UNIDAD DIDÁCTICA 3

PROGRAMACIÓN ORIENTADA A OBJETOS

MÓDULO PROFESIONAL: PROGRAMACIÓN

CESUR
Tu Centro Oficial de FP

Índice

RESUMEN INTRODUCTORIO

A lo largo de esta unidad veremos la definición de clase y sus componentes como son los atributos y métodos, así como la representación de las relaciones entre ellas mediante diagramas de clase UML.

Se adquirirán conocimientos suficientes para instalar y manejar programas utilizados para la elaboración de diagramas de clase UML y se aprenderá a diferenciar entre las relaciones de herencia, agregación y composición principalmente, diseñando diagramas de clases sencillos en notación UML, que serán la base para que en próximas unidades se puedan traducir a código Java.

INTRODUCCIÓN

Los lenguajes de Programación Orientada a Objetos (POO) ofrecen medios y herramientas para describir los objetos manipulados por un programa. Más que describir cada objeto individualmente, estos lenguajes proveen una construcción (clase) que describe a un conjunto de objetos que poseen las mismas propiedades. Se podría decir que una clase es una plantilla a través de la cual se crean objetos de esa clase. Los paradigmas de la programación han ido evolucionando desde enfoques como el imperativo hasta la programación orientada a objetos.

CASO INTRODUCTORIO

En la empresa en la que trabajas tienes que realizar un nuevo proyecto en el cual el cliente exige que esté desarrollado en un lenguaje orientado a objetos debido a que le han informado de las importantes ventajas que estos tienen. Tienes que decidir si usar Java, C++ o C#.

Al finalizar el estudio de la unidad, serás capaz de reconocer las unidades que forman parte de la POO, identificarás los atributos y métodos que representan a una clase, conocerás los distintos tipos de relaciones existentes entre las clases y serás capaz de realizar representaciones del mundo real siguiendo el enfoque de la orientación a objetos con software destinado al diseño de diagramas de clase UML.

1. INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

Has estado valorando las fortalezas y debilidades de diferentes lenguajes orientados a objetos como son Java, C++ o C#, según las exigencias del nuevo cliente con el que trabajas actualmente.

Al final te has decantado por el lenguaje de programación Java para el desarrollo del nuevo producto software. Han sido muchos los factores a favor, entre los que destacan la portabilidad, rendimiento, escalabilidad y seguridad. Otro punto fuerte que ha influido considerablemente en dicha elección ha sido que la inversión en desarrollo Java es duradera y no se volverá obsoleta a corto plazo, prueba de ello es que lleva en el mercado mucho tiempo y todavía es muy utilizado.

La programación Orientada a Objetos en adelante POO es una metodología que basa la estructura de los programas en torno a clases y objetos.

Un objeto en el mundo real puede representar cualquier cosa: una silla, un coche, una persona, etc.

Si tenemos un conjunto de sillas podemos decir que son de la misma clase de objetos (objeto silla), sabemos que son sillas porque comparten unas propiedades comunes que las identifican como objetos pertenecientes al grupo silla, es decir, pueden tener patas, ser de un color (rojo, azul, verde, etc.), tener un tipo de tapicería (vinilo, microfibra, chenilla, etc.), incluso un uso distinto (silla de escritorio, silla de comedor, etc.).

A su vez cada una de estas sillas puede diferenciarse del resto de las otras sillas. Una silla puede ser de color madera, con 4 patas, tapiceria de vinilo, para comedor, otra de color blanco, con 3 patas, tapicería chenilla, para escritorio, etc. Dentro de la misma familia silla cada uno de los objetos silla puede tener características diferentes para cada una de sus propiedades como objeto.

Clases de sillas.
Fuente: https://industriaskerosti.com/blog/tipos-de-sillas-y-sus-nombres/

La POO ofrece ventajas como la modularidad, reutilización de código, estructuración de sistemas complejos y facilidad de mantenimiento. Java es un lenguaje orientado a objetos y utiliza clases y objetos.

¿SABÍAS QUE...?

Simula 67 fue el primer lenguaje de programación orientado a objetos. Simula 67 fue lanzado oficialmente por sus autores Ole Johan Dahl y Kristen Nygaard en mayo de 1967, en la Conferencia de Trabajo en Lenguajes de Simulación IFIO TC 2, en Lysebu cerca de Oslo.

1.1 Clase

La clase es la unidad de modularidad en la POO. La tendencia natural del individuo es la de clasificar los objetos según sus características comunes (clase). Por ejemplo, las personas que asisten a la universidad se pueden clasificar (haciendo una abstracción) en estudiante, docente, empleado e investigador. Una **clase** es una plantilla (descripción general de un conjunto de objetos) que define qué características tiene y cómo se comporta una determinada entidad, encapsula una serie de datos y funciones relacionadas en su interior.

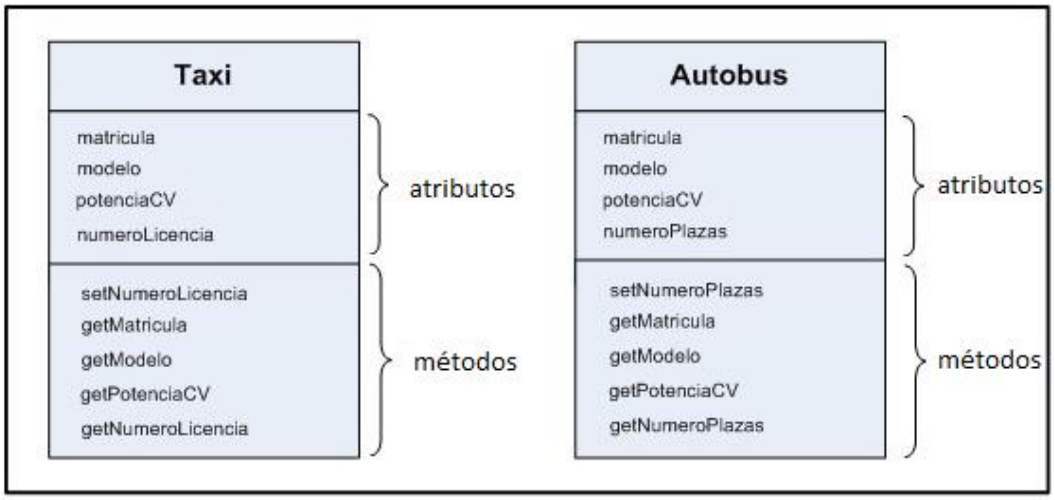


Diagrama UML de clases.

Una clase consta de **atributos** y **métodos** que resumen las características y el comportamiento comunes de un conjunto de objetos.

Como se ha dicho una clase es una plantilla de la cual se generarán objetos también conocidos como instancias de una clase. Mientras un **objeto** es una entidad concreta que existe en el tiempo y en el espacio, una **clase** representa solo una abstracción.

Todos los **objetos** de una clase tienen el mismo formato y comportamiento, son diferentes únicamente en los valores que contienen sus atributos. Todos ellos responden a los mismos mensajes.

En cuanto a las características generales, podemos destacar:

- **Una clase es un nivel de abstracción alto**. La clase permite describir un conjunto de características comunes para los objetos que representa.

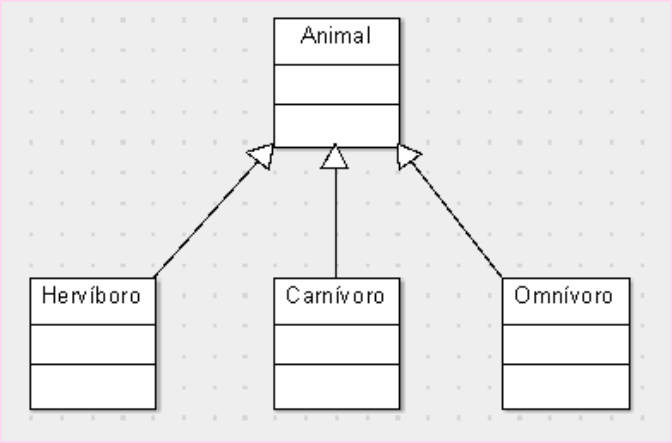
EJEMPLO PRÁCTICO

La clase Avión se puede utilizar para definir los atributos (tipo de avión, distancia, altura, velocidad de crucero, capacidad, país de origen, etc.) y los métodos (calcular posición en el vuelo, calcular velocidad de vuelo, estimar tiempo de llegada, despegar, aterrizar, volar, etc.) de los objetos particulares Avión que representa.

- **Las clases se relacionan entre sí mediante una jerarquía**. Entre las clases se establecen diferentes tipos de relaciones de herencia, en las cuales la clase hija (subclase) hereda los atributos y métodos de la clase padre (superclase), además de incorporar sus propios atributos y métodos.

EJEMPLO PRÁCTICO

Superclase: Clase Animal
Subclases de Animal: Clase Herbívoro, Carnívoro, Omnívoro



- Los **nombres o identificadores** de las clases deben colocarse en singular (`clase Animal, clase Hervíboro, clase Carnivoro, etc.`) y con notación `UpperCamelCase` (que se vio en la unidad anterior).

¿SABÍAS QUE...?

La manera más rápida para localizar una clase dentro de un contexto o enunciado de un problema a implementar es buscar los sustantivos que aparecen.

1.2 Atributo

Son los datos o variables que caracterizan a una clase y cuyos valores en un momento dado indican su estado.

Un atributo es una característica definida en una clase. Mediante los atributos se define información oculta dentro de la misma, que suele ser manipulada solamente por los métodos declarados en su interior Un atributo consta de un nombre y un valor. Cada atributo está asociado a un tipo de dato, que puede ser **simple** (entero, real, lógico, carácter, cadena de texto) o (array, registro, archivo, lista, etc.).

Su sintaxis general en Java se compone de las siguientes partes:

```
<Modo de Acceso o visibilidad> <Tipo de dato> <Nombre del Atributo>;
```

Los **modos de acceso** para **atributos** y **métodos** son:

Público*:* son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella. Este modo de acceso **también se puede** representar con el símbolo **+.**

Privado*:* sólo son accesibles dentro de la implementación de la clase. **También** se puede representar con el símbolo **–.**

Protegido*:* que son accesibles para la propia clase y sus clases hijas (subclases). También se puede representar con el símbolo **#**.

1.3 Método

Son las operaciones (acciones o funciones) que se aplican sobre los objetos y que permiten crearlos, cambiar su estado o consultar el valor de sus atributos. Representa el comportamiento o las acciones que puede realizar un objeto que pertenece a esa clase.

Los métodos constituyen la secuencia de acciones que implementan las operaciones sobre los objetos. La implementación de los métodos no es visible fuera del objeto.

Los métodos en Java se pueden expresar de la siguiente forma:

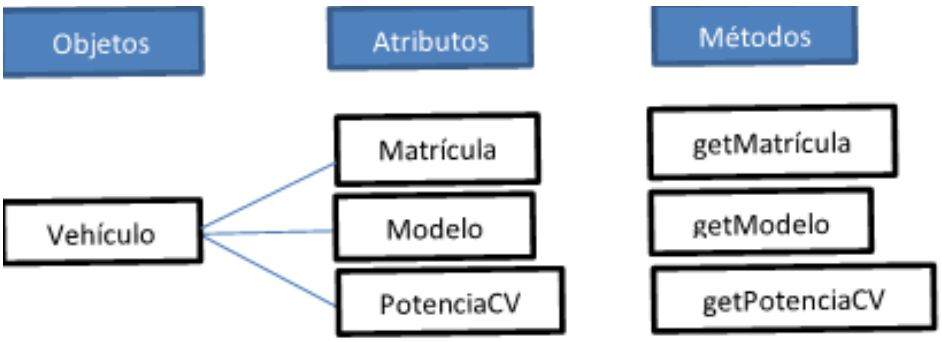
```
<Modo de Acceso o visibilidad> <Nombre del método>([Lista Parámetros]);

<Tipo de datos de retorno><Modo de Acceso o visibilidad> <Nombre del
método>([Lista Parámetros]);
```

donde los parámetros pueden ser opcionales

Ejemplo: Un rectángulo puede ser una clase caracterizada por los atributos largo y ancho y por varios métodos, entre otros calcular su área y calcular su perímetro.

Características Generales

- Cada **método** tiene un nombre, cero o más parámetros (por valor o por referencia) que recibe o devuelve un dato y un código que sería la implementación de dicho método.
- Los métodos pueden tener una visibilidad (igual que los atributos) que determinan desde qué partes del programa se puede acceder y utilizar métodos.
- Los métodos se ejecutan o activan cuando el objeto recibe un mensaje, enviado por un objeto o clase externo al que lo contiene, o por el mismo objeto de manera local.



Ejemplo de atributos y métodos en la clase Vehículo.

ARTÍCULO DE INTERÉS

Lee este interesante artículo sobre Simula, el primer lenguaje orientado a objetos:

1.4 Diagramas de clases UML

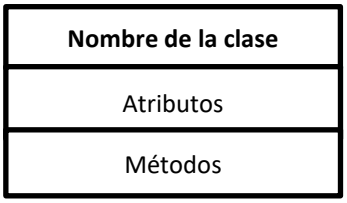
Un diagrama UML (Lenguaje Unificado de Modelado, en inglés Unified Modeling Language) ayuda a los equipos de desarrollo a planificar, diseñar y documentar sistemas software de una manera más efectiva.

Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. Ofrece un estándar para describir el modelo del sistema, incluyendo aspectos conceptuales como procesos, funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y compuestos reciclados.

Hay diferentes tipos de diagramas UML. El diagrama de clases UML, es muy utilizado en la modelización de sistemas orientados a objetos. Representa la estructura del sistema y muestra las clases, atributos, métodos y relaciones entre ellas.

Se recomienda utilizar este tipo de diagrama como un primer paso para descomponer la complejidad del problema.

Para representar a una clase utilizamos un rectángulo dividido en tres secciones: La primera contiene el nombre de la clase, la segunda los atributos y la tercera los métodos. Los atributos y métodos serán los miembros de la clase.



Representación de una clase en notación UML.

Se suele utilizar el siguiente formato para representar un atributo:

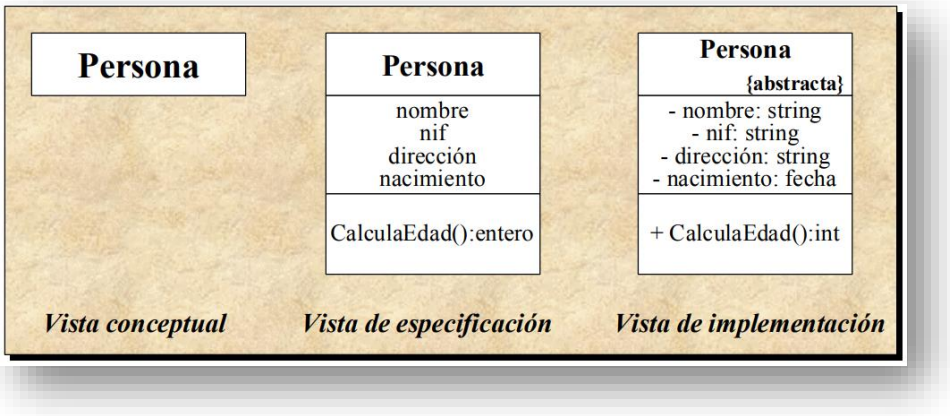
```
visibilidad nombre_atributo : tipo = valor-inicial { propiedades }
```

Y para los métodos este otro:

```
visibilidad nombre_metodo ({ parametros }) : tipo-devuelto {
                                propiedades }
```

Como se ha visto en el apartado anterior tanto los atributos como los métodos pueden incluir un tipo de visibilidad o modo de acceso que será identificado con un símbolo.

