

6. ELABORACIÓN DE DIAGRAMAS DE CLASE


GUILLERMO PALAZÓN CANO

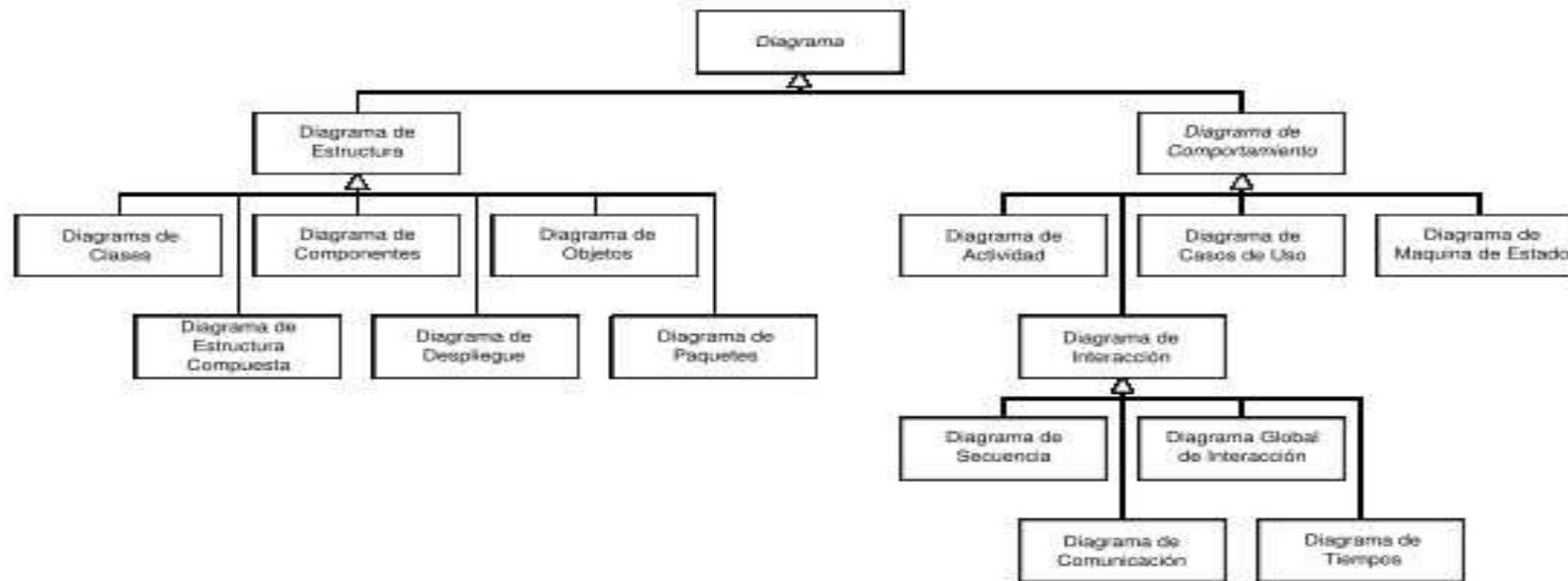
1. INTRODUCCIÓN


- ▶ 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 de una colección de métodos u operaciones que manipulan esos datos. Los datos definidos dentro de un objeto son sus atributos.
- ▶ Una clase no es más que una plantilla para la creación de objetos. Cuando se crea un objeto (instancia) se ha de especificar de qué clase es el objeto instanciado, para que el compilador comprenda las características del objeto.
- ▶ Para el análisis y diseño orientado a objetos se utiliza UML

1.1. UML

- ▶ UML (Unified Modeling Language – Lenguaje de modelado unificado) es un lenguaje de modelado basado en diagramas que sirve para expresar modelos (un modelo es una representación de la realidad donde se ignoran detalles de menor importancia).
- ▶ Se ha convertido en el estándar de facto de la mayor parte de las metodologías de desarrollo orientado a objetos que existen hoy en día.
- ▶ Es un lenguaje gráfico que permite visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software. Aunque su uso se utiliza generalmente en diseños de sistemas de software, se puede emplear también en hardware así como en organizaciones del mundo real.


- 
- ▶ Existen dos grandes versiones de UML
 - ▶ UML 1.X (comprende UML 1.1, 1.2, 1.3, 1.4 y 1.5). Desde finales de los 90 se empezó a trabajar con el estándar UML y fueron surgiendo nuevas versiones que introducían mejoras o ampliaban las anteriores.
 - ▶ UML 2.X (comprende UML 2.1 y sus sucesivas versiones). En torno a 2005 se difundió una nueva versión de UML a la que podemos denominar UML 2.X en la que pasamos de 9 a 13 tipos de diagramas.
 - ▶ Estos diagramas se pueden agrupar en dos categorías fundamentalmente:
 - ▶ Diagramas de estructura (parte estática del modelo y se centran en los elementos que deben existir en el sistema modelado).
 - ▶ Diagramas de comportamiento (parte dinámica del modelo y se centran en lo que debe suceder en el sistema). Dentro de los diagramas de comportamiento tenemos los diagramas de interacción que a su vez también se pueden dividir en otros conjuntos de diagramas.




- 
- ▶ UML sigue siendo criticado por no tener reglas lo suficientemente estrictas como para evitar problemas de interpretación.
 - ▶ Teniendo presente la superestructura de diagramas de UML, podríamos pensar que la tarea de definir y realizar dichos diagramas (y por supuesto que sean coherentes entre sí) sería una tarea abrumadora, pero no es necesario ni se suelen realizar todos los diagramas para modelar un software.
 - ▶ Por norma general, se utilizan solo una serie de diagramas para modelarlo (lo más clásico sería la triada diagrama de clases, de secuencia y de casos de uso, aunque en algunos casos se suele incorporar alguno más).

1.2 DIAGRAMAS MÁS UTILIZADOS

- ▶ 1. Diagramas de clases. Los diagramas de clases muestran las diferentes clases que componen un sistema y como se relacionan unas con otras. Es un diagrama de estructura.
- ▶ 2. Diagramas de objeto. Representan objetos y sus relaciones. Muestra una serie de objetos (instancias de las clases) y sus relaciones en un momento particular de la ejecución del sistema. Son útiles para complementar la comprensión del diagrama de clases. Es un diagrama de estructura.
- ▶ 3. Diagrama de casos de uso. Se utiliza para entender el uso del sistema, muestran un conjunto de actores, las acciones (casos de uso) que realizan en el sistema, y las relaciones entre ellos. Es un diagrama de comportamiento

- 
- ▶ 4. Diagramas de secuencia. Son una representación temporal de los objetos y sus relaciones. Enfatiza la interacción entre los objetos y los mensajes que intercambian entre sí junto con el orden temporal de los mismos. Es un diagrama de comportamiento.
 - ▶ 5. Diagramas de estado. Se utiliza para analizar los cambios de estado de los objetos. Muestra los estados, eventos, transiciones y actividades de los diferentes objetos. Es un diagrama de comportamiento.
 - ▶ 6. Diagramas de actividad. En UML, un diagrama de actividad se utiliza para mostrar la secuencia de actividad. Muestran el flujo de trabajo desde un inicio hasta el punto final detallando las decisiones que surgen en la progresión de los eventos contenidos en la actividad. Es un diagrama de comportamiento.

- 
- ▶ 7. Diagramas de paquetes. Los diagramas de paquetes se usan para reflejar la organización de los paquetes y sus elementos. Suele usarse para organizar diagramas de casos de uso y diagramas de clases, aunque el uso de diagramas de paquetes no se limita a estos elementos UML.
 - ▶ A lo largo de esta unidad nos vamos a centrar en el diseño de diagramas de clases.

