UNIDAD 3: PROGRAMACIÓN ORIENTADA A OBJETOS

Módulo Profesional: Programación

Índice

RESUMEN INTRODUCTORIO

A lo largo de esta unidad veremos la definición de clase y objeto y la diferencia que existe entre estos dos conceptos. Posteriormente trataremos el concepto de atributo y método, para centrarnos en una clase de métodos especiales: los métodos constructores. Veremos también cómo se comunican los objetos entre sí: mensajes, cómo se representa una clase en notación algorítmica y los tipos de relaciones que pueden existir entre clases. Para cada uno de los apartados veremos ejemplos en el lenguaje de programación 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 la unidad el alumnado:

- Será capaz de reconocer las unidades que forman parte de la POO.
- Identificará los principales atributos y métodos que representan a una clase.
- Conocerá los distintos tipos de relaciones existentes entre los objetos.
- Será capaz de realizar representaciones del mundo real siguiendo el enfoque de la orientación a objetos.

1. PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

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

Una clase es una plantilla que define qué características tiene y cómo se comporta una determinada entidad. Los objetos son entidades concretas que presentan un determinado estado, tienen comportamientos y se diferencian de otros objetos por sus atributos.

¿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.

2. 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.

La clase puede definirse como la agrupación o colección de objetos que comparten una estructura común y un comportamiento común.

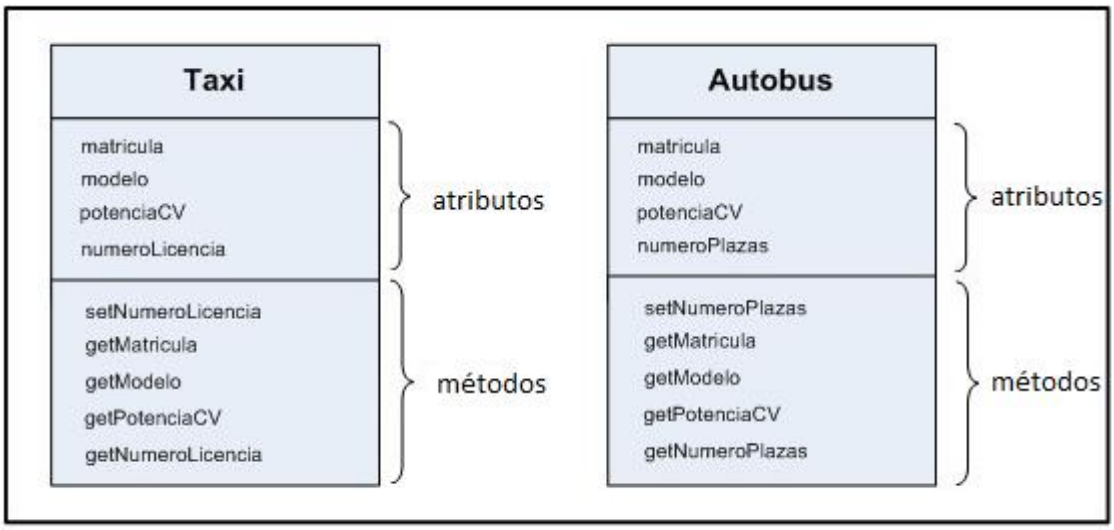


Ilustración: ejemplo clases (POO)

Podemos decir que una clase es una plantilla que contiene la descripción general de una colección de objetos. Una clase consta de atributos y métodos que resumen las características y el comportamiento comunes de un conjunto de objetos.

Todo objeto (también llamado instancia de una clase), pertenece a alguna 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.

Su sintaxis algorítmica es:

```
Clase <Nombre de la Clase>
            …
FClase <Nombre de la Clase>;
```

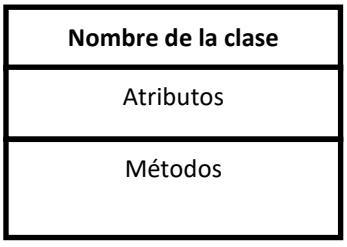
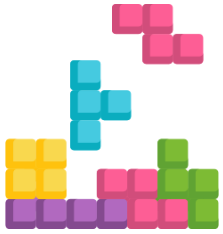


Ilustración: representación de una clase

Características Generales

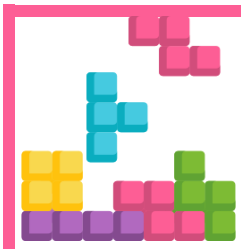
- **Una clase es un nivel de abstracción alto.** La clase permite describir un conjunto de características comunes para los objetos que representa.