Tanto la sección de atributos como la de métodos pueden omitirse o mostrar solo aquellos atributos o métodos más representativos dependerá de las indicaciones a la hora de realizarlo. Según el detalle especificado en el diagrama tendremos una perspectiva conceptual, de especificación o de implementación.



Diferentes vistas de una clase.
Fuente: https://repositorio.grial.eu/bitstream/grial/353/1/DClase.pdf

VÍDEO DE INTERÉS

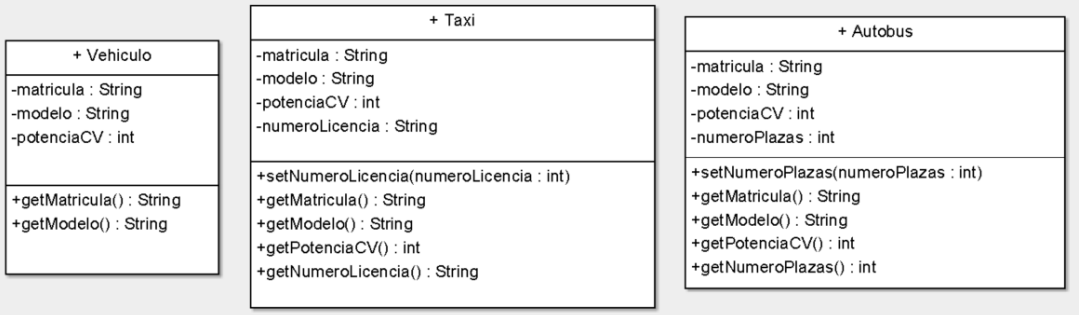
Visualiza las claves para realizar un diagrama de clases con notación UML y las diferentes relaciones que se pueden establecer entre clases:

RECUERDA

Un objeto es una entidad (tangible o intangible) que posee características y acciones que realiza por sí solo o interactuando con otros objetos. Consta de atributos y métodos.

EJEMPLO PRÁCTICO

Partiendo del ejemplo de clase Persona anterior, se plantea crear tres clases correspondientes a Vehículo, Taxi y Autobús. Cada una de ellas tendrá una serie de atributos y métodos que en unidades siguientes se ampliarán y explicarán.



Ejemplos de diagramas de clase UML.
Fuente: Elaboración propia.

1.5 Relación entre clase y objeto

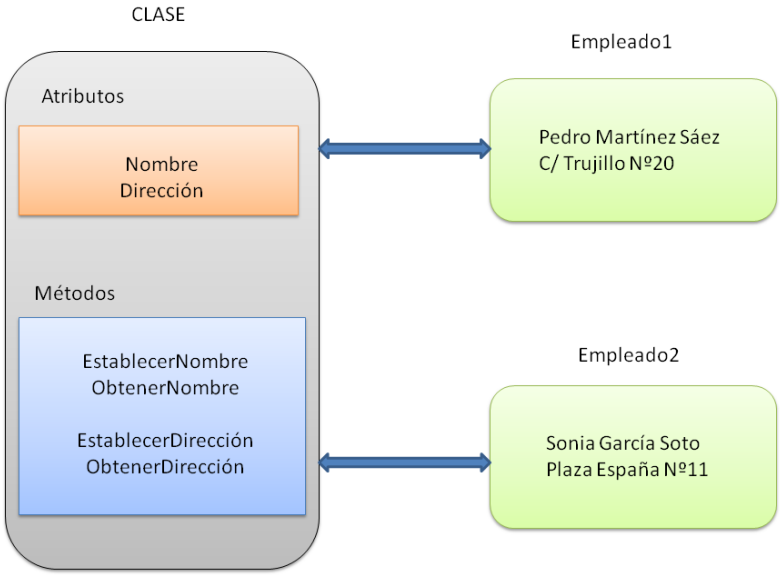
Algorítmicamente, las **clases son descripciones netamente estáticas o plantillas** que describen objetos. Su rol es definir nuevos tipos conformados por atributos y operaciones. Por el contrario, **los objetos son instancias particulares** de una clase. Las clases son una especie de molde de fábrica, con los que son construidos los objetos. Durante la ejecución de un programa sólo existen los objetos, no las clases.

La **declaración** de una variable de una clase **NO crea** el objeto.

La asociación siguiente: `<Nombre_Clase> <Nombre_Variable>;` (por ejemplo, `Rectángulo R`), no genera o **no crea automáticamente un objeto** `Rectángulo`. Sólo indica que R será una **referencia** o una variable de objeto de la clase Rectángulo.

La **creación de un objeto** debe ser indicada explícitamente por el programador, de forma análoga a como inicializamos las variables con un valor dado, sólo que para los objetos se hace a través de un **método Constructor**.

En esta imagen se puede ver la representación de una clase compuesta por 2 atributos y 4 métodos. De dicha clase se han instanciado dos objetos; el objeto `empleado1` y el objeto `empleado2`. Cada uno de estos objetos comparte los mismos tipos y nombres de atributos y acciones realizadas por los métodos (son objetos instanciados de la misma clase) sin embargo cada objeto es independiente del otro y puede almacenar distinta información en los atributos y por tanto los métodos pueden elaborar información de salida diferente.



Relación entre clase y objeto.

PARA SABER MÁS

Para saber más información sobre la programación orientada a objetos visita esta web:

ENLACE DE INTERÉS

En este enlace podrás encontrar más información sobre el concepto de Objetos y Clases en Java:

EJEMPLO PRÁCTICO

El siguiente código se encuentra almacenado en el archivo Calculadora.java.

Se trata de una clase llamada Calculadora que tiene definido un método llamado sumar que recibe dos parámetros de tipo entero, realiza la suma de ambos y retorna dicho valor.

```
package com.ejemplosUd3.calculadora;

public class Calculadora {
      public int sumar(int a, int b) {
            return a + b;
        }
}
```

En el siguiente archivo Main.java tenemos un método main que instancia un objeto llamado calculadora (primera letra con minúscula para referirnos a objetos y atributos) de la clase Calculadora (primera letra con mayúsculas para clases).

Al ser un objeto instanciado de la clase Calculadora dispone del método sumar. El valor retornado por dicho método se recoge en la variable de tipo entero llamada resultado y por último se imprime por pantalla.

Para llamar al método sumar solo tenemos que escribir el nombre del objeto seguido de un punto e inmediatamente el IDE nos mostrará un listado de atributos y métodos que forman parte del objeto, en este caso podemos ver que el método sumar está en la lista, lo escogemos e introducimos los parámetros que espera recibir.

```
package com.ejemplosUd3.calculadora;

public class Main {
    public static void main(String[] args) {
        Calculadora calculadora = new Calculadora();
        int resultado = calculadora.sumar(5, 3);
        System.out.println("El resultado de la suma es: " +
resultado);
    }
}
```

Cuando ejecutemos el archivo Main.java el resultado en pantalla será el siguiente:

El resultado de la suma es: 8.

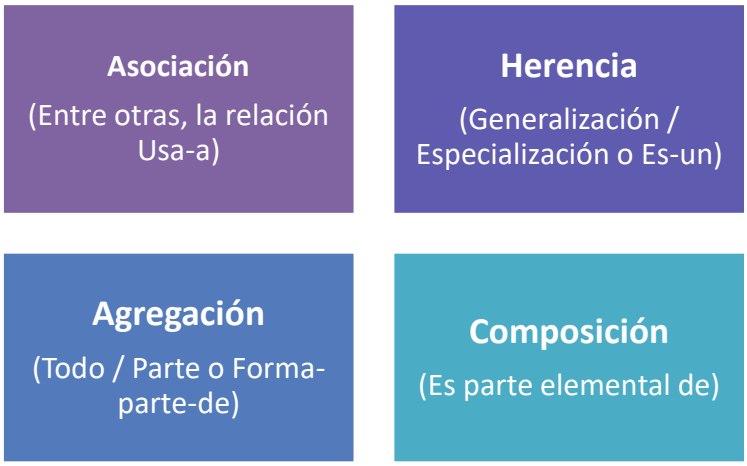
2. RELACIONES ENTRE CLASES

Tu jefe ha tenido una reunión con un cliente muy importante del sector bancario que ha solicitado a la empresa un sistema de gestión de cuentas bancarias. Los requisitos demandados por el cliente son varios, entre ellos que permita gestionar las cuentas bancarias de manera eficiente y segura.

Las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan **relacionar entre sí**, de forma que compartan atributos y métodos sin necesidad de rescribirlos.

La posibilidad de establecer **jerarquías** entre las clases es una característica que diferencia esencialmente a la programación orientada a objetos de la programación tradicional, ello debido fundamentalmente a que permite **extender y reutilizar el código existente** sin tener que reescribirlo cada vez que se necesite.

Los cuatro tipos de relaciones entre clases estudiados en esta unidad serán:



2.1 Relación de asociación («uso», usa, cualquier otra relación)

La relación de asociación se establece cuando dos clases tienen una dependencia de utilización, es decir, una clase **utiliza atributos y/o métodos de otra** para funcionar. Estas dos clases pueden ser clases independientes, por lo tanto, no necesariamente están en jerarquía, es decir, no necesariamente una es clase padre de la otra, a diferencia de las otras relaciones de clases.

Las asociaciones pueden ser binarias (relacionan dos clases) o n-arias (relacionan n clases).

Una relación binaria se representa mediante una línea que une las dos clases. Se tratará de una relación bidireccional cuando no hay punta de flecha por tanto ambas clases podrán llamar a métodos de la otra.

Cuando la línea que une las clases termina en punta de flecha se trata de una relación unidireccional donde la flecha apunta hacia la clase que es referenciada o utilizada por la otra clase.

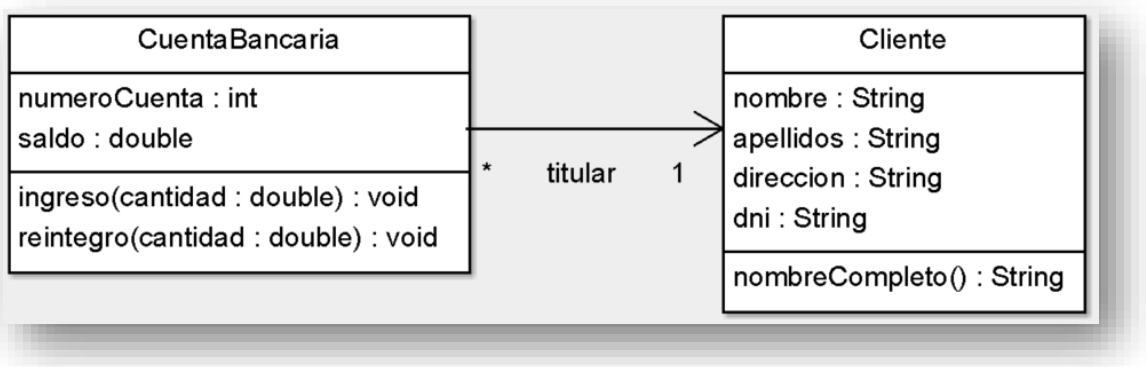
Generalmente existen dos roles en una relación de asociación, uno por cada dirección. Cada rol tiene asociada una cardinalidad (o multiplicidad) que indica cuantas instancias de una de las clases puede estar relacionada con una instancia de la otra clase. Se denota mediante la siguiente notación LI…LS (especifica un rango con un límite inferior a un límite superior respectivamente) que será el rango de cardinalidad permitido que puede asumir la asociación. Se puede usar * en el límite superior para representar una cantidad ilimitada (ejemplo: 3…*).

Un ejemplo de asociación sencillo puede ser la relación entre las entidades o clases Cliente y CuentaBancaria donde una CuentaBancaria "usa" los datos del Cliente para establecer a quién pertenece dicha CuentaBancaria (la dirección de la relación es de CuentaBancaria hacia Cliente), por tanto, CuentaBancaria podrá acceder a los atributos de Cliente y sus métodos.

CuentaBancaria contiene dos atributos que indican el número de cuenta y el saldo de esta y dispone de dos métodos; uno para realizar ingresos y otro para reintegros. Por otro lado la entidad Cliente almacena datos personales (nombre, apellidos, dirección y DNI) y dispone de un método llamado nombreCompleto() que puede servir para realizar cualquier funcionalidad relacionada con el nombre y los apellidos.

Para interpretar esta relación podemos empezar a leer de izquierda a derecha, es decir, una CuentaBancaria tendrá como titular a un único Cliente (cardinalidad del lado de la entidad Cliente igual a 1), ahora toca leer de derecha a izquierda… Un cliente puede ser titular de más de una cuenta bancaria (cardinalidad del lado de CuentaBancaria * que significa muchos, sin límite). En cada extremo se especifica la cardinalidad.

Ejemplos de cardinalidades en relaciones	
1	Solo una instancia de la clase.
0…1	Ninguna o como máximo solo una.
0…*	De cero a cualquier número, sin límite.
1…*	De 1 (no puede ser cero) a cualquier número, sin límite.
*	Equivalente a 0…*
Número inicial…número final Ej:1…6	Rango comprendido desde el número inicial y el número final, ambos incluidos. Ej.: mínimo una y como máximo 6. Cualquier valor incluido en el rango.
Número, número Ej:3,5	Solo los números indicados. Tendrá 3 o 5 instancias de la clase (solo uno de estos dos valores, no es rango).



Ejemplo de relación de asociación.
Fuente: Elaboración propia.

- La *CuentaBancaria* depende de *Cliente*.
- La *CuentaBancaria* está asociada a *Cliente*.
- La *CuentaBancaria* conoce la existencia de *Cliente,* pero *Cliente* desconoce que existe la *CuentaBancaria*.

Todo cambio en *Cliente* podrá afectar a *CuentaBancaria*.

Esto significa que *CuentaBancaria* tendrá como atributo un objeto o instancia de *Cliente* (su componente). La *CuentaBancaria* podrá acceder a las funcionalidades o atributos de su componente usando sus métodos.

En la próxima unidad veremos cómo se realiza la transformación de las entidades que intervienen en un diagrama de clases UML a código Java y como se incluyen estas instancias u objetos en la clase componente.

Ejemplo relación de asociación.

ENLACE DE INTERÉS

Amplia información y conoce más ejemplos sobre las relaciones de asociación en la POO visitando la web:

2.2 Relación de herencia (generalización / especialización, es un)

Es una forma de asociación en la que se establece una jerarquía entre clases, cada subclase contiene los atributos y métodos de una (herencia simple) o más superclases (herencia múltiple).

Mediante la herencia las instancias de una clase hija (o subclase) pueden acceder tanto a los atributos como a los métodos públicos y protegidos de la clase padre (o superclase).

Cada **subclase** o **clase hija** en la jerarquía es siempre una **extensión** (esto es, conjunto estrictamente más grande) de la(s) **superclase(s)** o **clase(s) padre(s)** y además incorpora atributos y métodos propios, que a su vez serán heredados por sus hijas.

La clase padre es considerada como una generalización de la clase hija, es decir, representa una clase más general que define características y comportamientos comunes que heredan las hijas.

Y la clase hija es considerada como una especialización de la clase padre, es decir es una clase más específica que hereda y reutiliza atributos y comportamientos(métodos) de la clase padre o superclase y además añade otras características adicionales y comportamientos propios exclusivos de esta clase hija.

Este tipo de relación de herencia favorece la reutilización de código, la estructura jerárquica y modular de un sistema.