2. DIAGRAMAS DE CLASES

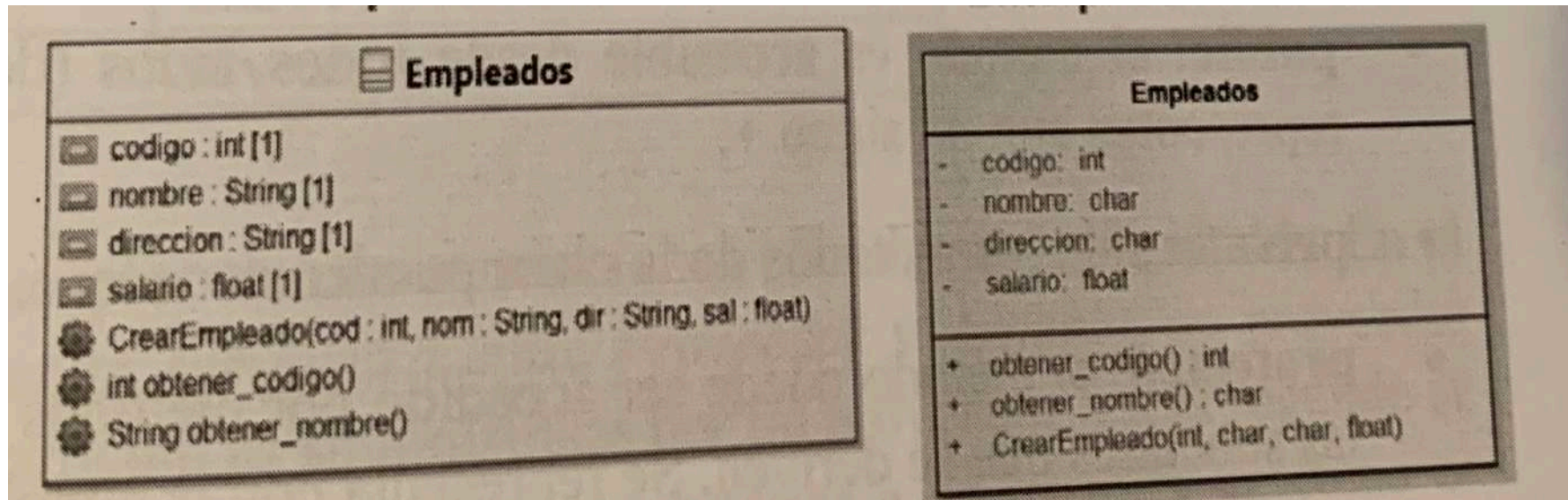
- ▶ Uno de los diagramas más básicos e importantes que realizaremos en nuestras labores de diseño de software serán los diagramas de clases.
- ▶ Se basan en ciertas notaciones y reglas sencillas para relacionar las clases y sus diferentes operaciones entre sí.
- ▶ En la POO son un recurso básico y recurrente, usado para mostrar los bloques de construcción de cualquier sistema y describir la capa del modelo y las relaciones entre las entidades del sistema.
- ▶ Un diagrama de clases está compuesto por los siguientes elementos:
 - ▶ Clases, atributos y métodos
 - ▶ Relaciones: Asociación, herencia, agregación, composición, realización y dependencia.

2.1 CLASES, ATRIBUTOS Y MÉTODOS

- ▶ Las clases son la unidad básica que encapsula toda la información de un objeto (un objeto es una instancia de una clase).
- ▶ En UML, la notación de una clase se representa por un rectángulo que posee tres divisiones:
 - ▶ La parte superior contiene el nombre de la clase
 - ▶ El intermedio contiene los atributos (o variables de la instancia) que caracterizan a la clase (pueden ser private, protected, package o public).
 - ▶ Inferior contiene los métodos u operaciones, los cuáles son la forma como interactúa el objeto con su entorno.
 - ▶ En la representación de una clase los atributos y métodos pueden omitirse



- Representación de una misma clase con dos herramientas distintas (Eclipse UML y Enterprise Architect)



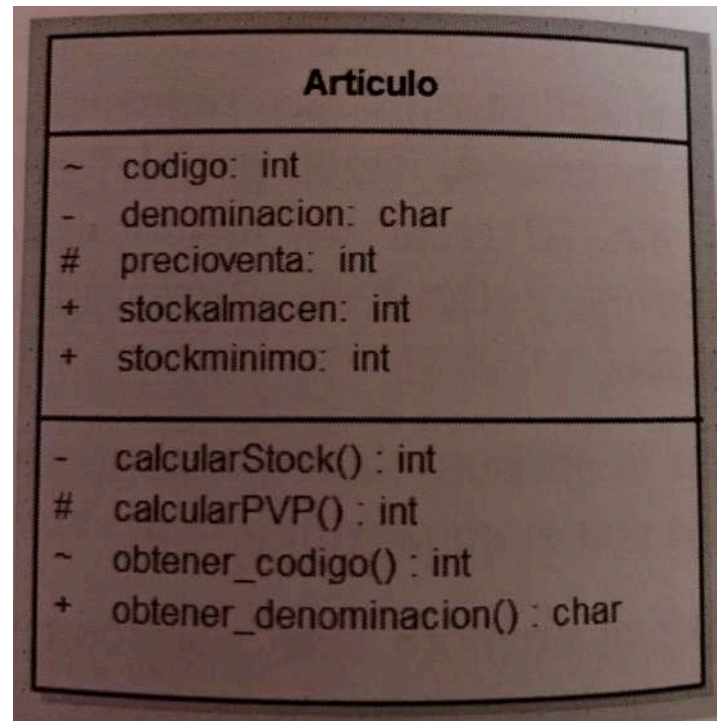
2.1.1 ATRIBUTOS

- ▶ Un atributo representa alguna propiedad de la clase que se encuentra en todas las instancias de la clase.
- ▶ Los atributos pueden representarse solo mostrando su nombre, o mostrando su nombre y su tipo, e incluso su valor por defecto.
- ▶ Al crear los atributos se indicará el tipo de datos (los tipos básicos UML son Integer, String y Boolean).
- ▶ También indicaremos la visibilidad del atributo con el entorno.
 - ▶ public: El atributo será publico y por tanto visible fuera de la clase (accesible desde todos lados). Se representan con el signo +
 - ▶ private: El atributo solo será accesible desde dentro de la clase (solo sus métodos pueden acceder al atributo). Se representan con el signo -
 - ▶ protected: El atributo no será accesible desde fuera de la clase, pero si podrá ser accedido por todos los métodos de la clase, además de las subclases que se deriven. Se representan con la almohadilla #
 - ▶ package: El atributo empaqueta es visible en las clases del mismo paquete. Se representa con el carácter tilde ~

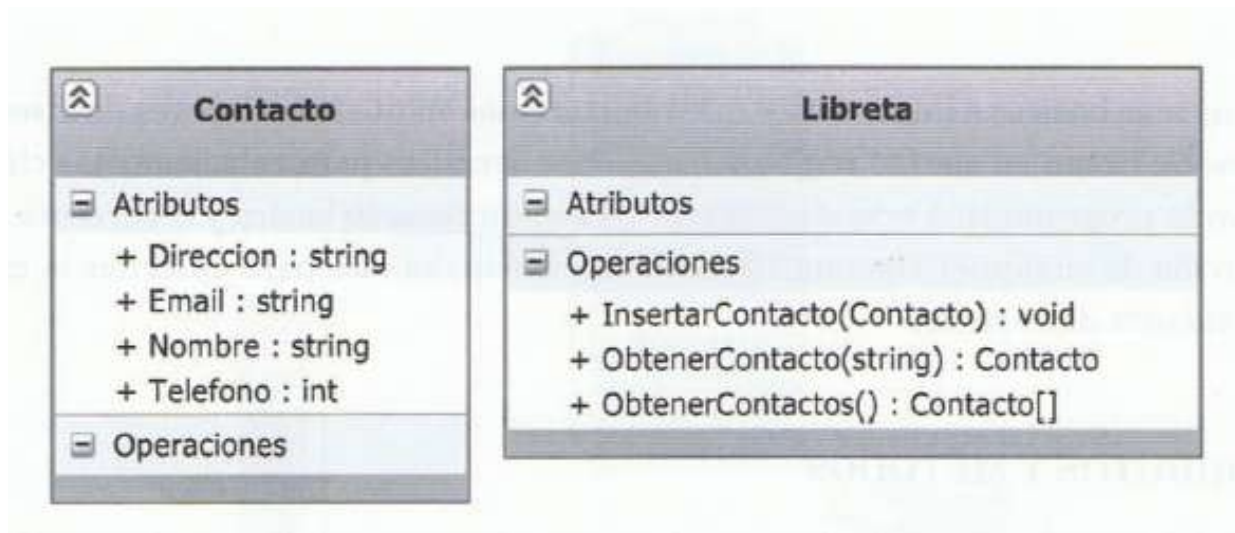
2.1.2 MÉTODOS

- ▶ Un método, también llamado operación, es la implementación de un servicio de la clase.
- ▶ Igual que los atributos los métodos pueden ser:
 - ▶ public: El método es accesible desde todos lados (desde dentro y fuera de la clase). Se representa con un signo +
 - ▶ private: Sólo los métodos de la clase pueden acceder a él. Se representa con un signo -
 - ▶ protected: El método puede ser accedido por métodos de la clase, además de los métodos de las subclases que deriven. Se representa con la almohadilla #
 - ▶ Package: El método empaquetado o paquete solo es visible en las clases del mismo paquete. Se representa con el carácter tilde ~

- Dada la clase Artículo que puedes ver a continuación, ¿Cuáles son los métodos y atributos públicos, privados, protegidos y empaquetados?



