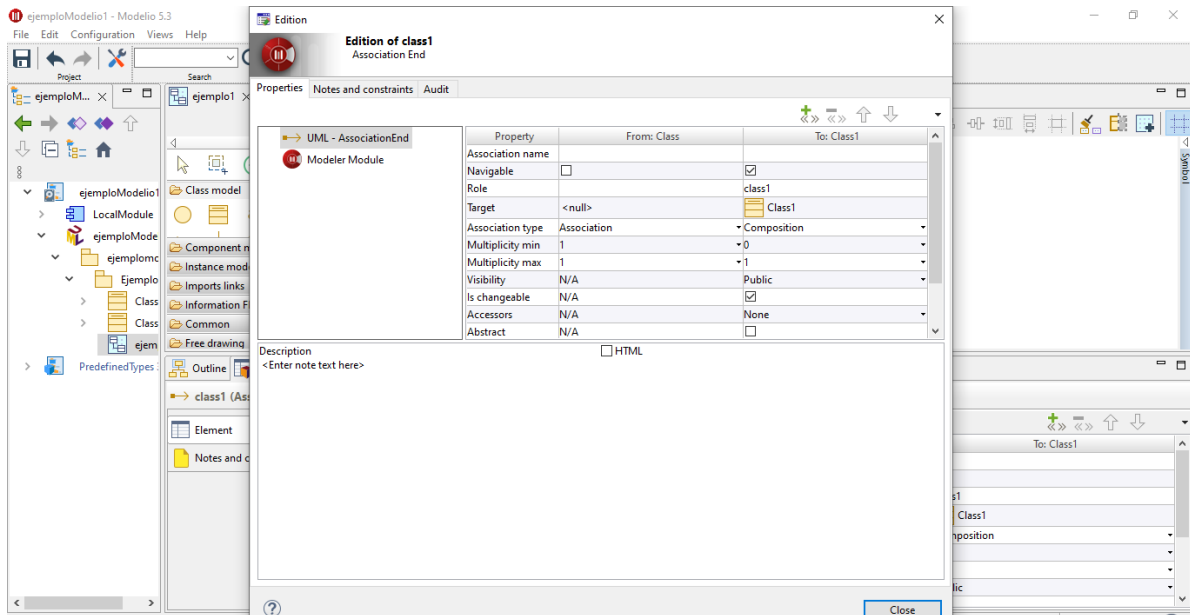
- Una vez que se ha agregado un método a una clase **"Operation()"** haciendo doble clic se nos abre esta pestaña de edición en la que podemos añadir el return y los parámetros correspondientes si los hay.



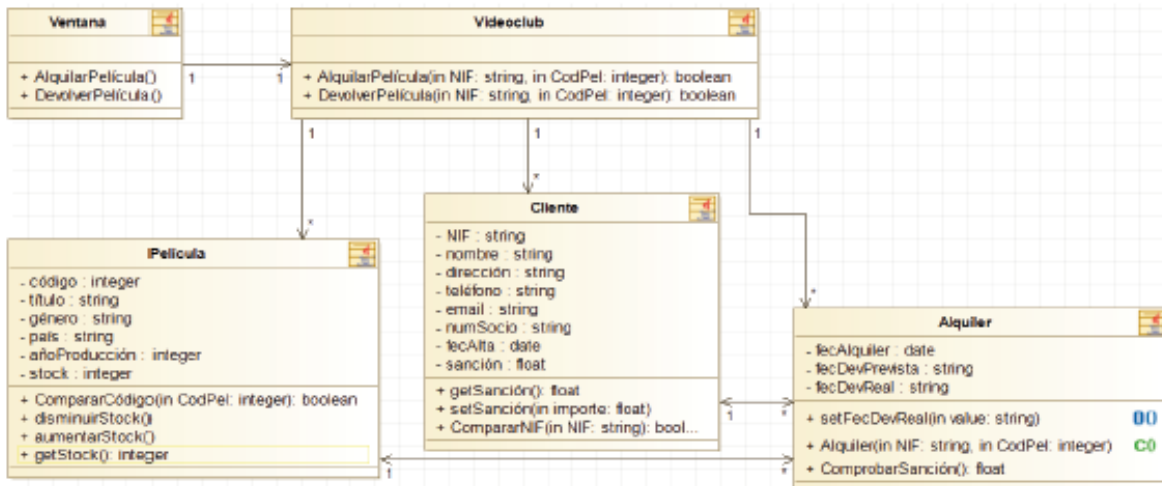
- Lo mismo sucede con las relaciones (doble clic) o también con **"Edit Element"**

Tema 5. ED

Análisis y diseño orientado a objetos. Diagramas de clases.