Representación:

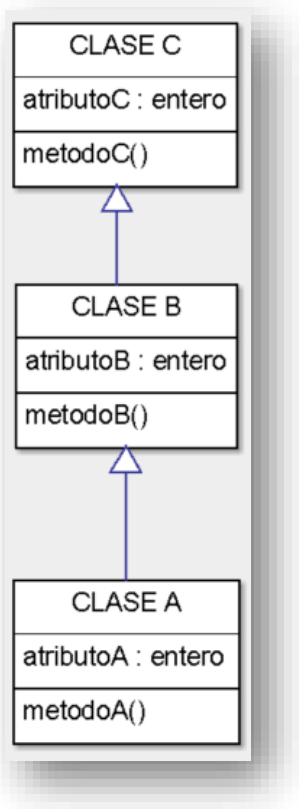
En un **diagrama de clases UML**, la herencia se representa mediante una flecha cuya punta es un triángulo vacío. Esta flecha apunta a la clase padre, es decir la flecha va desde la subclase o clase hija a la superclase o clase padre. A la clase padre llegarán tantas flechas como clases hijas tenga.

Se representa de la siguiente forma:

```
clase hija → clase padre  ó   subclase → superclase
```

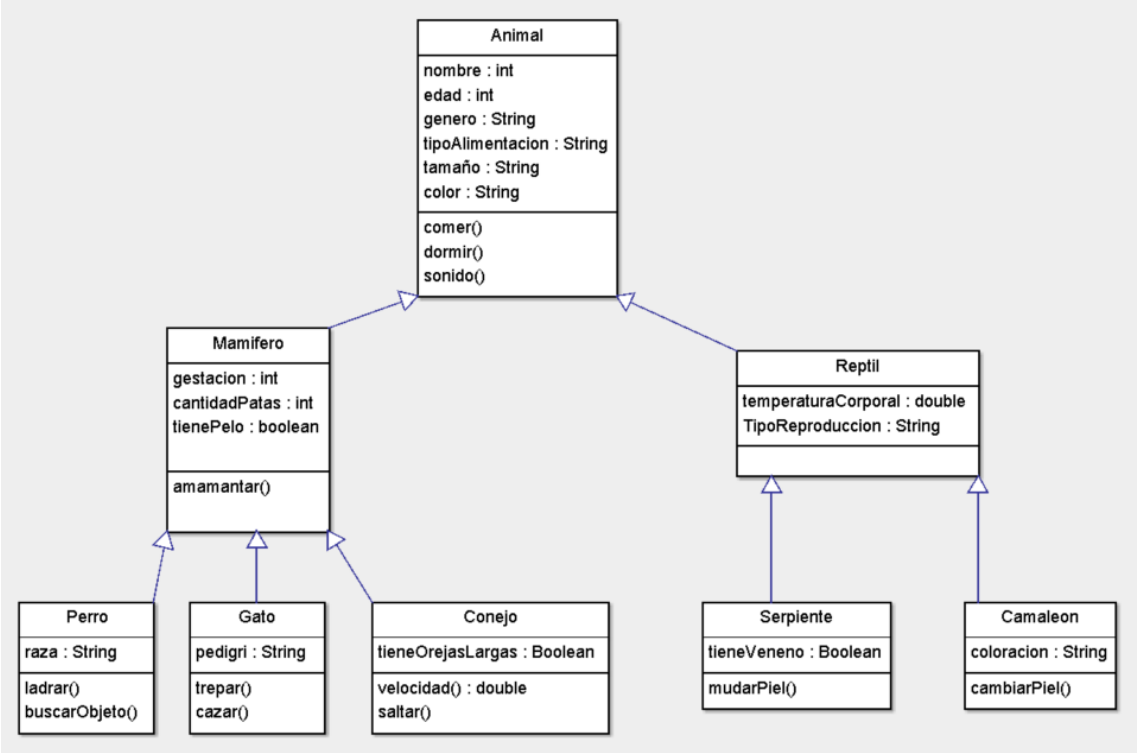
La representación de clases siguiente, indica que C es superclase de la clase B y B es superclase de A y a su vez, C es superclase de B y de A.

```
A→B→C
```



Relación de Herencia entre las clases A, B y C.

A continuación, se representa un diagrama de clases, algo más complejo sobre relaciones de herencia.



Relación de Herencia entre la Clase Animal y sus clases hijas.

En este ejemplo de herencia representado con notación UML, la clase Animal es padre o superclase directo de las clases Mamifero y Reptil. La clase Mamifero es superclase o padre directo de las clases Perro, Gato y Conejo y la clase Reptil es superclase o padre directo de las clases Serpiente y Camaleon. Y a su vez la clase Animal es superclase o padre de todas ellas.

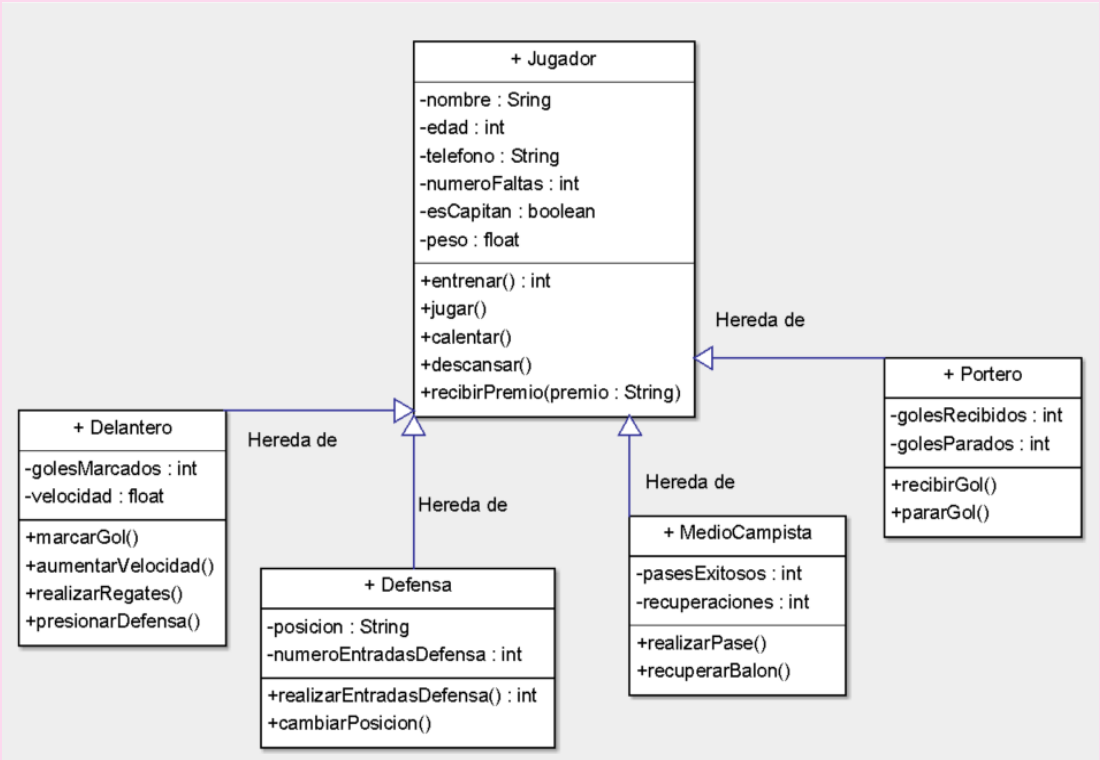
No se ha incluido el modificador de acceso o visibilidad suponemos que todos los atributos y métodos son públicos. (+)

¿SABÍAS QUE...?

Hay dos tipos de herencia: Simple y Múltiple. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. Java sólo permite herencia simple.

EJEMPLO PRÁCTICO

Se quiere desarrollar una aplicación software para gestionar los jugadores de un determinado equipo de fútbol. Para empezar con el diseño se han definido una serie de clases con atributos y métodos. Inicialmente se ha escogido una relación de herencia entre la entidad Jugador y las demás. Esta relación favorece que las clases hijas no tengan que declarar atributos y métodos que ya han heredado.

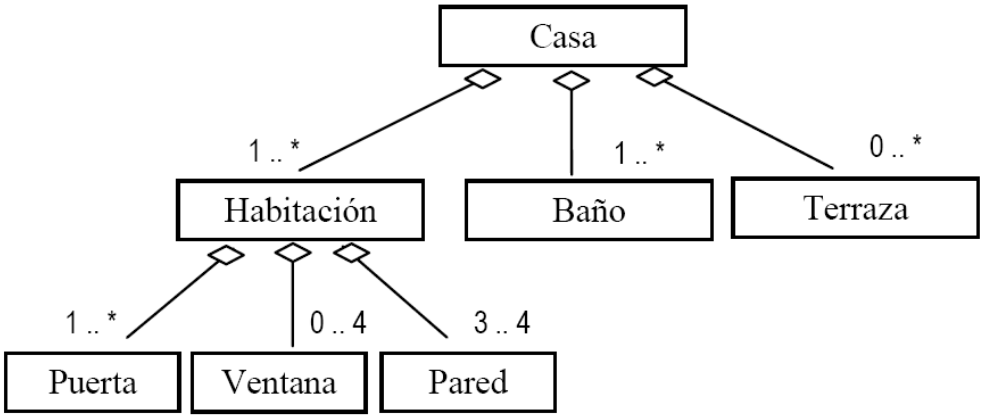


2.3 Relación de agregación (todo / parte, forma parte de)

La agregación es un tipo de asociación usada para representar que una clase es parte de otra clase. Se conocen como relaciones todo/parte/forma parte de.

El todo o compuesto es la clase que contiene a las otras clases y las clases que forman parte de un todo o compuesto se llaman componentes. En este tipo de relación la destrucción del todo o compuesto, no conlleva la destrucción de los componentes ya que las partes agregadas(componentes) son independientes del todo y pueden existir y ser usadas por otro todo o compuesto o por varios.

La relación de agregación se representa de forma habitual con un rombo blanco que se coloca en el extremo cercano a la clase que representa el todo.



Entidad Casa descrita en términos de sus componentes.

Este ejemplo está formado por entidades o clases que hacen el papel de "Todo" y otras con el rol de "Componentes" o "partes agregadas".

Entre las clases "Todo" tenemos, Casa y Habitación (a su vez es componente o parte agregada de Casa) y "componentes" serían Baño, Terraza, Puerta, Ventana y Pared. La anterior relación se puede interpretar leyendo de arriba abajo, de la siguiente forma:

Una Casa se compone de:

- Como mínimo de 1 **Habitación** o puede tener un número indeterminado de habitaciones (no hay límite en el máximo, 1….*)
- Como mínimo de 1 **Baño** o puede tener un número indeterminado de Baños (no hay límite en el máximo, 1….*)
- Puede que no tenga **Terraza** o puede que tenga un número indeterminado de Terrazas (no hay límite en el máximo, 0….*)

Una Habitación se compone de:

- Como mínimo de 1 **Puerta** o puede tener un número indeterminado de Puertas (no hay límite, 1….*)
- Puede que no tenga **Ventana** o que tenga Ventanas pero no podrán ser superiores en número a 4, (0….4)
- Tendrá 3 **Paredes** o 4 Paredes, ni una más ni una menos, (3….4)

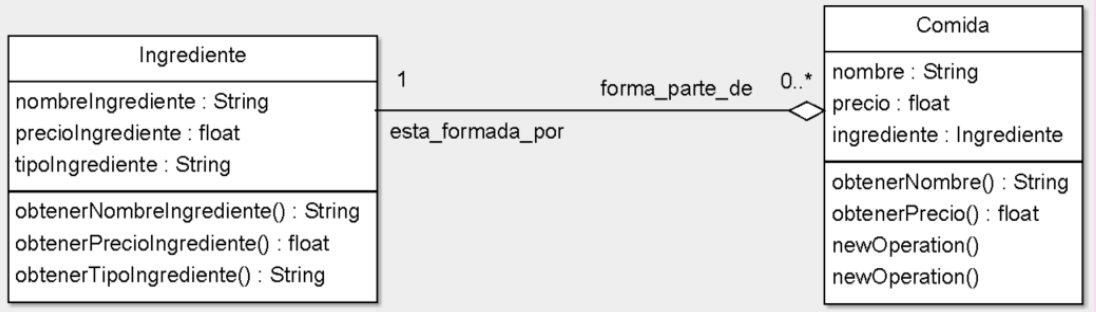
También se podría hacer una lectura de abajo a arriba, por ejemplo, *una Puerta o más de una Puerta "forma parte de" LI…LS* (límite inferior…límite superior) *Habitación/Habitaciones.*

EJEMPLO PRÁCTICO

Este ejercicio describe la relación de agregación entre la clase Ingrediente y la clase Comida. Sabemos que es una relación de agregación porque se ha utilizado un rombo vacío del lado de una de las clases en este caso la clase Comida, esto nos indica que la clase Comida es el "Todo o contenedora" y la clase Ingrediente "el componente o agregado" y que la clase Ingrediente puede seguir existiendo, aunque se elimine la clase Comida e incluso la clase Ingrediente puede estar asociada con otras clases.

Se podría hacer la lectura siguiente:

"Un ingrediente forma parte de ninguna o más comidas" y *"Una comida está formada o compuesta por un solo ingrediente".*



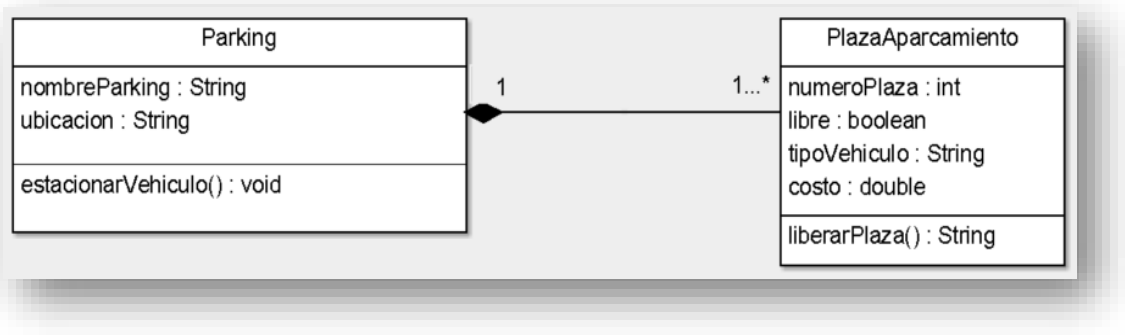
2.4 Relación de composición (es parte elemental de…)

La relación de composición es también una relación de asociación, pero más restrictiva y fuerte que la relación de agregación. Expresa el concepto de *"es parte elemental de"* e indica que las partes agregadas o componentes solo existen para la clase "Todo" y cuando esta última desaparece las partes agregadas también dejan de existir.

La notación empleada es parecida a la utilizada en la agregación con la diferencia que el rombo será completamente negro.

Podemos decir que un componente es **parte esencial** de un elemento.

La relación de *composición* se denota de la siguiente forma:



Relación de composición entre Parking y Plaza de aparcamiento.

- La clase Parking **agrupa o está compuesta por varios elementos** del tipo *PlazaAparcamiento, como mínimo Parking se compone de una PlazaAparcamiento y como máximo no hay límite, es decir Parking se compone de infinitas plazas (en un ejemplo real debería estar acotado el número máximo de plazas en un Parking).*

- *Una PlazaAparcamiento en concreto solo pertenece a un Parking.*

- El tiempo de vida de los objetos de tipo *PlazaAparcamiento* está condicionado por el tiempo de vida del objeto de tipo Parking.

La relación de agregación y composición son muy parecidas, a continuación, se muestra una tabla con las diferencias más importantes:

Característica	Agregación	Composición
Relación de propiedad	Las clases "componentes" pueden existir independientemente y formar parte también de otra clase "Todo"	La clase "Todo" es dueña de la clase "componente"
Ciclo de vida	Las clases "componentes" pueden existir, aunque se elimine la clase "Todo"	Ciclo de vida de la clase "componente" ligado al de la clase "Todo"
Cardinalidad	Puede ser cualquiera. Ej.: (1…1), (1…*), (*…*), etc.	Normalmente uno a muchos (1…*)
Representación gráfica	Línea con un rombo vacío en el extremo cerca de la clase "Todo"	Línea con un rombo relleno de negro en el extremo cerca de la clase "Todo"

En resumen, la diferencia conceptual entre la agregación y la composición es la dependencia y el grado de acoplamiento entre la clase “Todo” y la clase “componente”.