- Es posible tener un diagrama de clases en el que no se tenga ningún método o ningún atributo y ser perfectamente válido.



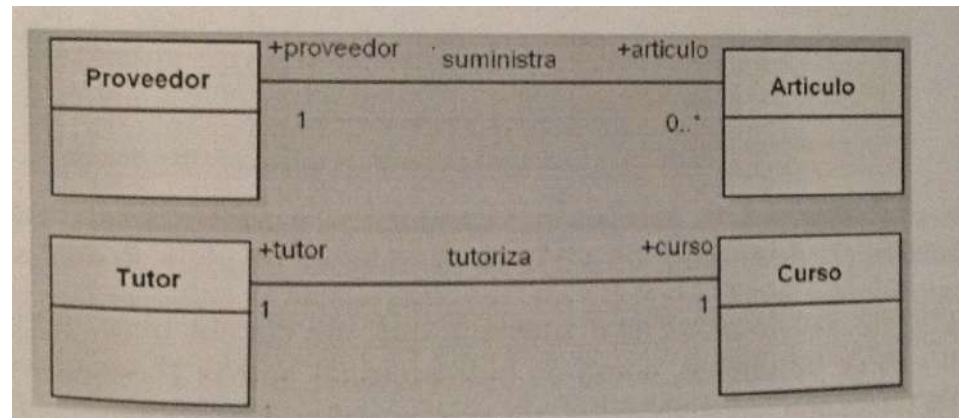
2.2 RELACIONES

- ▶ En el mundo real muchos objetos están vinculados o relacionados entre sí.
- ▶ Los vínculos se corresponden con asociaciones entre los objetos (por ejemplo, el vínculo existente entre un alumno y un curso que está matriculado o el vínculo entre un profesor y el centro en el que trabaja).
- ▶ En UML, estos vínculos se describen mediante asociaciones, de igual modo que los objetos se describen mediante clases.
- ▶ Las asociaciones tienen un nombre y como ocurre con las clases este es un reflejo de los elementos de la asociación.
- ▶ También poseen una cardinalidad que se llama multiplicidad que representa el número de instancias de una clase que se relacionan con las instancias de otra clase (esta multiplicidad es similar a la cardinalidad utilizada en el MER)

- Para expresar las multiplicidades mínima y máxima se utiliza la siguiente notación

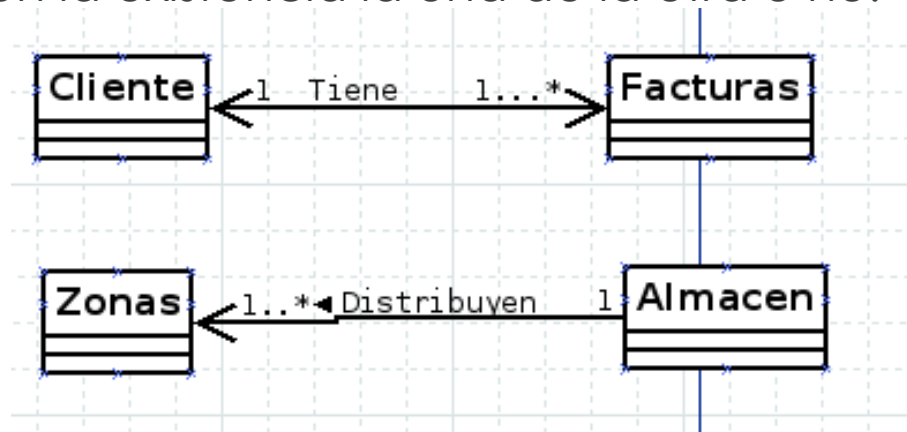
Notación	Cardinalidad/Multiplicidad
0..1	Cero o una vez
1	Una y solo una vez
*	De cero a varias veces
1..*	De una a varias veces
M..N	Entre M y N veces
N	N veces


- ▶ Primer Ejemplo. Tenemos la asociación suministra entre Proveedor y Artículo. Un proveedor suministra 0 o muchos artículos y un artículo es suministrado por un proveedor.
- ▶ Segundo Ejemplo. Tenemos la asociación tutoriza entre tutor y curso. Un tutor tutoriza a un curso y un curso es tutorizado por un tutor (1 a 1)



2.2.1 ASOCIACIÓN

- ▶ La asociación es la más básica de las relaciones.
- ▶ Se representa mediante una flecha.
- ▶ Estas pueden ser bidireccional o unidireccionales, dependiendo de si ambas conocen la existencia la una de la otra o no.



- 
- ▶ La primera asociación Tiene muestra que un cliente tiene muchas facturas, y la factura es de un cliente, como es bidireccional ambas clases conocen su existen, ambas clases son navegables (desde una factura puedo obtener los datos de un cliente y desde un cliente a sus facturas).
 - ▶ La segunda asociación, un almacén distribuye artículos en varias zonas. La asociación es unidireccional, solo la clase origen Almacén conoce la existe de la clase destino Zonas, por tanto, desde Almacén a Zonas es navegable, en cambio Zonas a Almacén no es navegable.
 - ▶ Podemos ver el código generado a las asociaciones en las siguientes diapositivas (HashSet se utiliza para guardar colecciones de objetos)

```
public class Cliente { // 1 cliente tiene muchas facturas (utiliza HashSet)
    public HashSet<Facturas> facturas = new HashSet<Facturas>();
    public Cliente() { // constructor
    }
    public HashSet<Facturas> getFacturas() {
        return this.facturas;
    }
    public void setFacturas(HashSet<Facturas> newFacturas) {
        this.facturas = newFacturas;
    }
}

public class Facturas { // 1 factura es de un cliente
    public Cliente cliente = null;
    public Facturas() { // constructor
    }
    public Cliente getCliente() {
        return this.cliente;
    }
    public void setCliente(Cliente newCliente) {
        this.cliente = newCliente;
    }
}
```

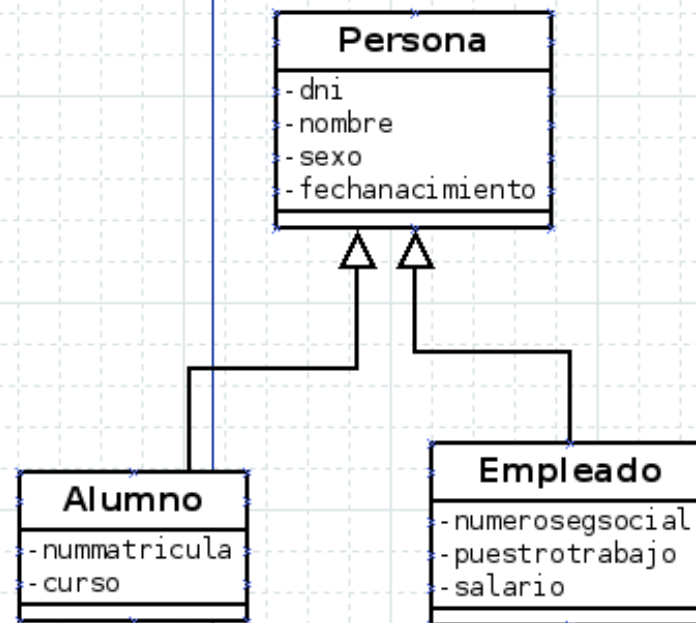
```
public class Zonas { // no sabe de la existencia de Almacen
    public Zonas() { // constructor
    }
}

public class Almacen { // 1 almacén distribuye en muchas zonas (HashSet)
    public HashSet<Zonas> zonas = new HashSet<Zonas>();
    public Almacen() { // constructor
    }
    public HashSet<Zonas> getZonass() {
        return this.zonas;
    }
    public void setZonass(HashSet<Zonas> newZonass) {
        this.zonas = newZonass;
    }
}
```