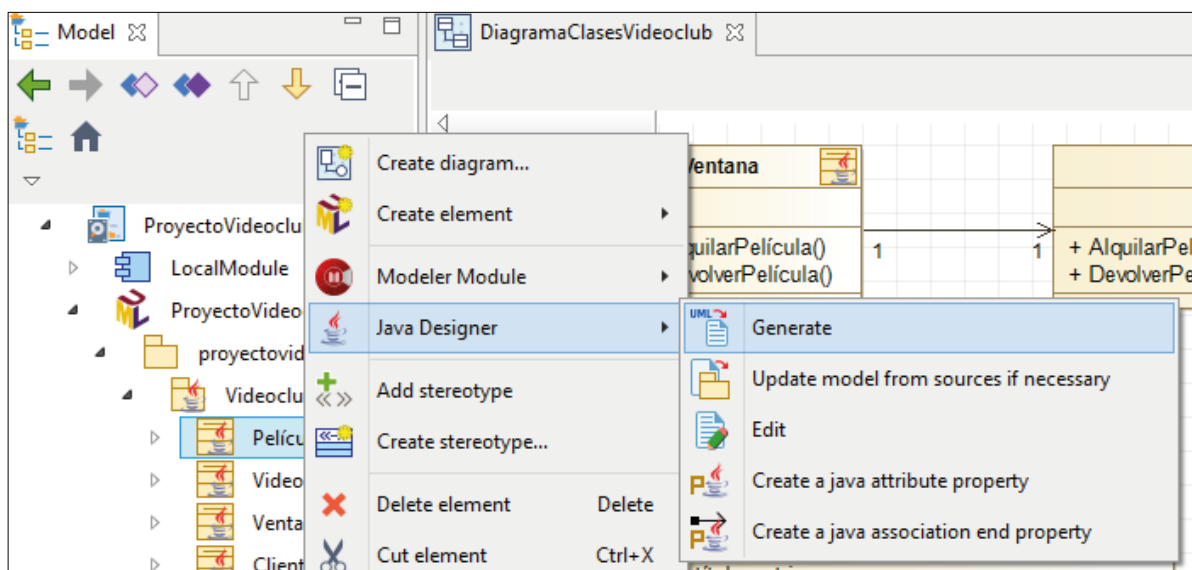


Ejercicio: Vamos a recrear el siguiente Diagrama de clases creado en Modelio que refleja la información que es necesario gestionar en un videoclub



Generación de código a partir de diagramas de clases:

- En Modelio es posible generar el código Java correspondiente a una clase activando la opción del menú contextual "Java Designer > Generate".



Tema 5. ED

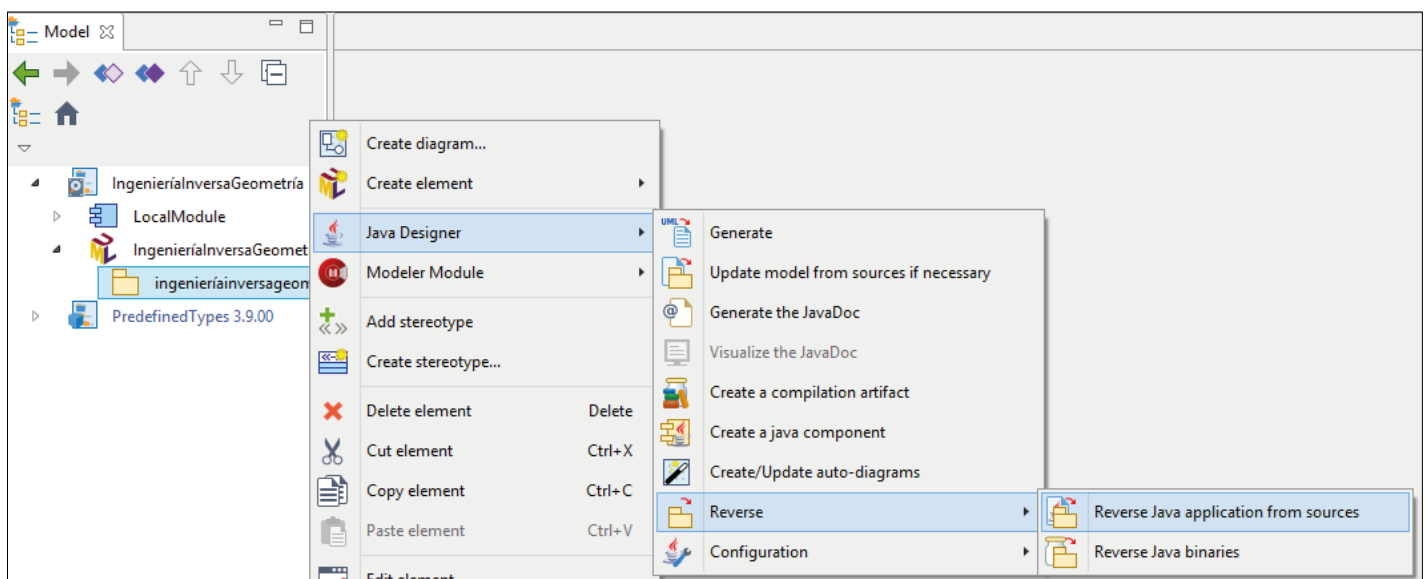
Análisis y diseño orientado a objetos. Diagramas de clases.