En definitiva, la representación UML de clases nos permite diferenciar entre diferentes tipos de relaciones existentes, aunque a la hora de implementarlas en código (dependiendo del lenguaje empleado) haya veces que esta restricción no se pueda reflejar por lo que el desarrollador deberá decidir, representar, codificar y documentar cada uno de los elementos que formarán parte del desarrollo de un sistema software.

PARA SABER MÁS

A la hora de realizar la planificación de tu proyecto software vas a necesitar incorporar diferentes tipos de diagramas UML. En este enlace puedes obtener información interesante.

VÍDEO DE INTERÉS

En este vídeo encontrarás algunas pautas a seguir a la hora de diseñar un diagrama de clases con notación UML:

EJEMPLO PRÁCTICO

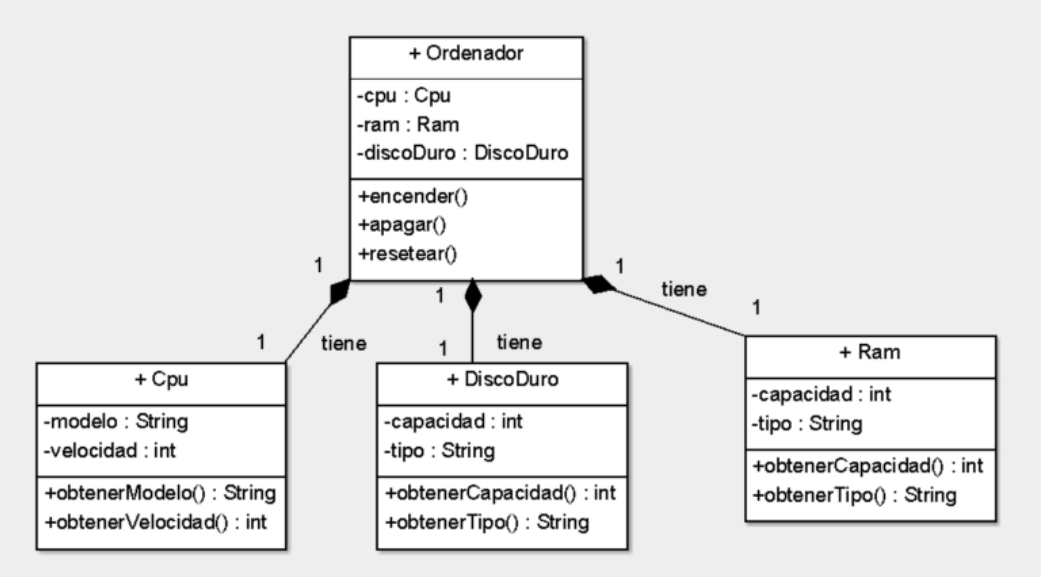
En este caso, la relación de **composición** indica que los componentes (Cpu, Ram y DiscoDuro) son partes esenciales de la clase Ordenador y su existencia está ligada al Ordenador. Ordenador está compuesto por estos componentes, lo que significa que depende de ellos para su funcionamiento. Si Ordenador se destruye, también se destruirán los componentes asociados.

Estas cardinalidades indican cuántas instancias de una entidad están relacionadas con una instancia de la otra entidad en la relación de composición.

Se podría hacer la lectura siguiente:

"Un Ordenador se compone de una sola Cpu, de un solo DiscoDuro y de una sola Ram" y *"Una Cpu, un DiscoDuro y una Ram forma parte de un solo Ordenador".*

Esta relación puede ser diferente y tener otras cardinalidades asociadas (más de una Cpu, de una Ram o de un DiscoDuro) dependerá de los requisitos y directrices del sistema a representar.



3. SOFTWARE DE MODELADO

Para optimizar el proceso de desarrollo y asegurar la calidad del software que se está desarrollando, tu equipo ha decidido utilizar herramientas de modelado UML (Lenguaje de Modelado Unificado) como ArgoUML o Papyrus. De esta forma se mejorará la comunicación entre los miembros del equipo y se facilitará la detección temprana de posibles problemas o errores en el diseño del software.

Para desarrollar los distintos diagramas UML que ayudan a interpretar lo que se va a codificar, se dispone de diversas herramientas software.

De entre ellas cabe destacar algunas que requieren instalación y otras que se pueden utilizar en la nube, como, por ejemplo:

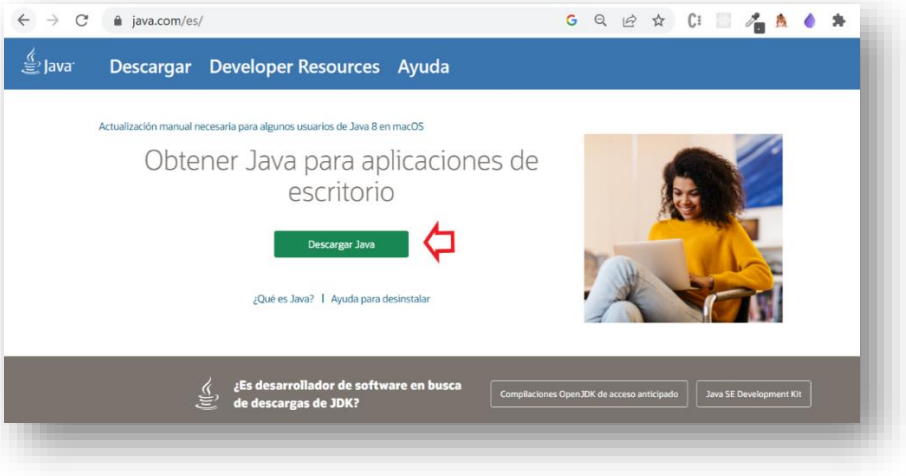
- ArgoUML.
- Papyrus – Plugin para Eclipse.
- Lucid Software.
- Dia.
- Miro, etc.

Instalación ArgoUML

Pero antes de proceder a la instalación de ArgoUML, se necesita el JRE de Java.

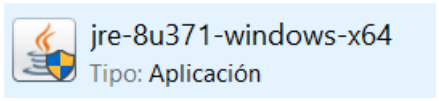
Aunque anteriormente se ha descargado el JDK (que incluye JRE) para poder desarrollar aplicaciones en Java en la unidad anterior, para ArgoUML se necesita tenerlo aparte.

Accederemos a la dirección web https://www.java.com/es/ para descargar el JRE, haciendo clic en el botón verde.



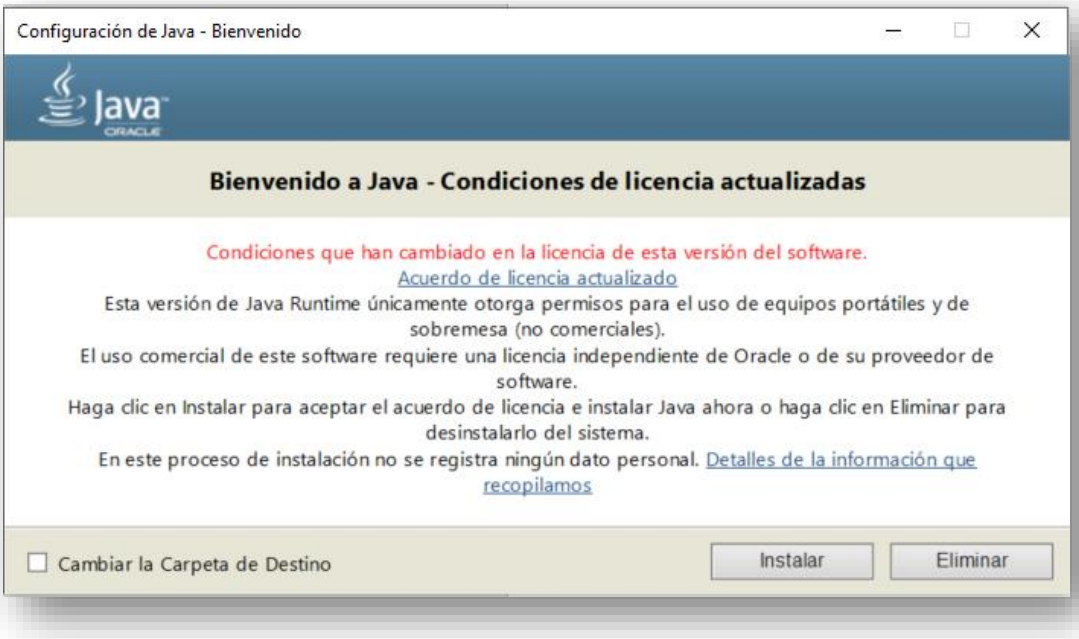
Descarga de JRE.
Fuente: Elaboración propia.

En descargas tendremos el siguiente archivo:



Archivo descargado JRE.
Fuente: Elaboración propia.

Lo ejecutamos con doble clic para instalarlo y seguimos las indicaciones.



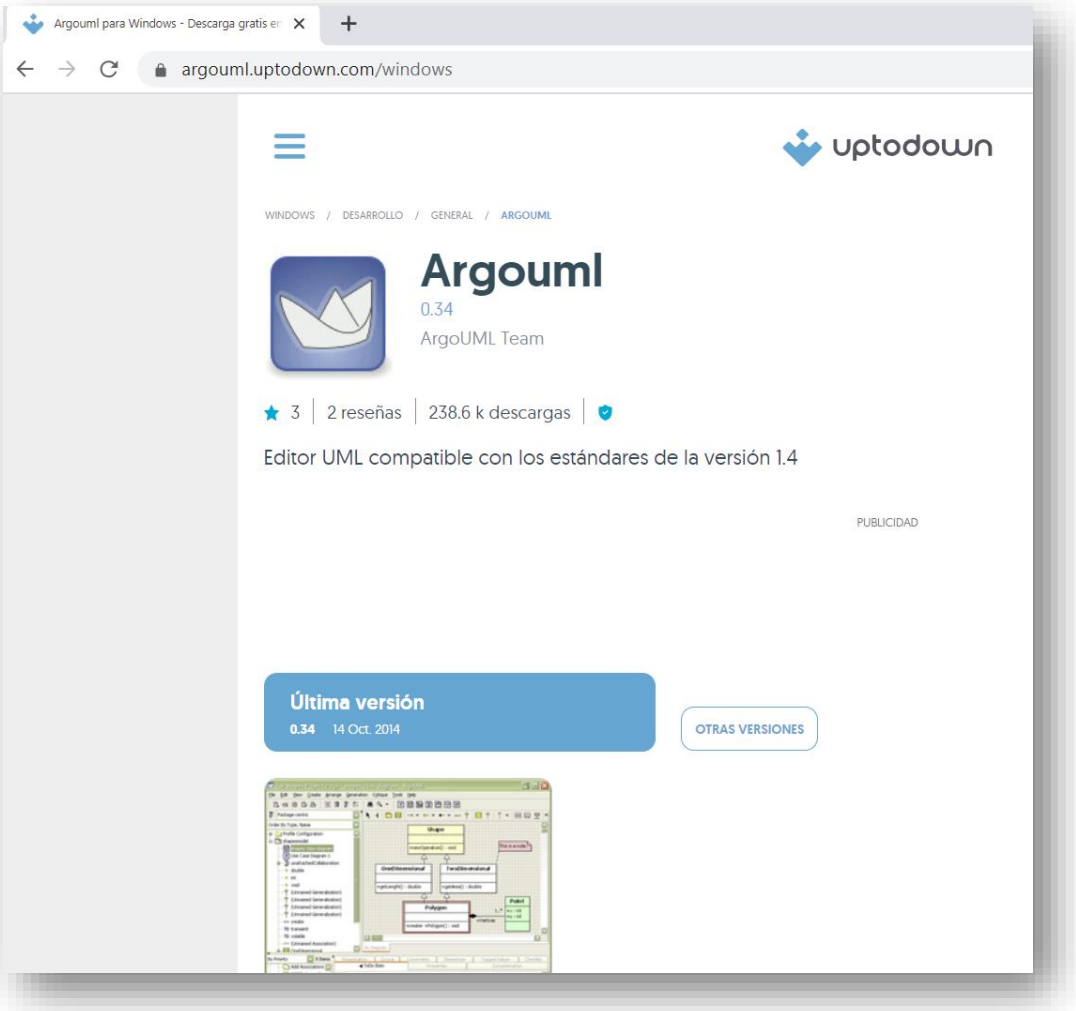
Instalación JRE.
Fuente: Elaboración propia.

Instalación JRE terminada.
Fuente: Elaboración propia.

ENLACE DE INTERÉS

Haz clic al botón verde para descargar el JRE:

A continuación, realizamos la descarga e instalación de ArgoUML.



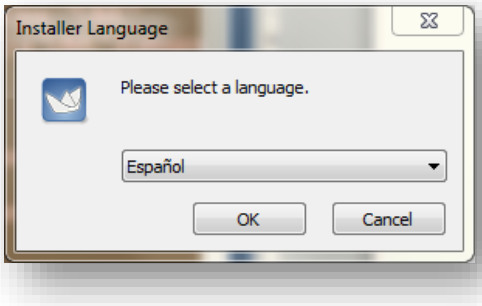
Instalación ArgoUML.
Fuente: Elaboración propia.

Se realizará la descarga de la última versión de ArgoUML disponible y se procederá a instalarla en el sistema. Es una herramienta sencilla de modelado, compatible con cualquier sistema y está disponible en diversos lenguajes.