2.2.2 HERENCIA

- ▶ Las clases con atributos y operaciones comunes se pueden organizar de forma jerárquica mediante la herencia.
- ▶ La herencia es una abstracción importante para compartir similitudes entre clases, donde todos los atributos y operaciones comunes a varias clases se pueden compartir por medio de una superclase, una clase más general. Las clases más refinadas se conocen como las subclases.
- ▶ Para representar esta asociación se utiliza una flecha. El extremo de la flecha apunta a la superclase





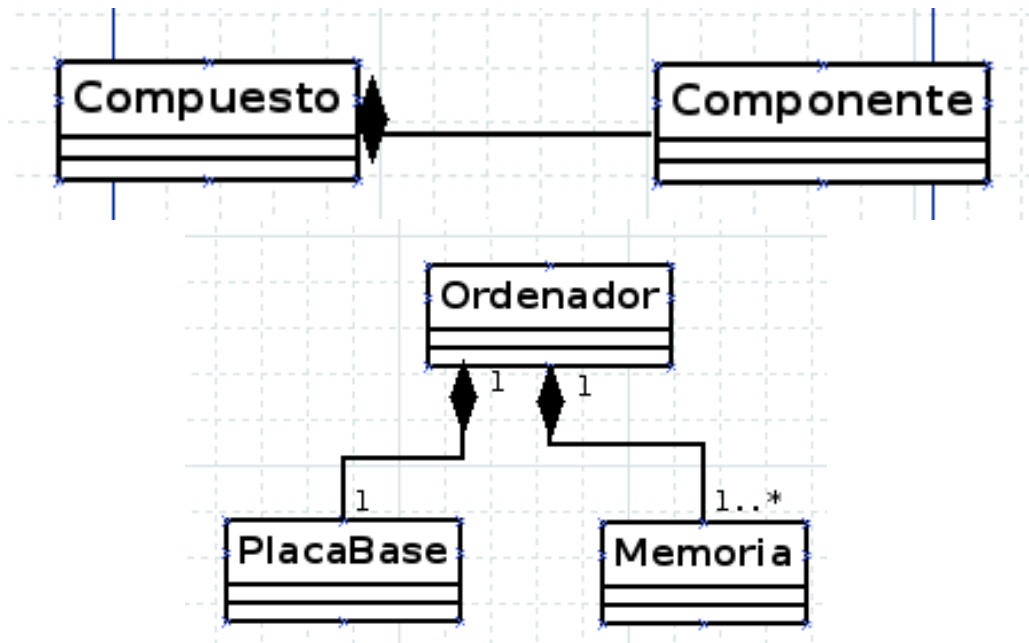
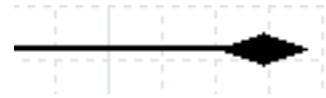
- El código java generado para estas clases sería el siguiente:

```
public class Persona {  
    private int dni;  
    private char nombre;  
    private char sexo;  
    private char fechanacimiento;  
    public Persona(){  
    }  
}  
  
public class Alumno extends Persona {  
    private int nummatricula;  
    private int curso;  
    public Alumno(){  
    }  
}  
  
public class Empleado extends Persona {  
    private char numerosegnsocial;  
    private char puestostrabajo;  
    private int salario;  
    public Empleado(){  
    }  
}
```

2.2.3 COMPOSICIÓN

- ▶ Un objeto puede estar compuesto por otros objetos, en estos casos nos encontramos ante una asociación entre objetos llamada **Asociación de composición**.
- ▶ Esta asocia un objeto complejo con los objetos que lo constituyen, es decir, sus componentes.
- ▶ Existen dos formas de composición, fuerte o débil. La fuerte se conoce como **composición** y la débil como **agregación**.
- ▶ En la composición fuerte los componentes constituyen una parte del objeto, y estos no pueden ser compartidos por varios objetos compuestos. Por tanto, la cardinalidad máxima, a nivel del objeto compuesto, es obligatoriamente uno.
- ▶ La supresión del objeto compuesto comporta la supresión de los componentes.

- Se representa por una línea con un rombo relleno



- El código java generado para la clase Ordenador, con los getter y los setter sería el siguiente:

```
* @author guillermopalazoncano
*/
public class Ordenador {

    public HashSet<Memoria> memorias = new HashSet<Memoria>();
    public PlacaBase placa = null;

    public Ordenador() { // constructor
    }

    public HashSet<Memoria> getMemorias() {
        return memorias;
    }

    public void setMemorias(HashSet<Memoria> memorias) {
        this.memorias = memorias;
    }

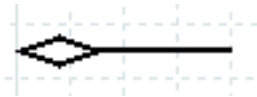
    public PlacaBase getPlaca() {
        return placa;
    }

    public void setPlaca(PlacaBase placa) {
        this.placa = placa;
    }

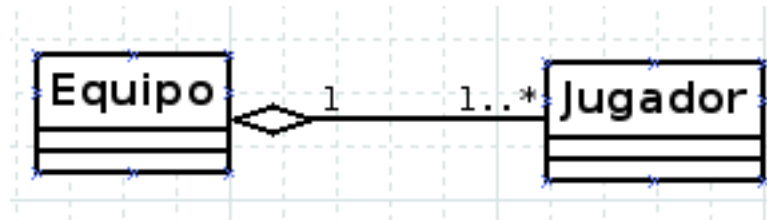
}
```

2.2.4 AGREGACIÓN

- ▶ La agregación es una composición más débil, en este caso los componentes pueden ser compartidos por varios compuestos y la destrucción del compuesto no implica la destrucción de los componentes.
- ▶ La agregación se da con mayor frecuencia que la composición, por lo que en muchas ocasiones se suele utilizar en las primeras fases la agregación y luego determinar si es una composición y realizar el cambio.
- ▶ Se representa con una línea con un rombo vacío



- En la siguiente asociación de agregación entre clase Equipo y la clase jugador, el equipo está compuesto por jugadores, sin embargo, el jugador puede jugar también en otros equipos. Si desaparece el equipo, el jugador no desaparece.

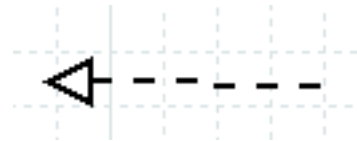


- En la siguiente tabla podemos ver las diferencias entre agregación y composición

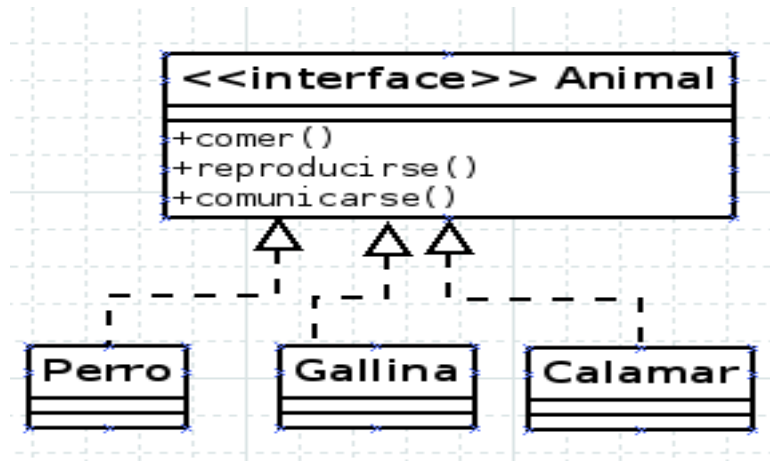
	Agregación	Composición
Símbolo		
Varias asociaciones comparten los componentes	SI	NO
Destrucción de los componentes al destruir el compuesto	NO	SI
Cardinalidad del compuesto	Cualquiera	0..1 o 1

2.2.5 REALIZACIÓN

- ▶ RECUERDA: Una clase Interfaz es una clase totalmente abstracta, no tiene atributos y todos sus métodos son abstractos y públicos, sin desarrollar. Estas clases no desarrollan ningún método.
- ▶ Una relación de realización es la relación de herencia existente entre una clase interfaz y la subclase que implementa esa interfaz.
- ▶ Esta relación de herencia se representa gráficamente mediante una flecha con línea discontinua en lugar de una línea completa.