EJEMPLO PRÁCTICO

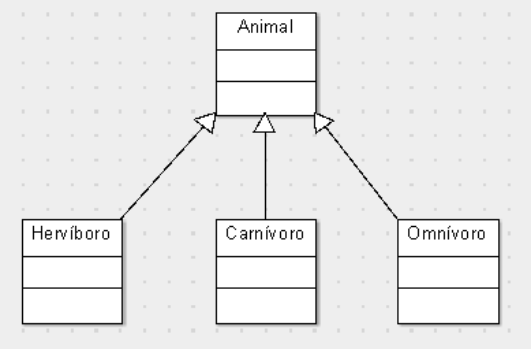
La clase Avión se puede utilizar para definir los atributos (tipo de avión, distancia, altura, velocidad de crucero, capacidad, país de origen, etc.) y los métodos (calcular posición en el vuelo, calcular velocidad de vuelo, estimar tiempo de llegada, despegar, aterrizar, volar, etc.) de los objetos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

EJEMPLO PRÁCTICO

Ejemplo de clase `Vehiculo` en Java:

```
public class Vehiculo {
	private String matricula;
	private String modelo;
	private int potenciaCV;

	public Vehiculo(String matricula, String modelo, int pCV) {
		this.matricula=matricula;
		this.modelo=modelo;
		this.potenciaCV=pCV;
	}

	public String getMatricula() {
		return matricula;
	}

	public String getModelo() {
		return modelo;
	}

}
```

3. OBJETO

Es una entidad (tangible o intangible) que posee **características** y **acciones** que realiza por sí solo o interactuando con otros objetos.

Un **objeto** es una entidad caracterizada por sus atributos propios y cuyo comportamiento está determinado por las acciones o funciones que pueden modificarlo, así como también las acciones que requiere de otros objetos.

Un **objeto** tiene identidad e "inteligencia" y constituye una unidad que oculta tanto datos como la descripción de su manipulación. Puede ser definido como una **encapsulación** y una **abstracción**: una encapsulación de atributos y servicios, y una abstracción del mundo real.

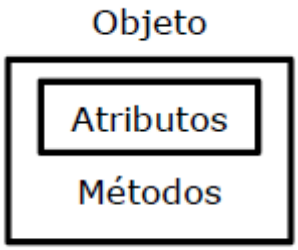
Para el contexto del Enfoque Orientado a Objetos (EOO) un objeto es una entidad que encapsula datos (atributos) y acciones o funciones que los

manejan (métodos). También para el EOO un objeto se define como una **instancia o particularización de una clase**.

Los objetos de interés durante el desarrollo de software no sólo son tomados de la vida real (objetos visibles o tangibles), también pueden ser abstractos. En general son entidades que juegan un rol bien definido en el dominio del problema. Un libro, una persona, un carro, un polígono son apenas algunos ejemplos de objeto.

Cada objeto puede ser considerado como un proveedor de servicios utilizados por otros objetos que son sus clientes. Cada objeto puede ser a la vez proveedor y cliente. De allí que un programa pueda ser visto como un conjunto de relaciones entre proveedores clientes. Los servicios ofrecidos por los objetos son de dos tipos:

1. Los datos, que llamamos **atributos**.
2. Las acciones o funciones, que llamamos **métodos**.



Características Generales:

- **Un objeto se identifica por un nombre o un identificador único que lo diferencia de los demás**. Ejemplo: el objeto Cuenta de Ahorros número 12345 es diferente al objeto Cuenta de Ahorros número 25789. En este caso el identificador que los hace únicos es el número de la cuenta.

- **Un objeto posee estados**. El estado de un objeto está determinado por los valores que poseen sus atributos en un momento dado.

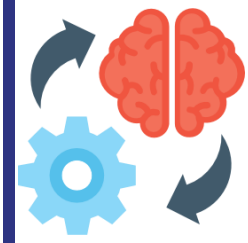
- **Un objeto tiene un conjunto de métodos**. El comportamiento general de los objetos dentro de un sistema se describe o representa mediante sus operaciones o métodos. Los métodos se utilizarán para obtener o cambiar el estado de los objetos, así como para proporcionar un medio de comunicación entre objetos.

- **Un objeto tiene un conjunto de atributos**. Los atributos de un objeto contienen valores que determinan el estado del objeto durante su tiempo de vida. Se implementan con variables, constantes y estructuras de datos (similares a los campos de un registro).

- **Los objetos soportan encapsulamiento**. La estructura interna de un objeto normalmente está oculta a los usuarios de este. Los datos del objeto están disponibles solo para ser manipulados por los propios métodos del objeto. El único mecanismo que lo conecta con el mundo exterior es el paso de mensajes.

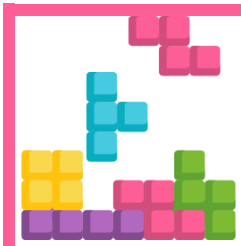
- **Un objeto tiene un tiempo de vida dentro del programa o sistema que lo crea y utiliza**. Para ser utilizado en un algoritmo el objeto debe ser creado con una instrucción particular (New o Nuevo) y al finalizar su utilización es destruido con el uso de otra instrucción o de manera automática.