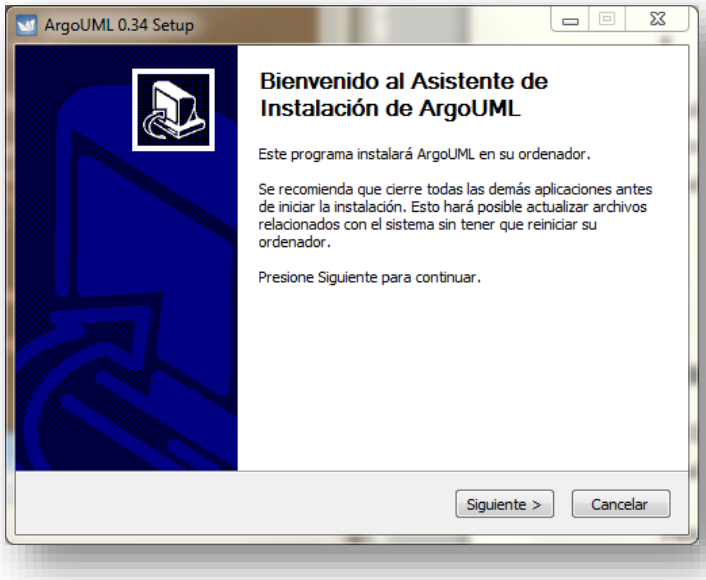
ENLACE DE INTERÉS

Descárgate, ArgoUML, accediendo a esta web:

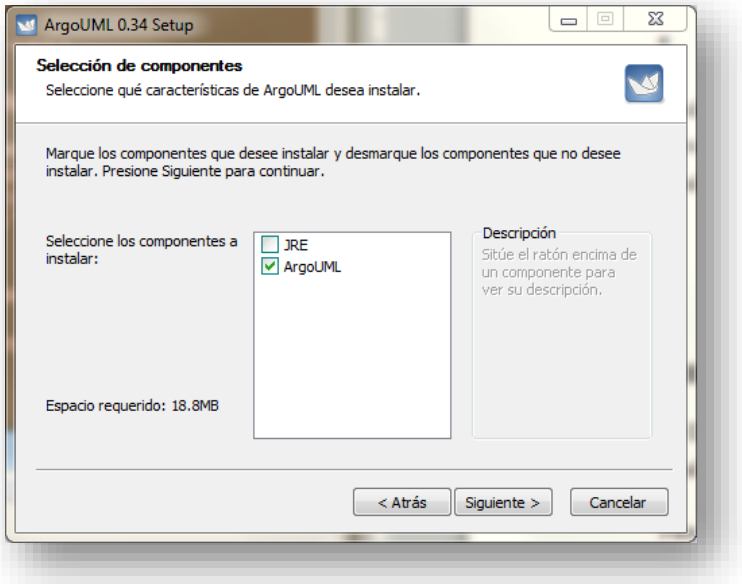
Se seleccionará el lenguaje empleado en el software ArgoUML:



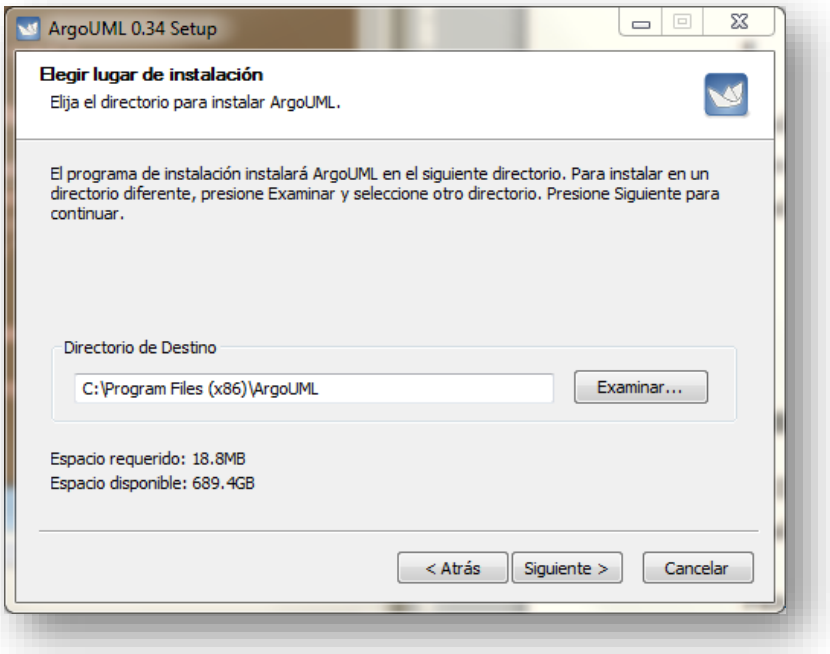
Se siguen los pasos que indica el asistente de instalación de ArgoUML.



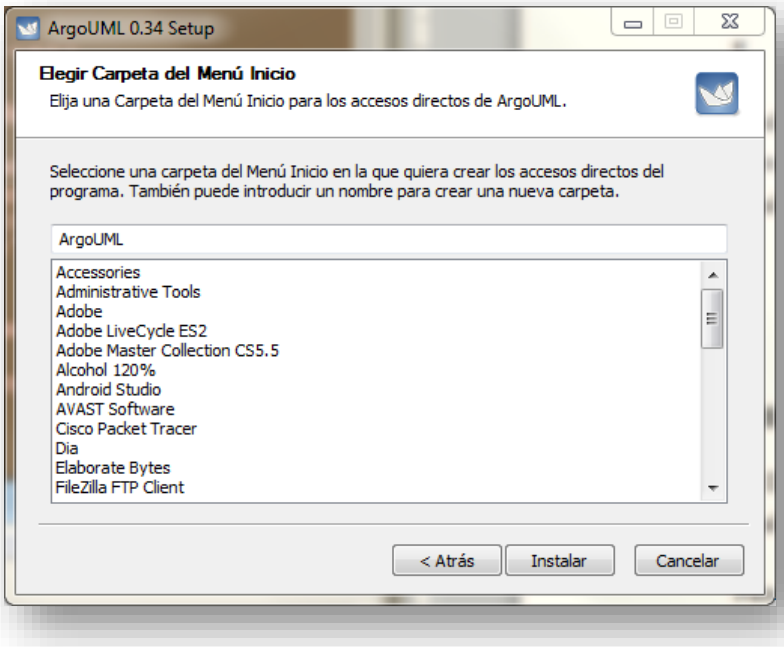
En esta pantalla **no** escogemos la opción JRE, solo opción ArgoUML.



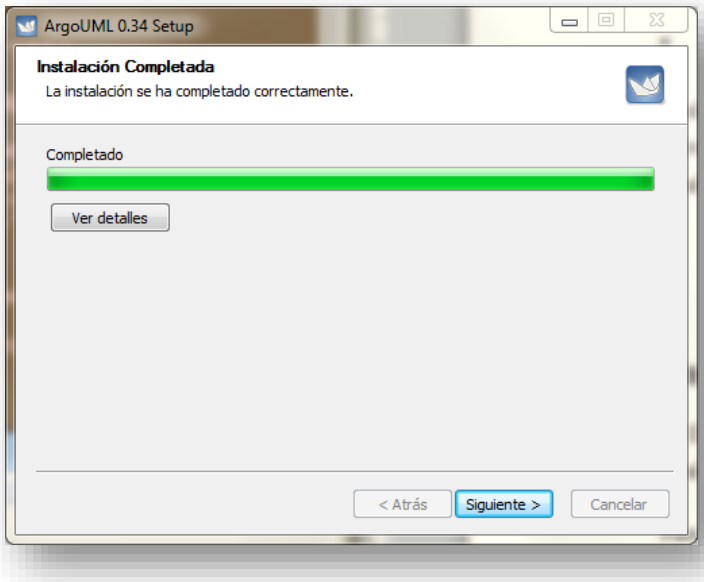
Se indica el directorio en el cual se instalará el software. Se recomienda dejar el que se propone por defecto:



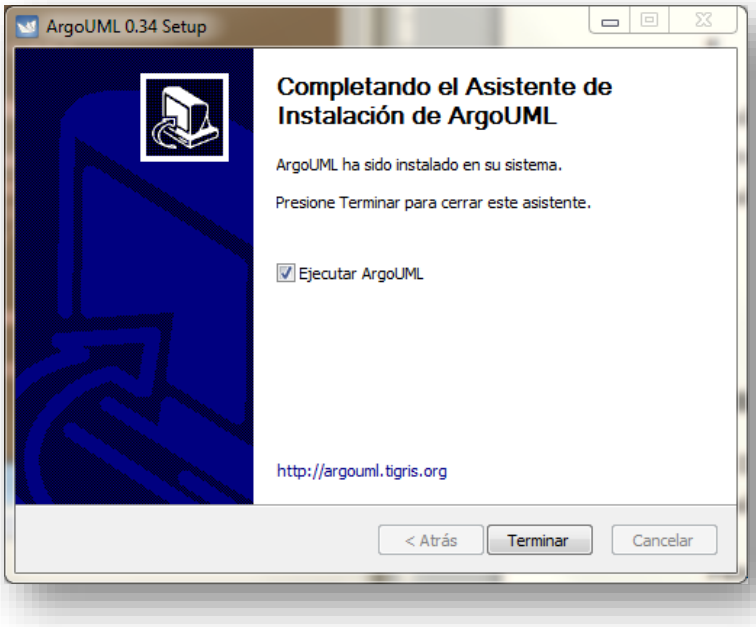
Pulsamos en el botón Siguiente >

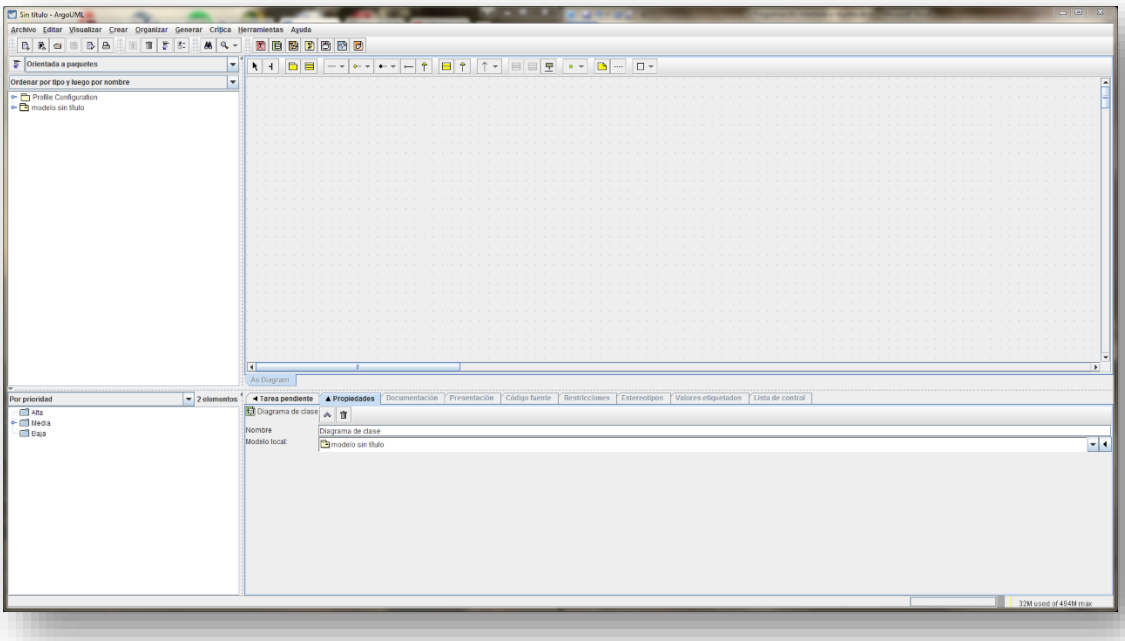


Después se pulsará sobre el botón Instalar.

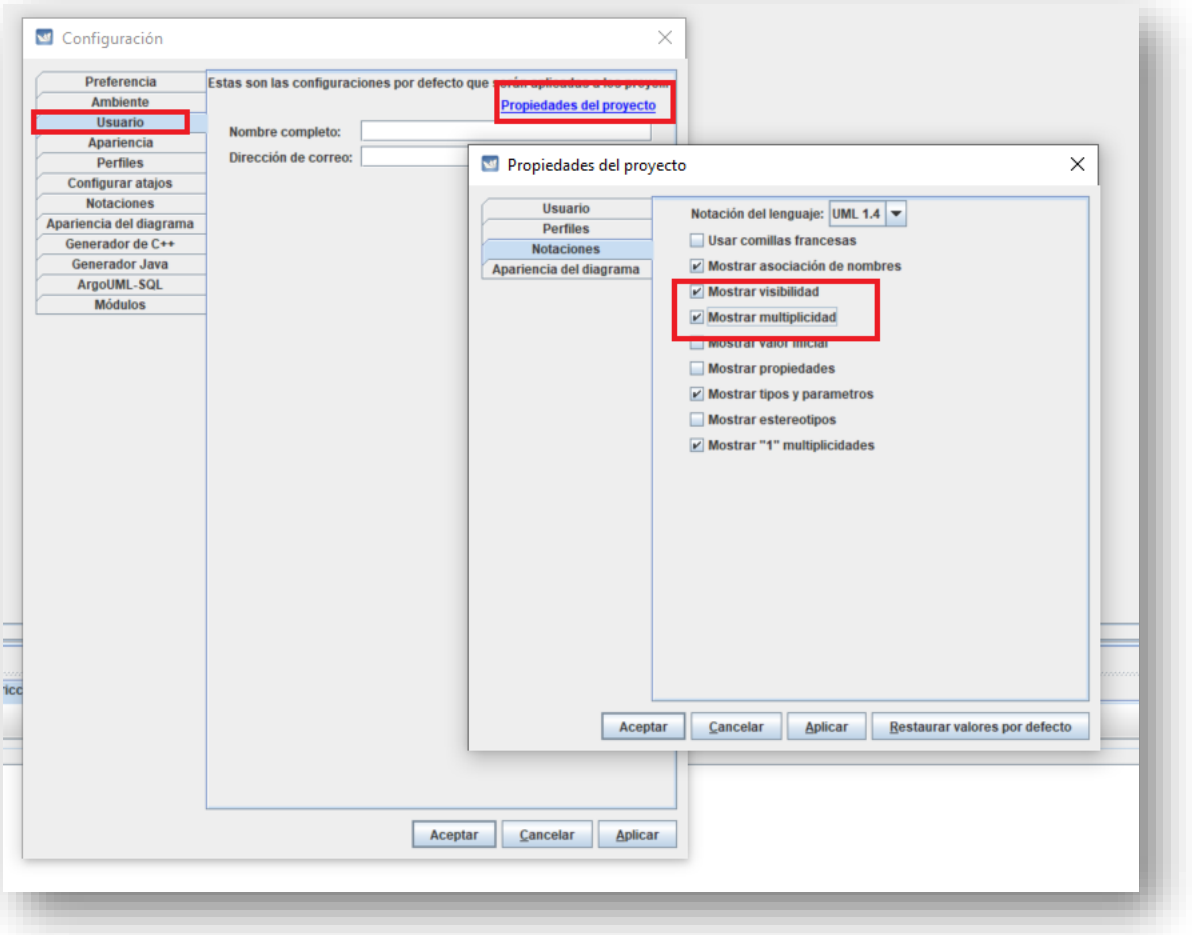


Una vez finalizada la instalación, se pulsa en el botón Terminar y se podrá iniciar el programa y comenzar a realizar los diversos diagramas UML disponibles:





Para que se muestre la visibilidad de los atributos y métodos, es decir, si son públicos (+), privados (-) o protegidos (#), seleccionamos del menú la opción Editar/Configuración como se muestra en la imagen.



Configuración ArgoUML (mostrar visibilidad clases, atributos y métodos).
Fuente: Elaboración propia.

A partir de ahora cuando se indique el tipo de `visibilidad` (que se puede establecer desde la pestaña `Propiedades` que suele aparecer en la parte inferior de la pantalla, según se puede ver en imagen siguiente) se podrá ver un signo +, - o # según se haya indicado, en el lado izquierdo del atributo o método (si la opción “Mostrar visibilidad” no está marcada no aparece).

Configuración ArgoUML (escoger visibilidad a mostrar).
Fuente: Elaboración propia.

PARA SABER MÁS

Conoce más sobre el manejo de ArgoUML:

ENLACE DE INTERÉS

En este enlace conocerás algunas características e información de interés sobre ArgoUML.

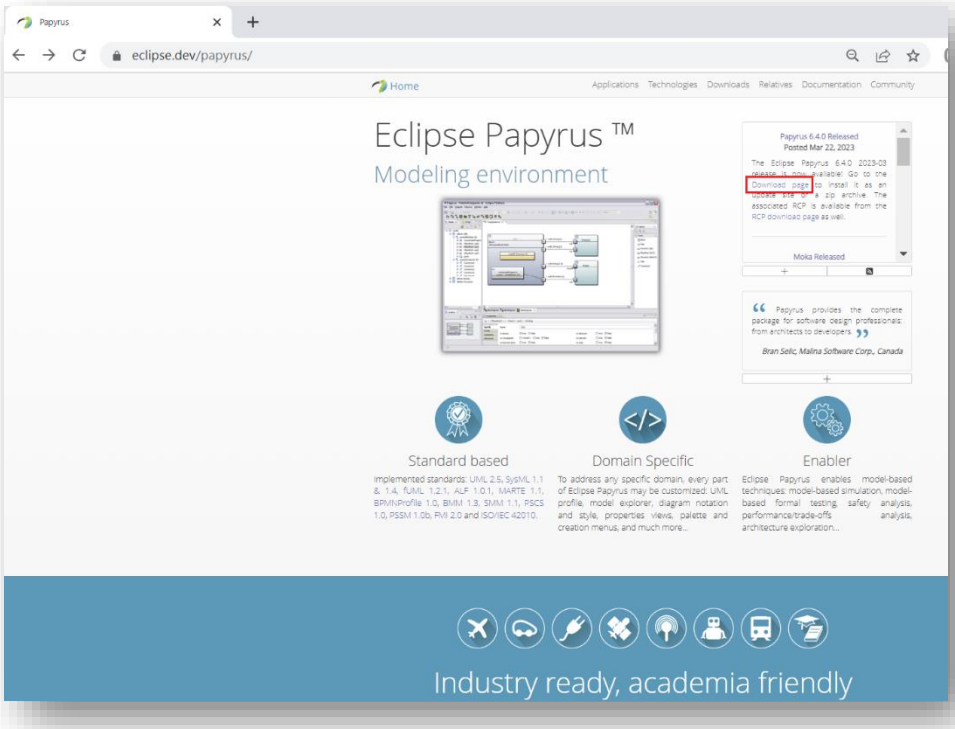
Instalación Plugin Papyrus en Eclipse

El IDE Eclipse permite instalar diversos plugins o módulos. Entre ellos está disponible la instalación del lenguaje de modelado o UML Papyrus.