- A continuación tenemos una asociación de realización entre una clase interfaz Animal y las clases Perro, Gallina y Calamar. Se considera que cualquier animal come, se reproduce y se comunica pero cada tipo de animal lo hace de manera diferente.



- Así sería el código resultante (el mismo código de Calamar sería también para Perro y Gallina)

```
/**
 *
 * @author guillermopalazoncano
 */
public interface Animal {
    public void comer();
    public void comunicarse();
    public void reproducirse();
}
```

```
/**
 *
 * @author guillermopalazoncano
 */
public class Calamar implements Animal {

    @Override
    public void comer() {
    }

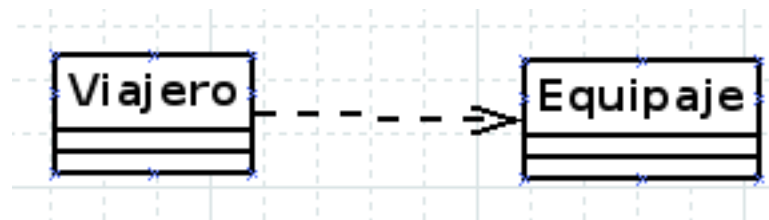
    @Override
    public void comunicarse() {
    }

    @Override
    public void reproducirse() {
    }

}
```

2.2.6 DEPENDENCIA

- ▶ Es una relación que se establece entre dos clases cuando una clase usa la otra, es decir, que la necesita para su cometido (las instancias de la clase se crean y se emplean cuando se necesitan).
- ▶ Se representa con una flecha sin rellena discontinua que va desde la clase utilizadora a la clase utilizada. Con esto indicamos que un cambio en la clase utilizada puede afectar al funcionamiento de la clase utilizadora, pero no al contrario.



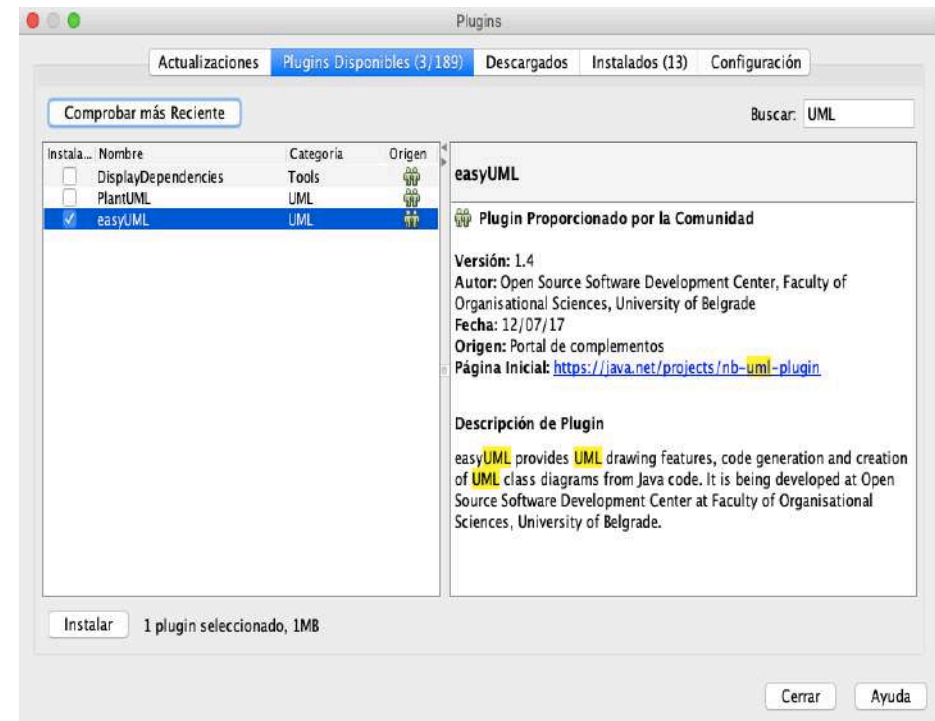
3. HERRAMIENTAS PARA EL DISEÑO DE DIAGRAMAS

- ▶ Existen muchas herramientas independientes para el diseño de diagramas de clase como pueden ser ArgoUML, UMLPAD, DIA o incluso algunas herramientas on-line como puede ser <https://www.draw.io/> o <https://www.lucidchart.com>
- ▶ No obstante, los IDE suelen incorporar sus propias herramientas UML o tienen la posibilidad de incorporar mediante plugins herramientas que satisfagan esta necesidad.
- ▶ A lo largo de esta unidad vamos a ver como diseñar estos diagramas de clases desde NetBeans.

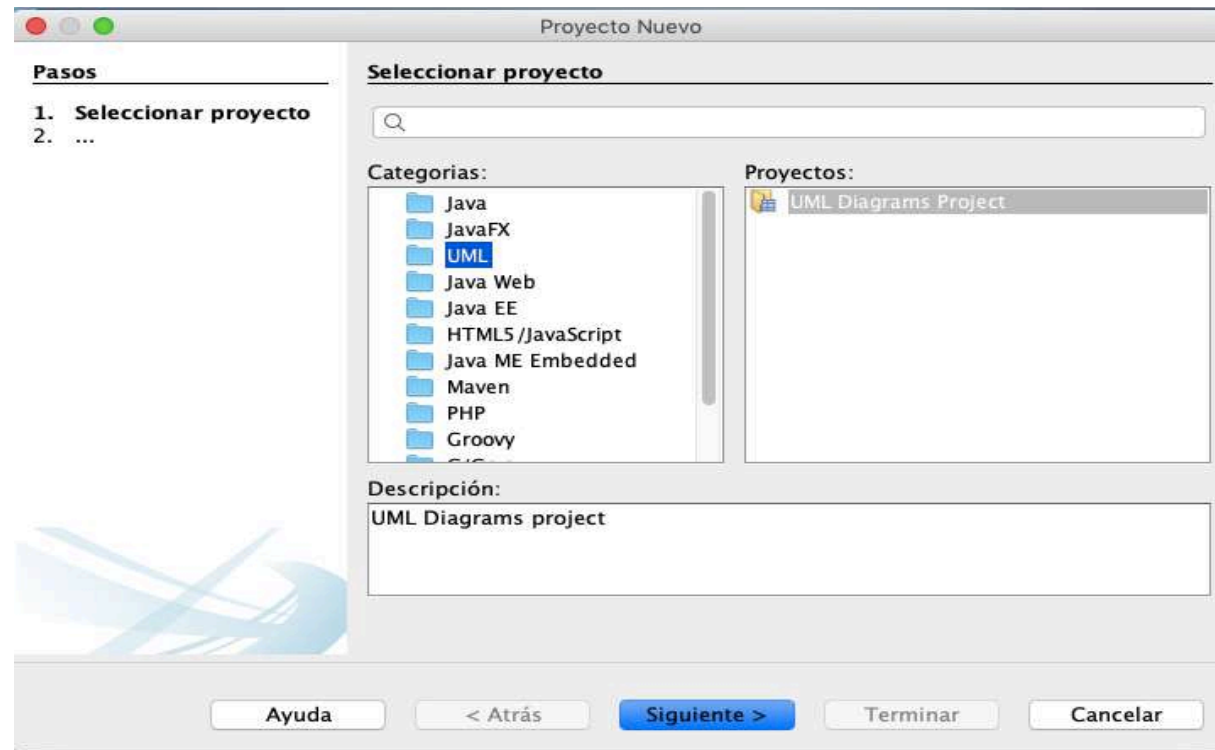
- NetBeans en su instalación normal no viene instalado para trabajar con UML. Podemos pulsar la opción de crear un nuevo proyecto y vemos que entre las opciones de proyecto a crear no se encuentra UML



- ▶ Por tanto, debemos recurrir a un plugin para instalar la correspondiente utilidad.
- ▶ Existen muchos plugins para NetBeans. Por su sencillez y porque viene preinstalado ya en la versión de Netbeans y por tanto no tenemos que descargar nada vamos a utilizar easyUML.
- ▶ Para su instalación nos iremos a la opción de menú Herramientas → Plugins y haciendo una búsqueda por UML será una de las opciones disponibles.