- En Modelio se puede visualizar el código Java correspondiente a una clase seleccionando la opción del menú contextual **Java Designer > Edit**.

```
DiagramaClasesVideoclub Película ✖  
  
package Videoclub;  
  
import java.util.ArrayList;  
import java.util.List;  
import com.modeliosoft.modelio.javadesigner.annotations.objjid;  
  
@objjid ("0ae7ce52-fd0f-4663-9f0f-f8652b91f153")  
public class Película {  
    @objjid ("5eac3cfc-b310-4790-b543-52759fc5108b")  
    private int código;  
  
    @objjid ("29e1c7a0-899f-44b6-93b1-bb79c4c6b198")  
    private String título;  
  
    @objjid ("a0864089-17c6-435f-b950-29b991a027ef")  
    private String género;  
  
    @objjid ("8a25c2c8-1c3e-4417-9980-8f0555191f8d")  
    private String país;  
  
    @objjid ("cc58639b-cb6d-4e5e-81c4-5d8d9b8b636d")  
    private int añoProducción;  
  
    @objjid ("d62bce6a-1ef1-461b-97c0-2f253ed51fc0")  
    private int stock;  
  
    @objjid ("8874f9ba-2127-43d3-929d-b2dbe028137c")  
    public List<Alquiler> = new ArrayList<Alquiler> ();  
  
    @objjid ("333be3d4-3e77-48d0-b7a9-67c2b0df7f47")  
    public boolean CompararCódigo(int CodPel) {  
    }  
  
    @objjid ("c51f22d1-b788-4298-b882-8e81cc9e7ca4")
```

Generación de diagramas de clases a partir de código (ingeniería inversa)

- Para realizar ingeniería inversa a partir de archivos con código fuente de Java en Modelio, se debe crear un proyecto Java y activar la opción del menú contextual **"Java Designer > Reverse > Reverser Java application from sources"**



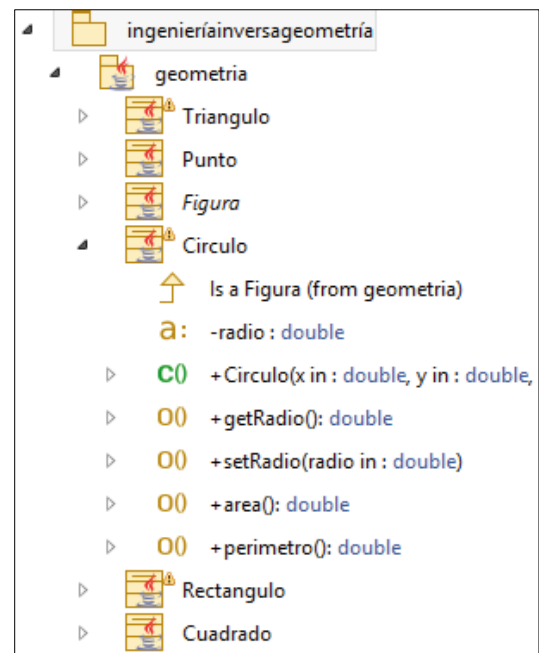
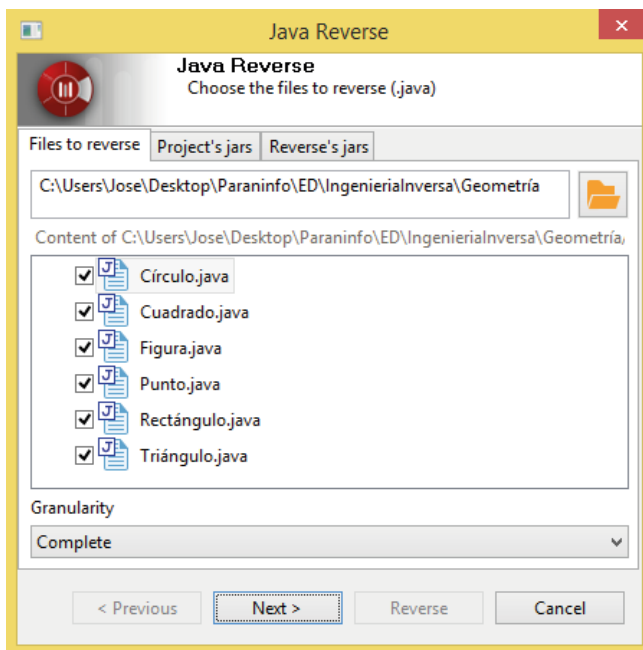


Diagrama de clases sin métodos resultado del proceso de ingeniería inversa en Modelio.