Para ello se deben seguir los siguientes pasos:

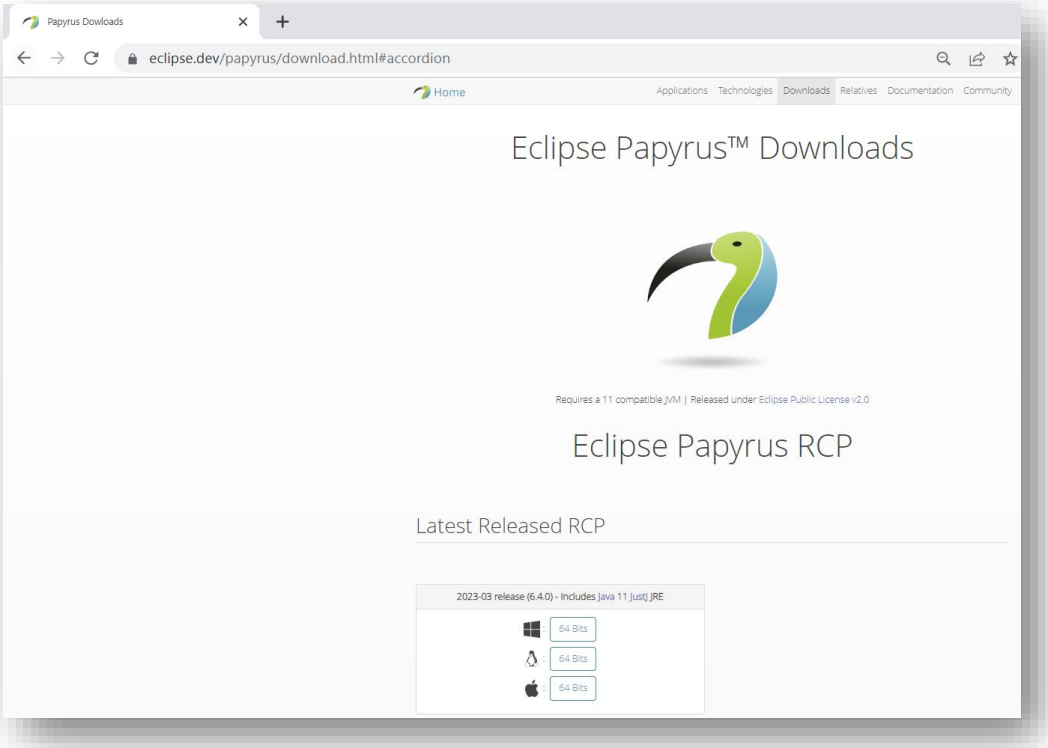
Se abre el IDE Eclipse, y se selecciona la opción `Help -> Install New Software`.
Se tiene que añadir la URL de descarga de Papyrus: https://eclipse.dev/papyrus/

Hacemos clic en el enlace marcado en rojo `Download Page`.

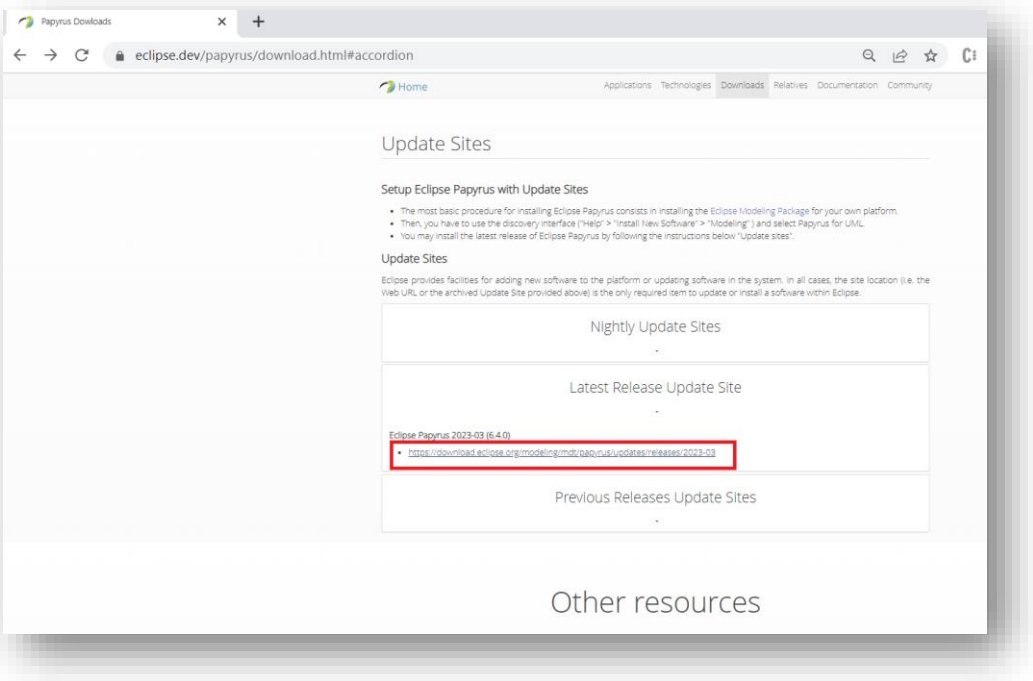


Descarga Papyrus de Eclipse.
Fuente: Elaboración propia.

Nos desplazamos en la página hacia abajo hasta ver el siguiente enlace marcado en rojo.

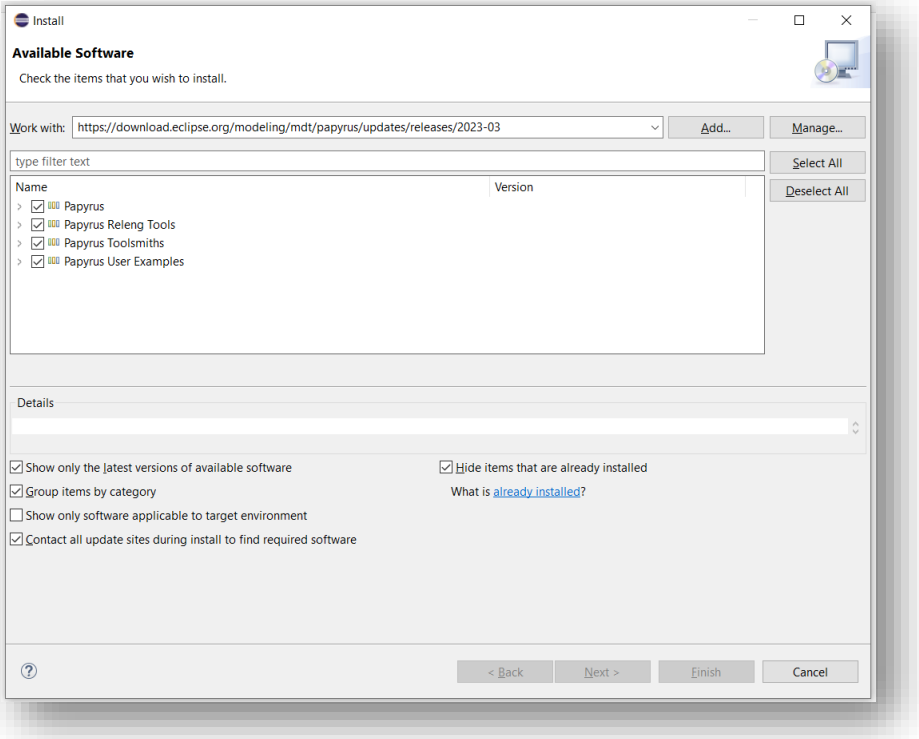


Descarga Papyrus de Eclipse según sistema operativo.
Fuente: Elaboración propia.



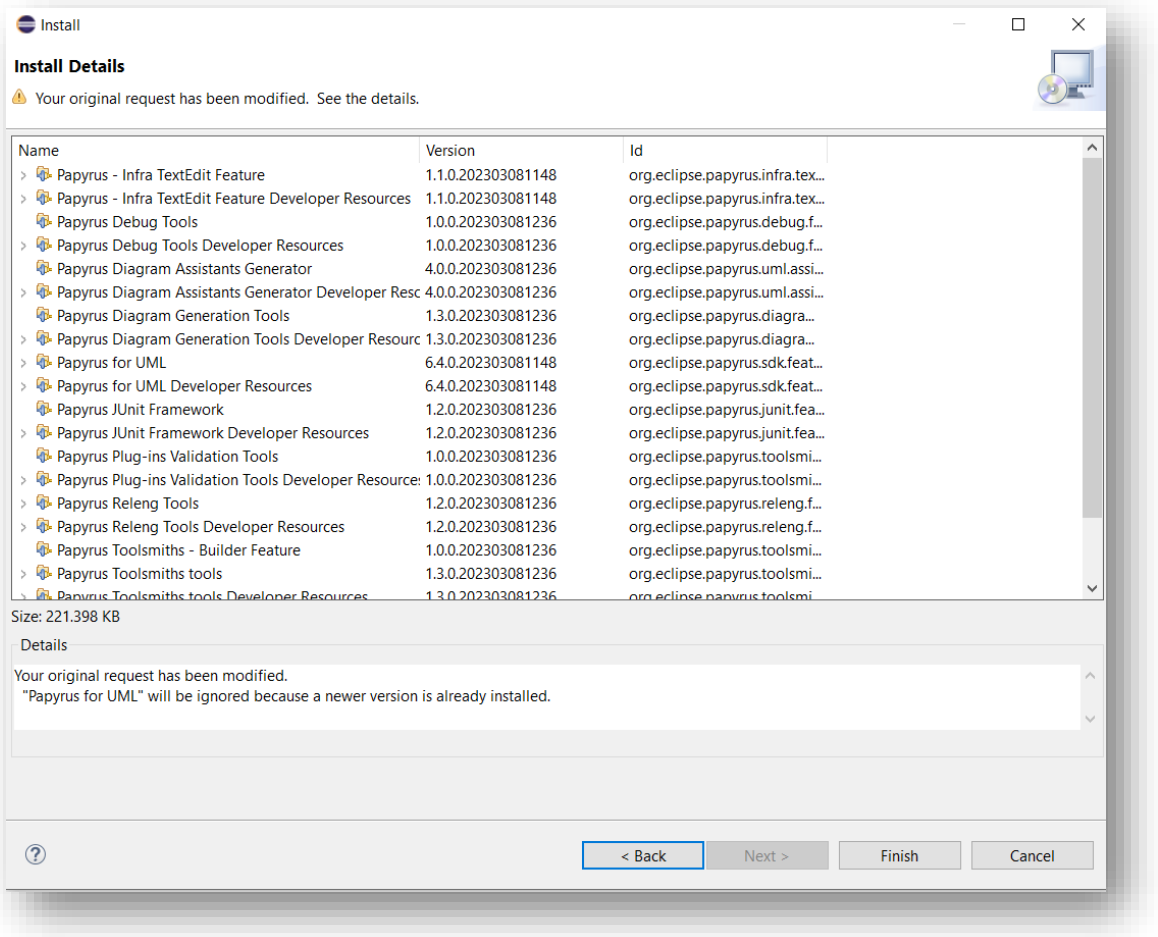
Descarga Papyrus de Eclipse (enlace a copiar).
Fuente: Elaboración propia.

Copiamos este enlace y en Eclipse escogemos la opción `Help` del menú y a continuación `Install` **`new`** `software` y lo pegamos, tal y como se muestra en la siguiente imagen.



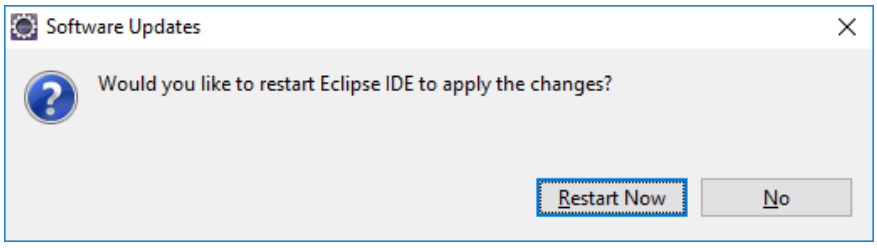
Instalando Papyrus de Eclipse.
Fuente: Elaboración propia.

A continuación, pulsamos `finish` para finalizar la instalación.

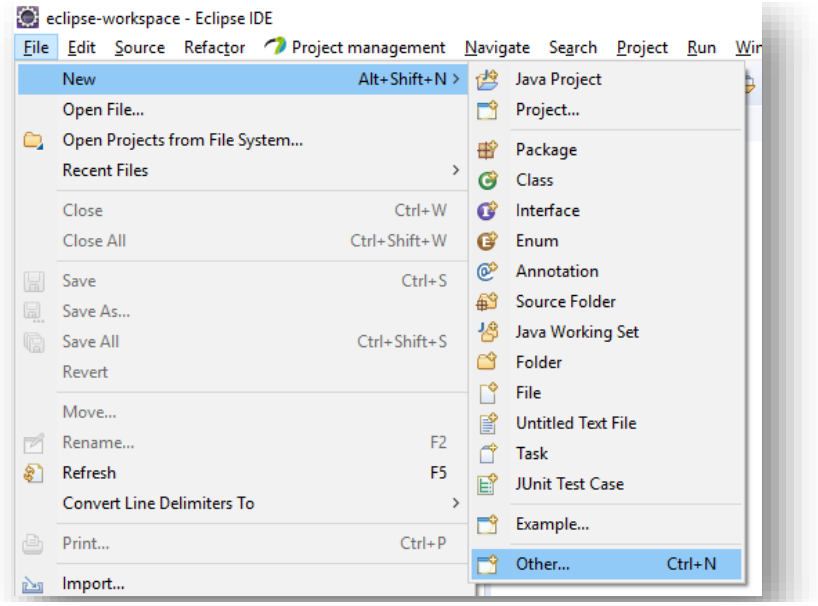


Detalles de instalación- Papyrus de Eclipse.
Fuente: Elaboración propia.