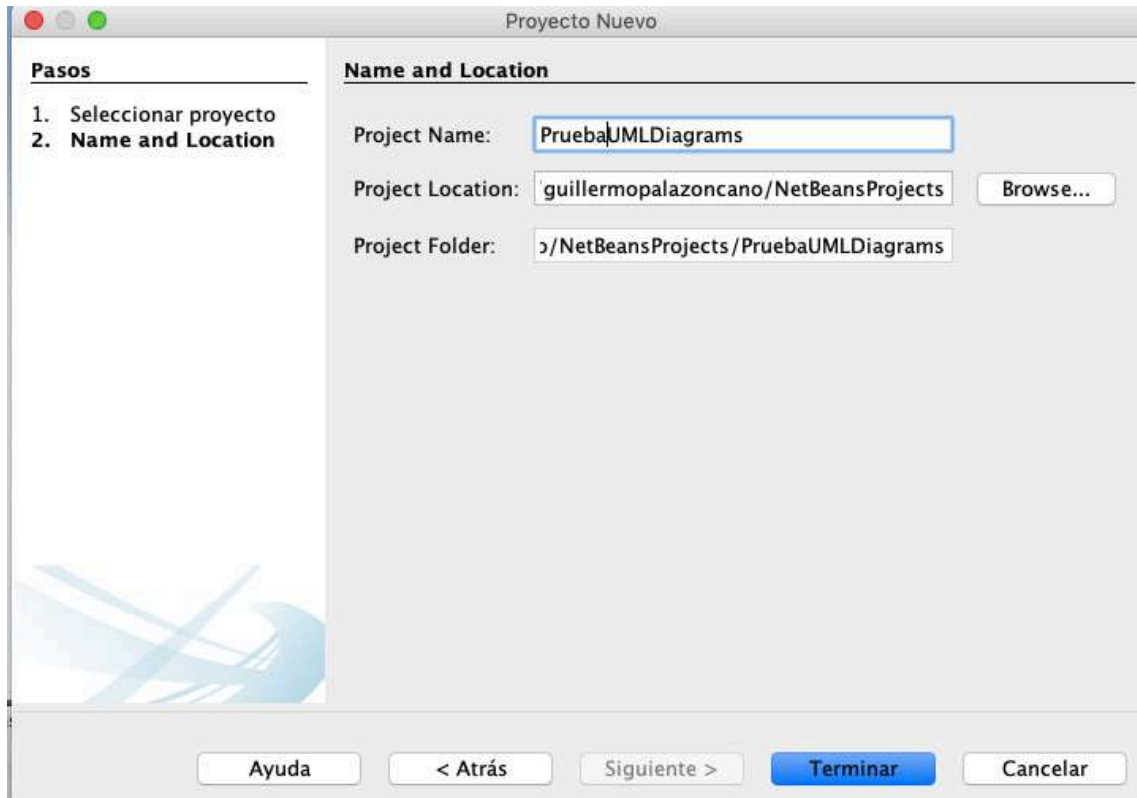


- Su instalación será igual que la de cualquier Plugin que ya hemos realizado y tras completar la misma y reiniciar el IDE nos debe aparecer la opción UML a la hora de crear un nuevo proyecto.



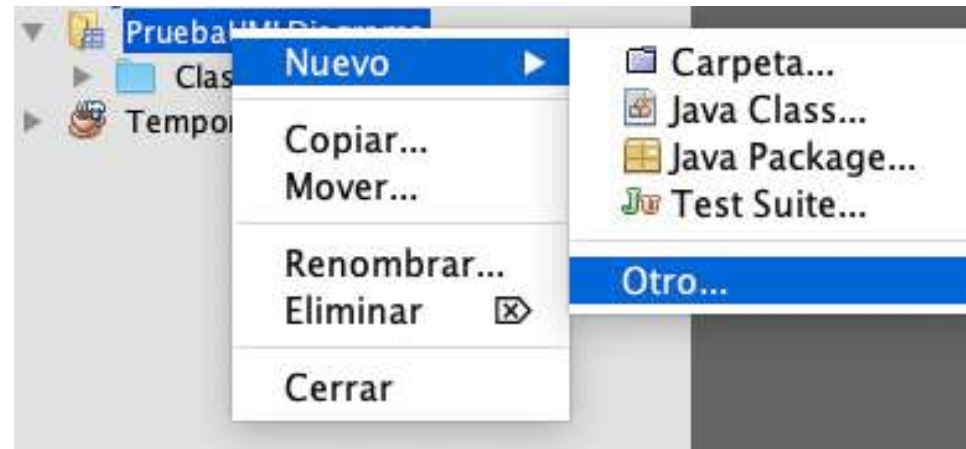
4. GENERACIÓN DE CÓDIGO A PARTIR DE DIAGRAMAS DE CLASES

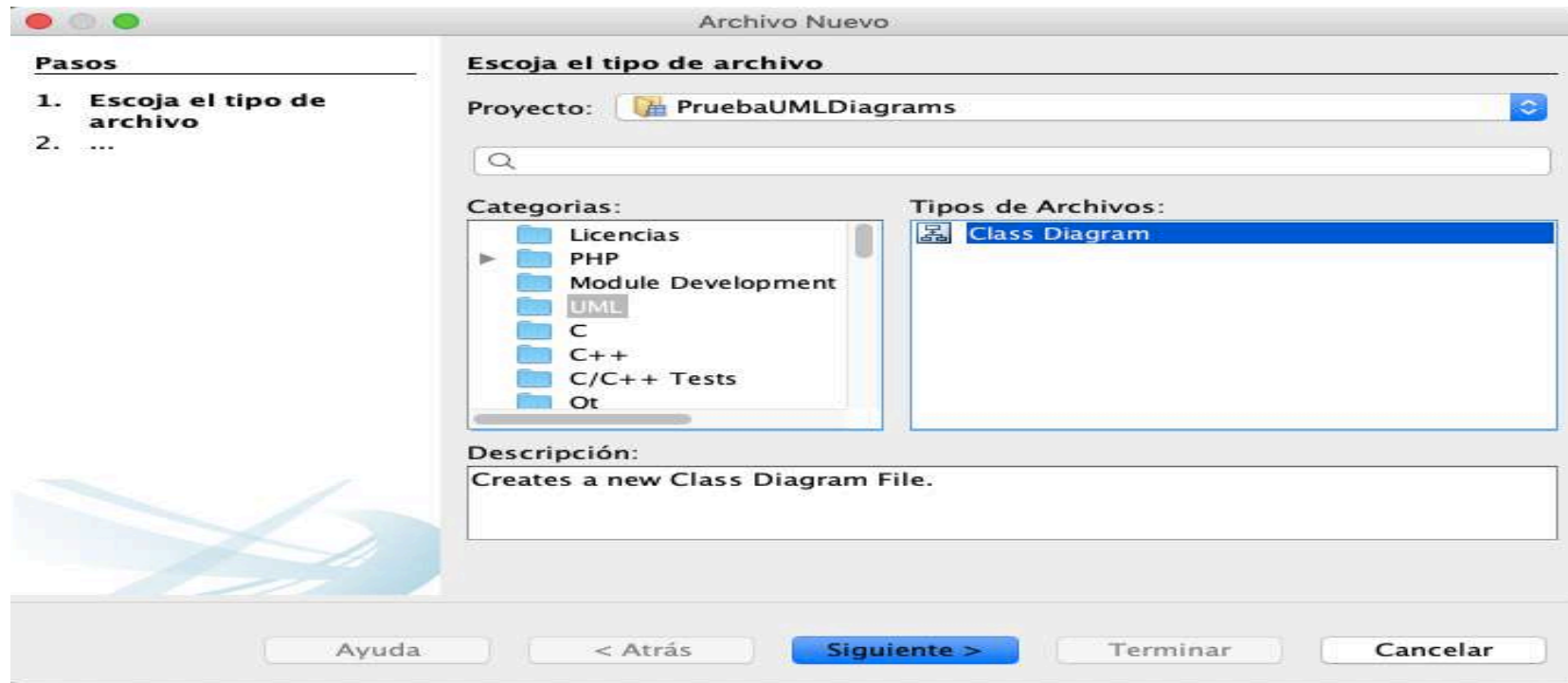
- ▶ Lo primero que vamos a hacer es aprender a generar diagramas de clase con easyUML.
- ▶ Lo primero que vamos a hacer es generar un proyecto **del tipo UML** que he denominado PruebaUMLDiagrams