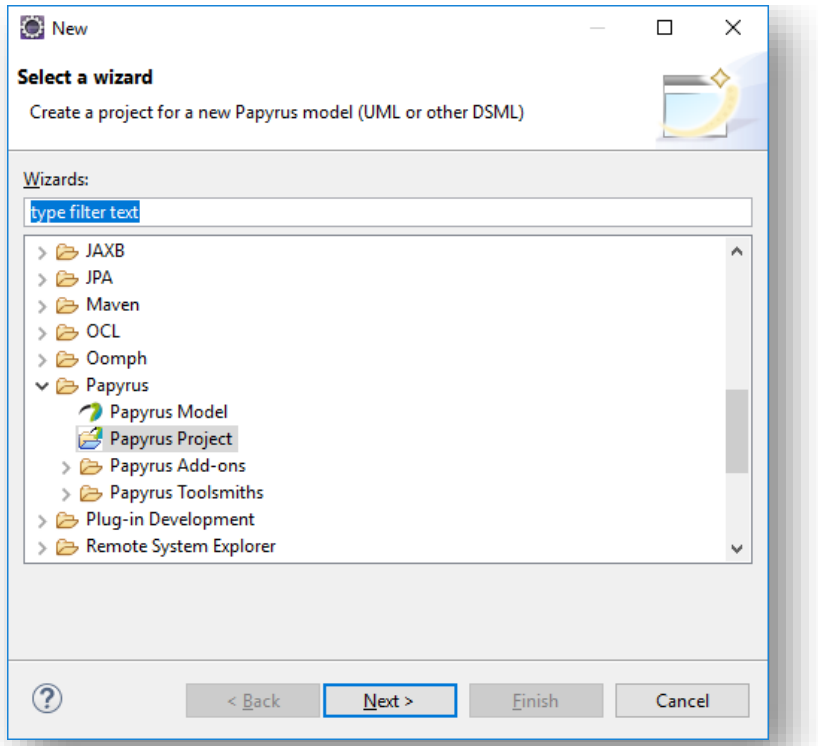
Una vez que se instala hay que pulsar el botón `Restart Now` para finalizar la instalación.

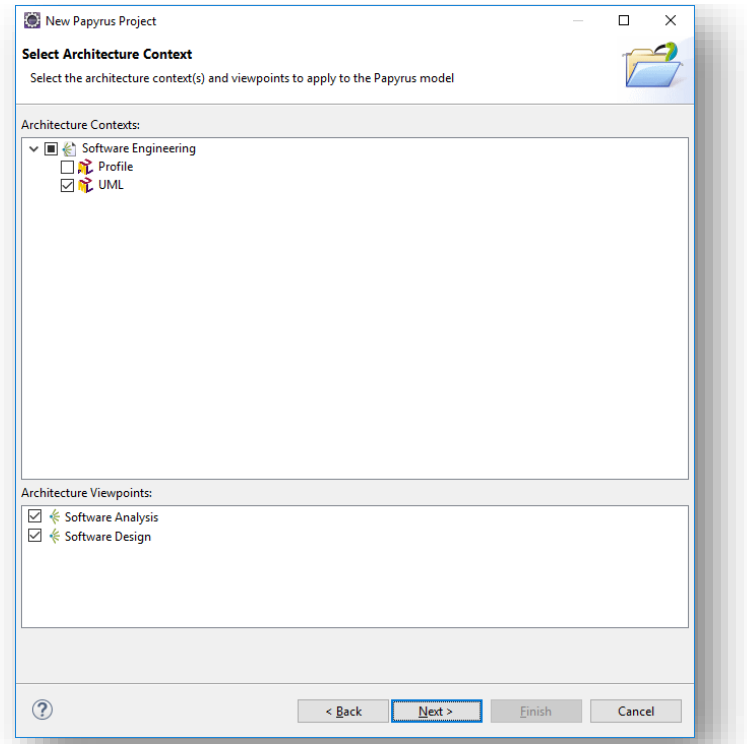


Posteriormente se podrá crear un nuevo proyecto de Papyrus. Se seleccionará `File -> New -> Other…`

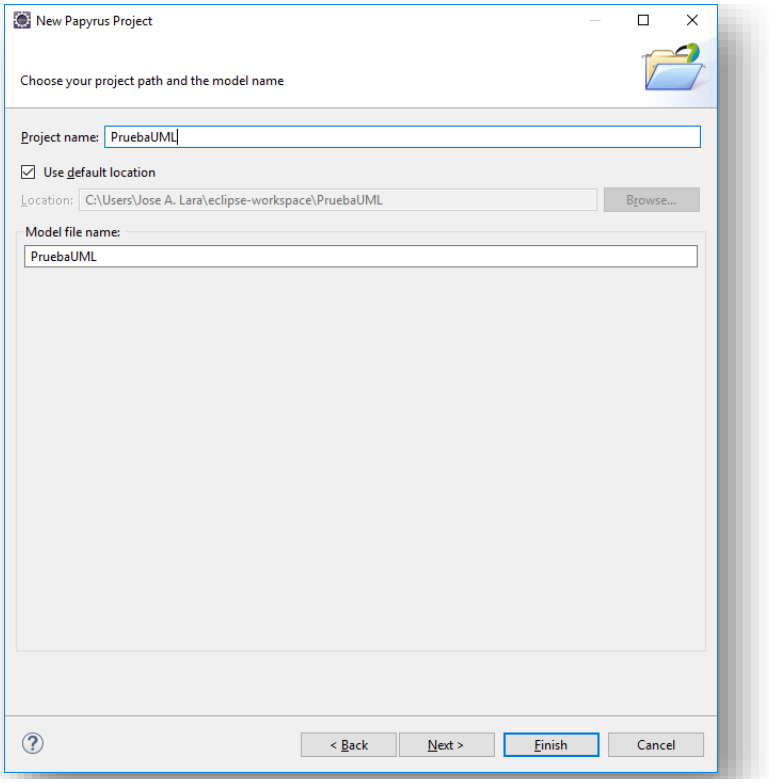


Se selecciona `Papyrus Project y se pulsa Next >`

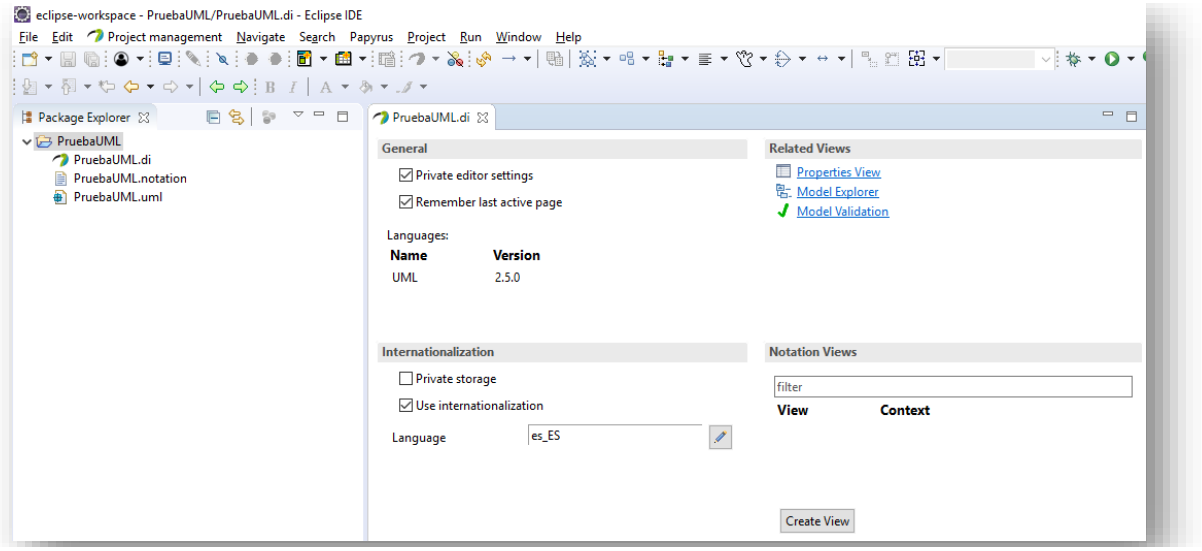




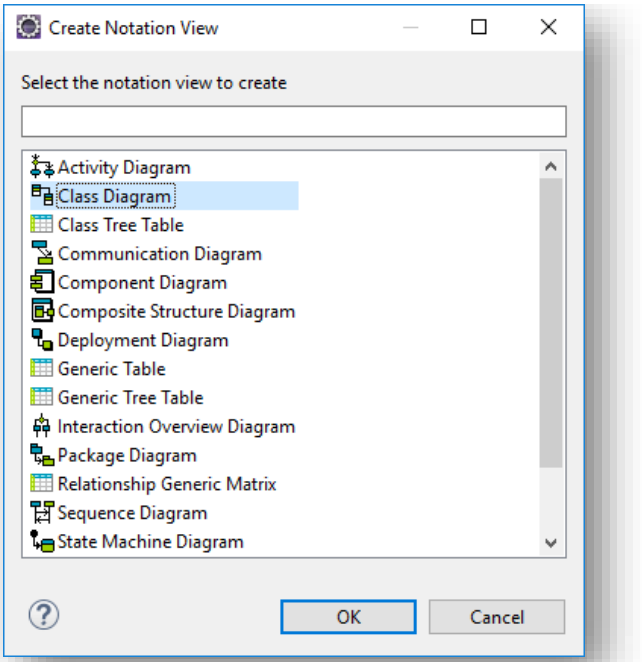
Se selecciona la opción UML y se pulsa `Next >`

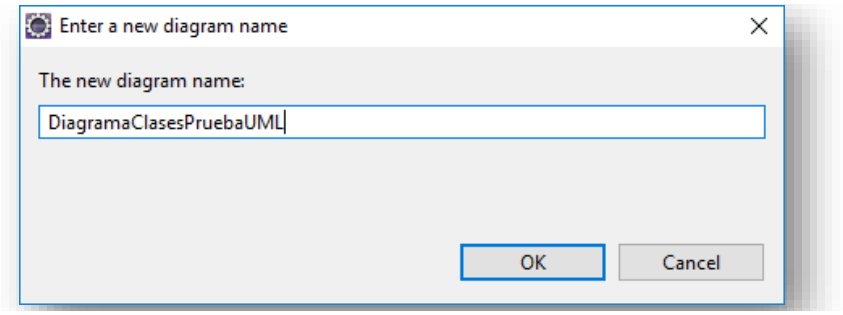


Finalmente se pulsa sobre `Finish`.

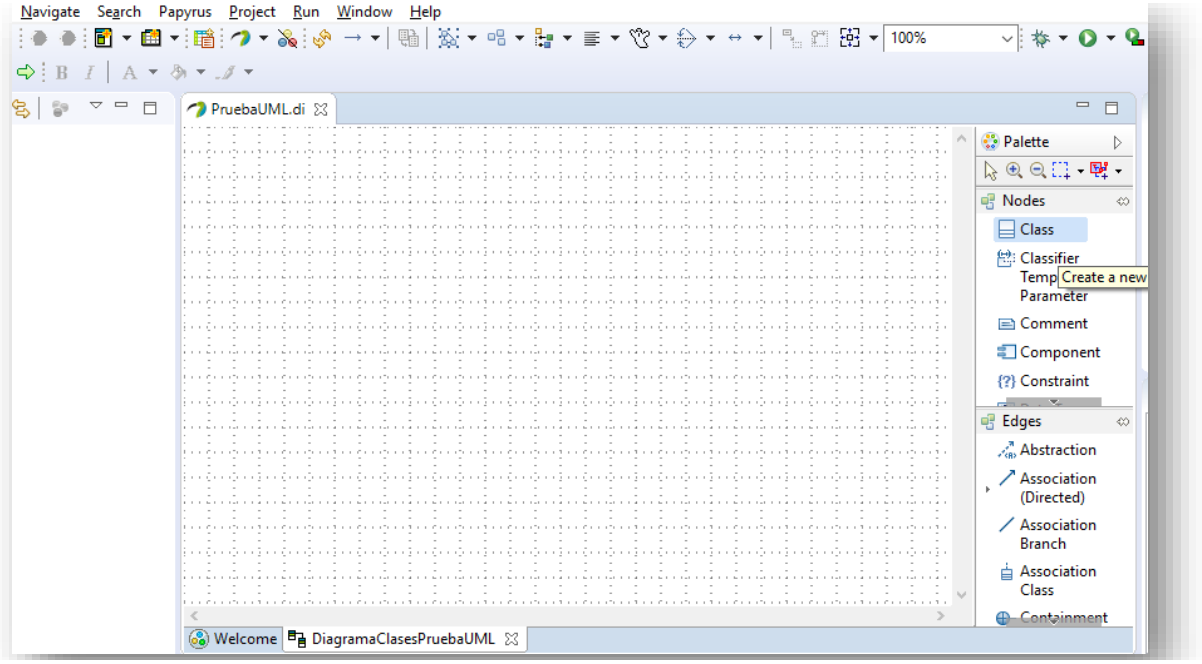


Se pulsa sobre el botón que aparece en la parte de abajo `Create View`, y se selecciona el tipo de diagrama que se quiere crear. Se le da un nombre





Se pulsa el botón OK y ya se puede trabajar en el diagrama.



VÍDEO DE INTERÉS

Accede a este vídeo tutorial sobre como instalar y usar Papyrus respectivamente:

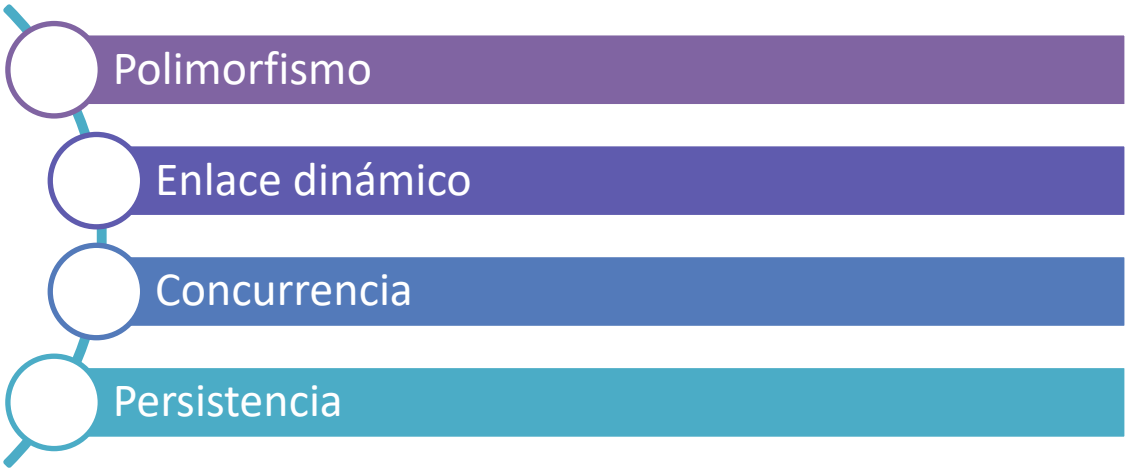
4. FUNDAMENTOS DEL ENFOQUE ORIENTADO A OBJETO

En la 'daily scrum' que tenéis a primera hora, además de justificar cada uno de los componentes, sus logros y los posibles problemas que hayan surgido en la tarea encomendada, se concreta el enfoque del proyecto. De forma unánime se ha decidido que se va a utilizar un enfoque orientado a objetos, basado en principios de herencia, polimorfismo y abstracción que permiten la reutilización de código y la organización jerárquica de clases.

El Enfoque Orientado a Objeto es un enfoque de diseño y modelado que se centra en identificar y definir objetos del mundo real, así como sus relaciones y comportamientos se basa en **cuatro principios** que constituyen la base de todo desarrollo orientado a objetos.