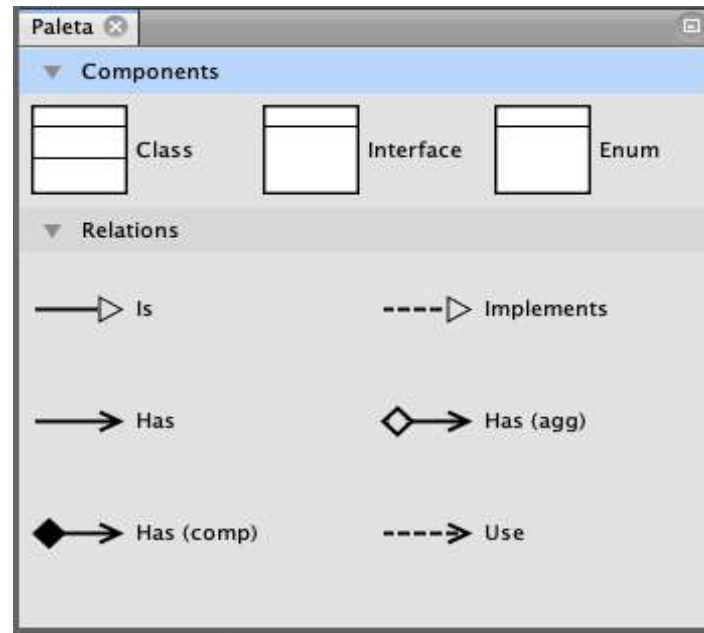
4.1 GENERACIÓN DEL DIAGRAMA DE CLASES

- Lo siguiente es generar dentro de ese proyecto, un diagrama de clases. Para ello hacemos clic derecho sobre el proyecto → Nuevo → Class diagram, si no esta visible esta opción, presionas en “otro...”, en categoría buscas UML y en tipo de archivo “Class diagram”, a continuación escribes el nombre del diagrama (en mi caso PruebaDiagrama) y clic en “terminar”.





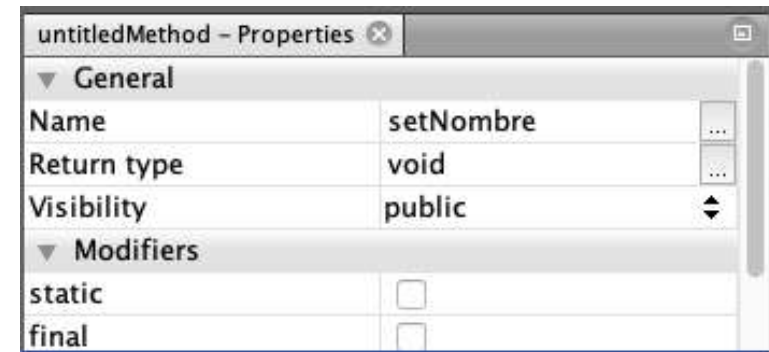
- ▶ Nos aparece una paleta con los principales componentes que tiene un diagrama de clases.
- ▶ En el caso de las relaciones:
 - ▶ Is: Herencia
 - ▶ Implements: Realización
 - ▶ Has: Asociación
 - ▶ Has (agg): Agregación
 - ▶ Has (comp): Composición
 - ▶ Use: Dependencia



- ▶ Para crear una clase, se arrastraría el componente de la paleta correspondiente.
- ▶ 1. Para cambiar el nombre de la clase pulsamos doble clic sobre el nombre que indica por defecto e indicamos el nombre que deseamos para nuestra clase
- ▶ 2. Para indicar un atributo hacemos clic en su cuadro correspondiente e indicamos el tipo de la propiedad y el nombre de la misma. Se genera un cuadro en el que podemos modificar nuevamente esas propiedades e indicar la visibilidad del atributo (private, public, protected o package)




- ▶ 3. Para añadir un nuevo método hacemos igual que con los atributos, hacemos doble clic en el cuadro correspondiente e indicamos el tipo que retorna ese método (puede ser void y que no retorne nada) y el nombre del método. Entre paréntesis le indicamos los parámetros del método.
- ▶ Al igual que con los atributos esto se puede modificar posteriormente.
- ▶ También se pueden añadir cada una de estas cuestiones (más el método constructor) pulsando con el botón derecho sobre el nombre de la clase