Estos principios son:

Otros elementos que destacar (aunque no fundamentales) en el EOO son:

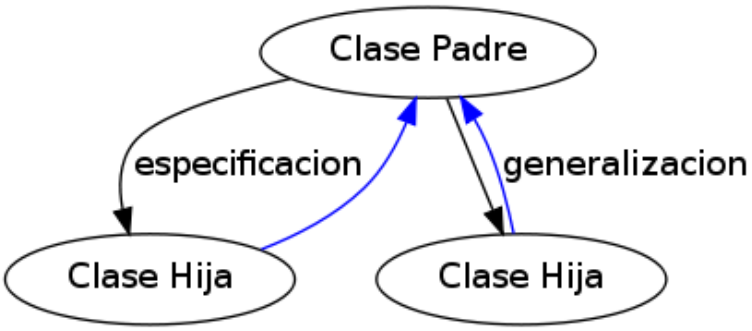


Fundamento 1: Abstracción

Es el principio de ignorar aquellos aspectos de un fenómeno observado que no son relevantes, con el objetivo de concentrarse en aquellos que sí lo son. Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo **distingue** de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

Los mecanismos de abstracción son usados en el EOO para extraer y definir del medio a modelar, sus características y su comportamiento. Dentro del EOO son muy usados los siguientes mecanismos de abstracción: la `Generalización` (y Especialización), la `Agregación` (y Descomposición) y la `Clasificación` (y Ejemplificación).

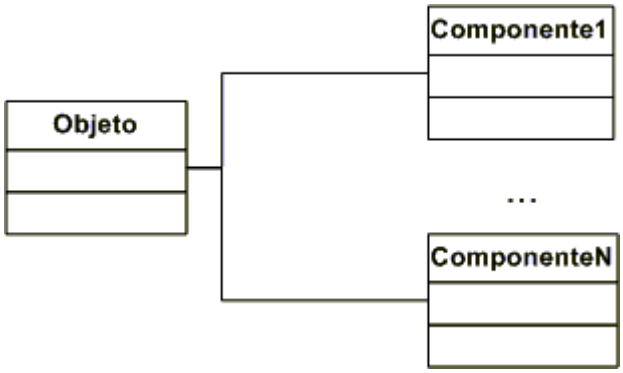
- La **generalización** es el mecanismo de abstracción mediante el cual un conjunto de clases de objetos es agrupado en una clase de nivel superior (**Superclase**), donde las semejanzas de las clases constituyentes (**Subclases**) son enfatizadas, y las diferencias entre ellas son ignoradas. En consecuencia, a través de la generalización, la superclase almacena datos generales de las subclases, y las subclases almacenan sólo datos particulares. La generalización es un proceso de diseño ascendente mientras que **especialización o especificación** que es lo contrario de la generalización es un diseño descendente.



Generalización y especificación.

Ejemplo: La clase Médico es una especialización de la clase Persona, y a su vez, la clase Pediatra es una especialización de la superclase Médico.

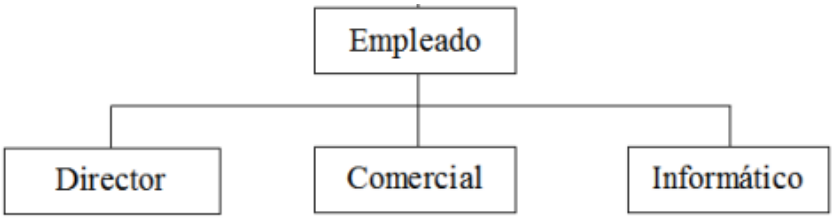
- La **agregación** es el mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). El contrario de agregación es la **descomposición**.



Agregación.

Ejemplo: Mediante agregación se puede definir por ejemplo un computador, que puede descomponerse en: la CPU, la ALU, la memoria y los dispositivos periféricos.

- La **clasificación** consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La **ejemplificación** es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular.



Clasificación.

Fundamento 2: Encapsulamiento (Ocultamiento de Información)

Es la propiedad de la POO que permite **ocultar al mundo exterior** la representación interna del objeto. Esto quiere decir que el objeto puede ser utilizado, pero los datos esenciales del mismo no son conocidos fuera de él.

La idea central del encapsulamiento es **esconder los detalles y mostrar lo relevante**. Permite el ocultamiento de la información separando el aspecto correspondiente a la especificación de la implementación; de esta forma, distingue el "qué hacer" del "cómo hacer". **La especificación es visible al usuario, mientras que la implementación se le oculta.**

El encapsulamiento en un sistema orientado a objeto se representa en cada clase u objeto, definiendo sus atributos y métodos con los siguientes **modos de acceso**:

Público (+) Atributos o métodos que son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella.

Privado (-) Atributos o métodos que solo son accesibles dentro de la implementación de la clase.

Protegido (#): Atributos o Métodos que son accesibles para la propia clase y sus clases hijas (subclases).

Los atributos y los métodos que son públicos constituyen la **interfaz de la clase**, es decir, lo que el mundo exterior conoce de la misma.

Normalmente lo usual es que se oculten los atributos de la clase y solo sean visibles los métodos. Se incluyen entonces algunos métodos de consulta para ver los valores de los atributos.

PARA SABER MÁS

Para saber más sobre conceptos clave de la programación orientada a objetos.

VÍDEO DE INTERÉS

Conoce más datos sobre el encapsulamiento en la programación orientada a objetos en este video.

Fundamento 3: Modularidad

Es la propiedad que permite tener **independencia** entre las diferentes partes de un sistema. La modularidad consiste en dividir un programa en **módulos** o partes, que pueden ser compilados separadamente, pero que tienen conexiones con otros módulos. En un mismo módulo se suele colocar clases y objetos que guarden una estrecha relación. El sentido de modularidad está muy relacionado con el ocultamiento de información.

Fundamento 4: Herencia

Es el proceso mediante el cual un objeto de una clase **adquiere propiedades definidas** en otra clase que lo precede en una jerarquía de clasificaciones. Permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos (Programación Diferencial), **evitando repetición de código y permitiendo la reusabilidad**.

Las clases **heredan** los datos y métodos de la superclase. Un método heredado puede ser sustituido por uno propio si ambos tienen el mismo nombre.

La herencia puede ser **simple** (cada clase tiene sólo una superclase) o **múltiple** (cada clase puede tener asociada varias superclases).

Ejemplo: La clase Docente y la clase Estudiante heredan las propiedades de la clase Persona (superclase, herencia simple). La clase Preparador (subclase) hereda propiedades de la clase Docente y de la clase Estudiante (herencia múltiple).

Fundamento 5: Polimorfismo

Es una propiedad de la POO que permite que un método tenga **múltiples implementaciones**, que se seleccionan teniendo en cuenta el tipo objeto indicado al solicitar la ejecución del método.

El polimorfismo operacional o **Sobrecarga operacional** permite aplicar operaciones con igual nombre a diferentes clases o están relacionados en términos de inclusión. En este tipo de polimorfismo, los métodos son interpretados en el contexto del objeto particular, ya que los métodos con nombres comunes son implementados de diferente manera dependiendo de cada clase.

Ejemplo: el área de un cuadrado, rectángulo y círculo, son calculados de manera distinta; sin embargo, en sus clases respectivas puede existir la implementación del área bajo el nombre común `Área`. En la práctica y dependiendo del objeto que llame al método, se usará el código correspondiente.

Ejemplos: `Superclase: Clase Animal Subclases: Clases Mamífero, Ave, Pez`

Se puede definir un método `Comer` en cada subclase, cuya implementación cambia de acuerdo con la clase invocada, sin embargo, el nombre del método es el mismo.

Mamifero.Comer ≠ Ave.Comer ≠ Pez.Comer

Es decir, el método comer ha sido adaptado o modificado en cada clase hija. Por tanto, el método `Comer` para la clase `Animal`, clase `Ave`, clase `Mamífero` y clase `Pez` tienen el mismo nombre, pero tendrán una funcionalidad distinta.

Ejemplo El operador +. Este operador tiene dos funciones diferentes de acuerdo con el tipo de dato de los operandos a los que se aplica. Si los dos elementos son numéricos, el operador + significa suma algebraica de los mismos, en cambio, si al menos uno de los operandos es un `String` o Carácter, el operador realiza la concatenación de cadenas de caracteres.

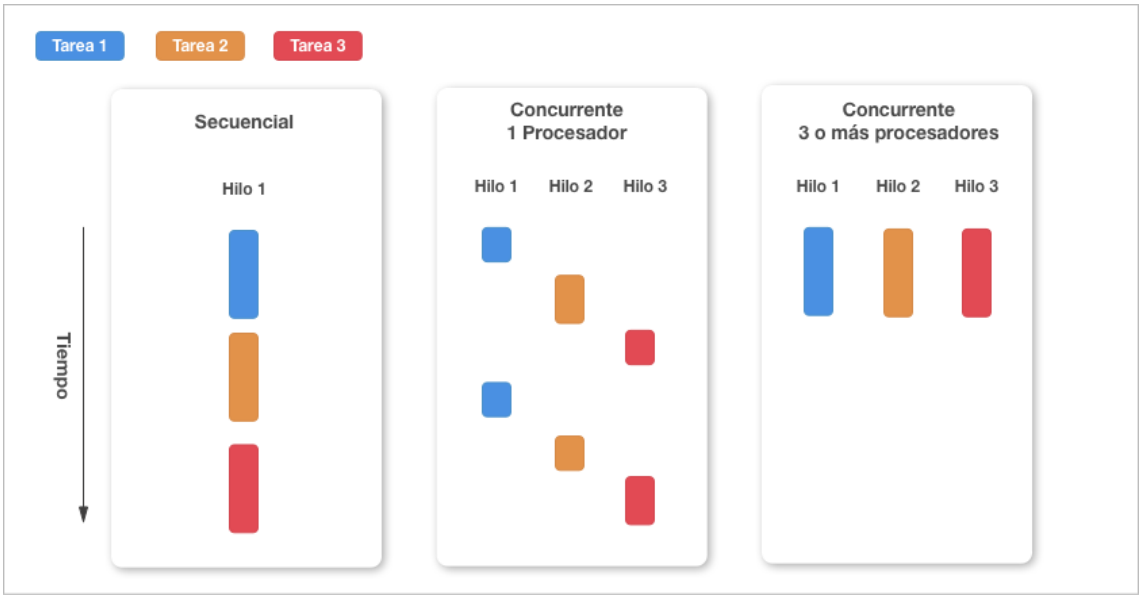
Fundamento 6: Enlace dinámico

El enlace dinámico es un mecanismo utilizado en lenguajes de programación orientados a objetos, como Java, que permite que la invocación de un método se resuelva en tiempo de ejecución en lugar de tiempo de compilación. Esto significa que la decisión sobre qué método se debe llamar se realiza en función del tipo de objeto con el que se está trabajando en ese momento. Esto permite lograr flexibilidad y polimorfismo en la programación, ya que un mismo método puede comportarse de manera diferente según el tipo específico de objeto que lo invoca.

Fundamento 7: Concurrencia

La concurrencia en el enfoque orientado a objetos se refiere a la capacidad de un sistema para ejecutar múltiples tareas o procesos de manera simultánea y cooperativa. En un sistema concurrente, varios objetos pueden trabajar de forma independiente o colaborar entre sí para lograr un objetivo común. Sin embargo, la concurrencia también puede presentar desafíos, como la coordinación y sincronización de los objetos para evitar problemas como condiciones de carrera (ejecución de diferentes hilos pueden interferir entre sí) y bloqueos (producen un acceso exclusivo a recursos compartidos y evitan que otros lo accedan o modifiquen hasta que se complete la tarea).

La gestión adecuada de estas situaciones es esencial para garantizar la consistencia y la integridad de los datos en aplicaciones concurrentes.



Diferencia entre ejecución secuencial, concurrente y paralela.
Fuente: https://blog.makeitreal.camp/concurrencia/

Fundamento 8: Persistencia

La persistencia en el enfoque orientado a objetos hace referencia a la capacidad de los objetos de mantener su estado más allá de la duración de un programa en ejecución. Es decir, la capacidad de guardar y recuperar objetos y sus datos desde un medio de almacenamiento permanente, como una base de datos o un archivo, para que puedan ser utilizados nuevamente en futuras sesiones del programa. La persistencia permite que los datos y objetos mantengan su estado entre ejecuciones y contribuye a la durabilidad y recuperación de información en aplicaciones de larga duración. La persistencia en Java se puede lograr de las siguientes formas y la elección de la técnica adecuada depende del tipo de aplicación y los requisitos de almacenamiento y recuperación de datos:

Persistencia en Base de Datos

- Java Database Connectivity (JDBC): Puedes utilizar JDBC para interactuar con bases de datos relacionales y guardar objetos en tablas de la base de datos. Por ejemplo, puedes crear una tabla "Usuarios" y guardar objetos Usuario con sus atributos como nombre, edad, etc.

- Framework de mapeo objeto-relacional (ORM) como Hibernate: Hibernate facilita el mapeo entre objetos Java y tablas de bases de datos, permitiéndote guardar y recuperar objetos sin escribir SQL directamente.

Persistencia en Archivos

- Serialización: En Java, puedes hacer que tus objetos sean serializables implementando la interfaz Serializable. Luego, puedes guardar los objetos en archivos utilizando la serialización y recuperarlos posteriormente.

Persistencia en Memoria Caché:

- Bibliotecas de caché en memoria como Ehcache o Hazelcast: Estas bibliotecas permiten mantener objetos en memoria caché, lo que mejora el rendimiento de acceso a datos y evita llamadas frecuentes a la base de datos.

Persistencia en Sistemas Distribuidos

- Tecnologías de mensajería como Apache Kafka o RabbitMQ: Estas tecnologías permiten enviar mensajes y eventos entre diferentes aplicaciones o microservicios, lo que puede ser útil para mantener la persistencia de datos en sistemas distribuidos.

RECUERDA

Estos son solo ejemplos simples y los atributos y métodos pueden variar según las características y requisitos específicos del sistema informático que estés representando. En unidades siguientes se verá como establecer relaciones con cardinalidades superiores a 1 del lado de la parte componente. (utilizando arrays).

ENLACE DE INTERÉS

Conoce más ejemplos de Diagrama de clases UML:

PARA SABER MÁS

Para saber más sobre la agregación, composición y generalización, accede a este enlace:

RESUMEN FINAL

En la Programación Orientada a Objetos los programas son representados por un conjunto de objetos que interactúan entre ellos. Un objeto engloba tanto datos como operaciones sobre estos datos.

La Programación Orientada a Objetos constituye una buena opción a la hora de resolver un problema, sobre todo cuando éste es muy extenso. En este enfoque de programación, se facilita evitar la repetición de código, no sólo a través de la creación de clases que hereden propiedades y métodos de otras, sino además a través de que el código es reutilizable por sistemas posteriores que tengan alguna similitud con los ya creados.

En esta unidad se ha comenzado analizando la programación orientada a objetos para posteriormente estudiar las unidades de modularidad de la POO: clases y objetos, así como las relaciones entre ellas y su representación en notación UML. A continuación, se han identificado los fundamentos básicos de este enfoque y uno de los principales conceptos del paradigma: la herencia.

Para representar las clases y sus relaciones podemos hacer uso de diagramas de clases UML, para modelar, atributos y métodos en un diagrama de clases.

Existe gran cantidad de software de modelado utilizado para crear y visualizar diagramas UML, lo que facilita la representación gráfica de la estructura y relaciones entre clases. Entre ellos cabe destacar ArgoUML y Papyrus que se utilizan para crear y editar diagramas UML.

La herencia es un concepto fundamental en la programación orientada a objetos que permite crear nuevas clases (llamadas subclases o clases derivadas) basadas en clases ya existentes (llamadas superclases o clases base). La subclase hereda los atributos y métodos de la superclase y puede agregar nuevos atributos y métodos propios o modificar los existentes. Esto facilita la reutilización de código y permite crear jerarquías de clases que representan relaciones de tipo "es un".

Además de la herencia existen otras relaciones importantes, como son, la asociación, la agregación y la composición. Estas relaciones son cruciales para modelar y diseñar sistemas complejos en la programación orientada a objetos, ya que permiten representar las interacciones y conexiones entre clases y objetos de manera efectiva. El uso adecuado de la herencia y otras relaciones de clases puede conducir a un diseño de software más claro, eficiente y fácil de mantener.