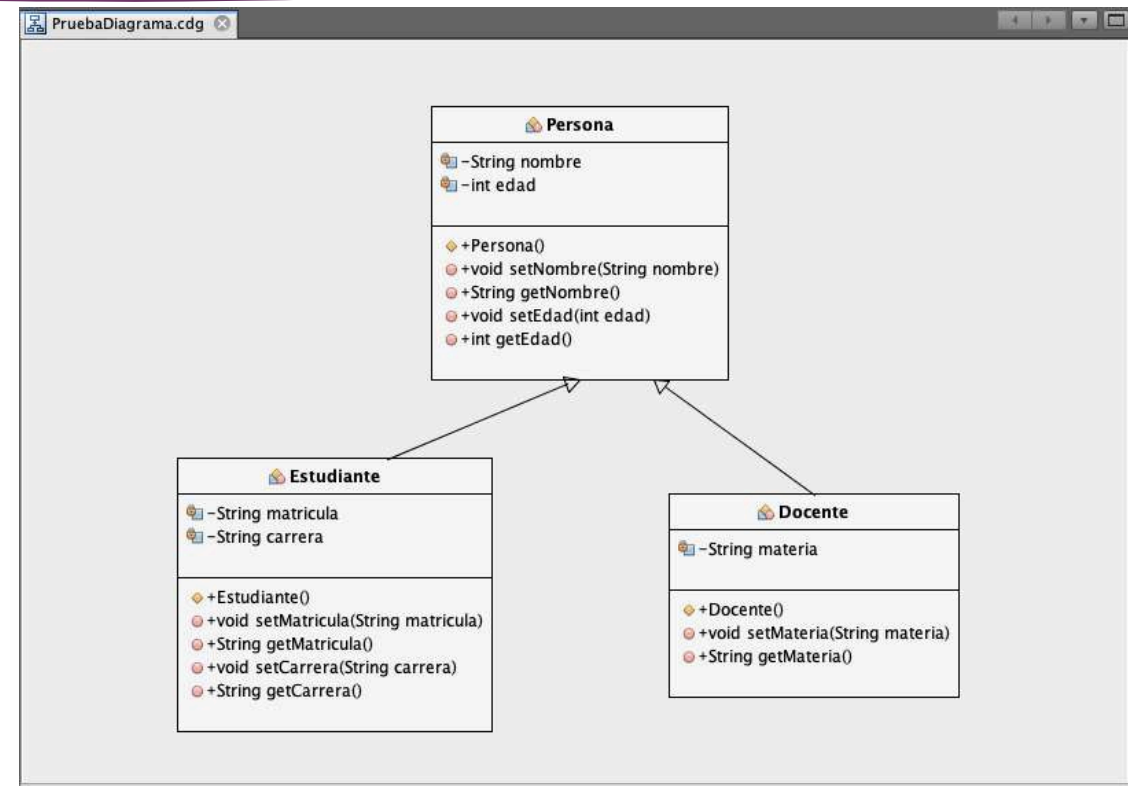


Add Constructor
Add Field
Add Method

Delete Class

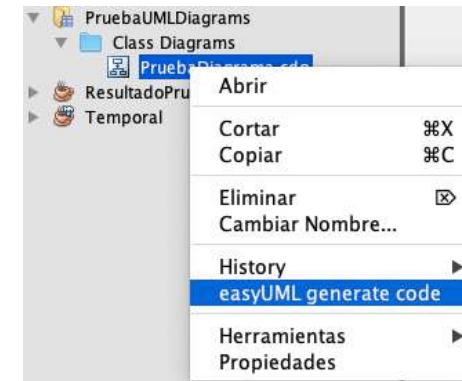
- 
- ▶ EJERCICIO. Genera un diagrama de clases en EasyUML que contenga las siguientes clases:
 - ▶ 1. Persona. Que tiene los atributos Nombre y edad y además del método constructor sin parámetros contiene los métodos getter y setter de los atributos correspondientes.
 - ▶ 2. Matrícula. Que será similar a Persona pero además tendrá los atributos de tipo String matrícula y carrera y los métodos correspondiente getter y setter (además del constructor).
 - ▶ 3. Docente. Que será similar a Persona pero además tendrá el atributo de tipo String materia y los métodos correspondientes getter y setter (además de su correspondiente constructor).

- SOLUCIÓN. Del enunciado del ejercicio podemos ver que Estudiante y Docente serán clases que heredan de Persona por lo que el diagrama de clases resultante será el siguiente.
- Si pulsamos sobre el citado diagrama podemos obtener el mismo en formato imagen (Exportando como imagen)

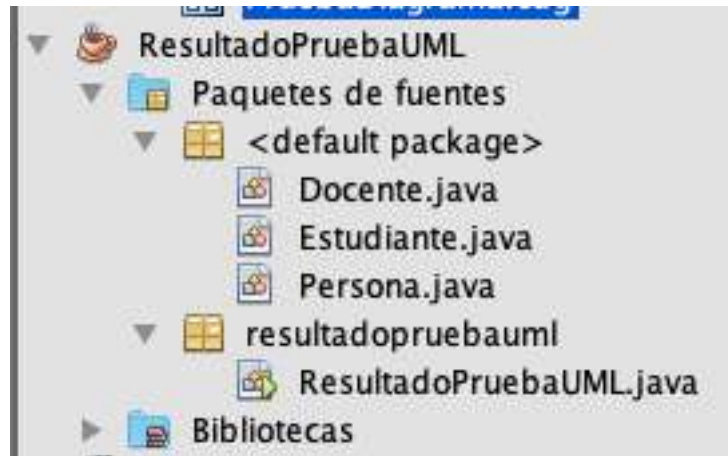


4.2 GENERACIÓN DE CÓDIGO

- ▶ Una vez que tenemos nuestra diagrama de clases vamos a generar el código correspondiente al mismo.
- ▶ En primer lugar vamos a generar un proyecto Java y no le vamos a agregar nada. En mi caso lo he llamado ResultadoPruebaUML.
- ▶ Volviendo a nuestro diagrama UML A haces clic derecho sobre el diagrama UML → presiona en “**easyUML generate code**” → en la ventana que aparece, buscas el proyecto java que acabas de crear y presionas “**generate code**”



- Podemos ver que en nuestro proyecto se ha creado un nuevo paquete que contiene las tres clases correspondientes.



- Podemos observar el código generado para cada una de las clases.

```
public class Persona {  
    private String nombre;  
    private int edad;  
    public Persona() {  
    }  
    public void setNombre(String nombre) {  
    }  
    public String getNombre() {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
    public void setEdad(int edad) {  
    }  
    public int getEdad() {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
}
```

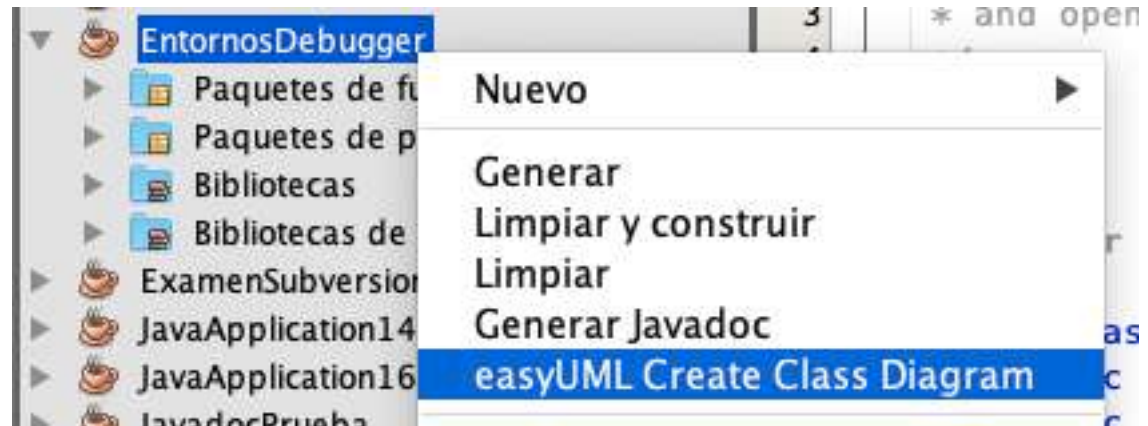
```
public class Estudiante extends Persona {  
    private String matricula;  
    private String carrera;  
    public Estudiante() {  
    }  
    public void setMatricula(String matricula) {  
    }  
    public String getMatricula() {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
    public void setCarrera(String carrera) {  
    }  
    public String getCarrera() {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
}
```

```
public class Docente extends Persona {  
    private String materia;  
  
    public Docente() {  
    }  
  
    public void setMateria(String materia) {  
    }  
  
    public String getMateria() {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
}
```

5. INGENIERÍA INVERSA


























- ▶ La mayoría de IDE y programas que trabajan con UML nos permiten realizar una ingeniería inversa de un proyecto con sus herramientas de modelado, pero ¿qué es la ingeniería inversa en el contexto de los diagramas de clase?
- ▶ La respuesta es bien sencilla: consiste en obtener de forma automática el diagrama de clases de un proyecto mediante su código.

- ▶ Para generar con easyUML el diagrama de clases correspondiente a un proyecto es muy sencillo.
- ▶ Para ello podemos elegir alguno de nuestros proyectos ya creados (en mi caso utilizaré el proyecto EntornosDebugger)
- ▶ Pulsamos sobre el botón derecho sobre el proyecto y nos saldrá la opción “easyUML Create Class Diagram)



- ▶ Tendremos que indicarle en que proyecto UML queremos que genere el diagrama de clases correspondiente, por lo que es necesario que exista al menos un proyecto UML (de lo contrario nos dará un error).
- ▶ Tras seleccionarlo pulsaremos en Create Class Diagram.
- ▶ De este proyecto y tras revisar su diagrama de clases podemos ver que hay dos clases que no se han generado. Esto es debido a que no tienen constructor, ni atributos ni ningún método salvo el método main por lo que el generador de diagrama de clases no las tiene en cuenta para generar este nuevo diagrama.
- ▶ Si pulsamos sobre la opción de generar la imagen correspondiente este sería el diagrama de clases que genera easy UML.

Ejercicio2_DividirNumeros

-  +final double num01
-  +final double num02
-  +final double num03
-  +final double num04
-  +final double num05
-  +final double num06
-  +final double num07
-  +final double num08
-  +final double num09
-  +final double num10
-  +final double num11
-  +final double num12
-  +final double num13
-  +final double num14
-  +final double num15
-  +final double num16
-  +final double num17
-  +final double num18
-  +final double num19
-  +final double num20
-  +final double num21
-  +final double num22
-  +final double num23
-  +final double num24
-  +static double resultado

-  +static void main(String[] args)
-  +static void multiplicar(double numero1, double numero2)

Ejercicio1

-  +static void main(String[] args)

Calculo

-  +static int calcularMayor(int x, int y)
-  +static int calcularMenor(int x, int y)
-  +static long calcularFactorial(int n)
-  +static int sumarDivisores(int n)
-  +static int introduceNumero(Scanner teclado, String cadena)