- **Un objeto es una instancia de una clase**. Cada objeto concreto dentro de un sistema es miembro de una clase específica y tiene el conjunto de atributos y métodos especificados en la misma.



RECUERDA

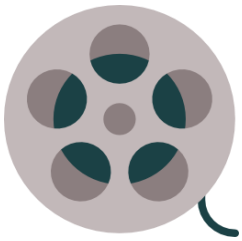
Un objeto es una entidad (tangible o intangible) que posee características y acciones que realiza por sí solo o interactuando con otros objetos. Consta de atributos y métodos.



EJEMPLO PRÁCTICO

Ejemplo de creación (instanciación) de objetos en Java

```
Vehiculo vehiculo1 = new Vehiculo("2345BCD","Porsche 911", 540);
Vehiculo vehiculo2 = new Vehiculo("8765FCD","Fiat Panda",69);
System.out.println("Matrícula "+vehiculo1.getModelo()+" - "
                                          +vehiculo1.getMatricula());
System.out.println("Matrícula "+vehiculo2.getModelo()+" - "
                                          +vehiculo2.getMatricula());
```



VIDEO DE INTERÉS

En el siguiente enlace se explican las Clases, Objetos y Métodos en Java:

https://www.youtube.com/watch?v=AEXLtATMkZM&

4. 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**.

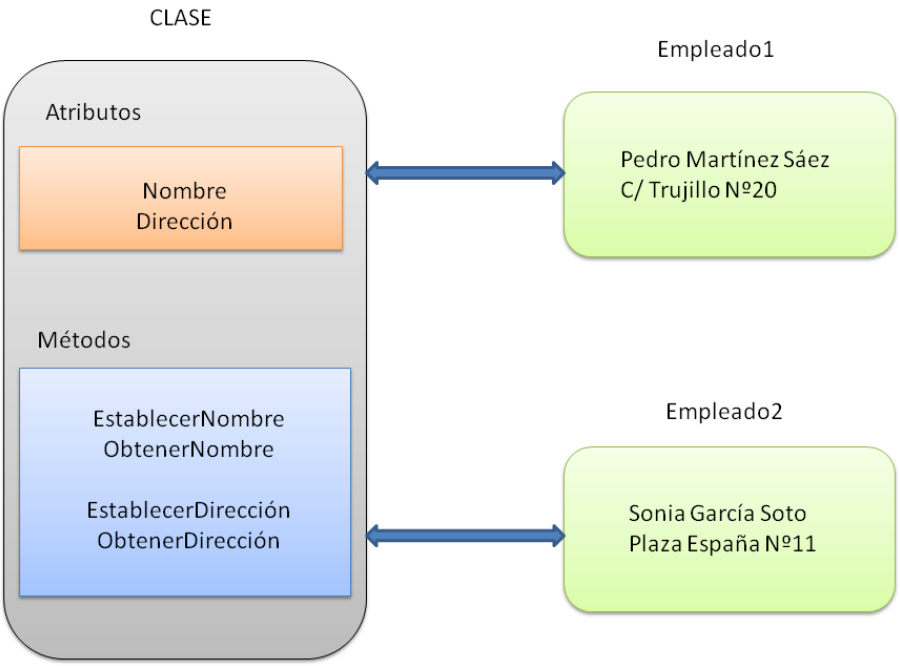


Ilustración: relación entre clase y objeto

ENLACE DE INTERÉS

Puede ampliar información sobre la programación orientada a objetos visitando la web:

http://www.desarrolloweb.com/articulos/499.php

ENLACE DE INTERÉS

Puede ampliar información sobre el concepto de Objetos y Clases en Java

http://www.aprenderaprogramar.es/index.php?option=com_content&view=article&id=411:conceptos-de-objetos-y-clases-en-java-definicion-de-instancia-ejemplos-basicos-y-practicos-cu00619b&catid=68:curso-aprender-programacion-java-desde-cero&Itemid=188

5. ATRIBUTO

Son los datos o variables que caracterizan al objeto y cuyos valores en un momento dado indican su estado.

Un atributo es una característica de un objeto. Mediante los atributos se define información oculta dentro de un objeto, que es manipulada solamente por los métodos definidos sobre dicho objeto. 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 **estructurado** (array, registro, archivo, lista, etc.)

Su sintaxis algorítmica es*:*

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

Los **modos de acceso** son:

Público*:* Atributos (o Métodos) que 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*:* Atributos (o Métodos) que sólo son accesibles dentro de la implementación de la clase. **También** se puede representar con el símbolo **–.**

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

6. 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.

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.

La **sintaxis algorítmica** de los métodos expresados como funciones y acciones es:

Para **funciones** se pueden usar cualquiera de estas dos sintaxis:

<Modo de Acceso> Función <Nombre> [(Lista Parámetros)]: <Descripción del Tipo de datos>

Para **acciones**:

<Modo de Acceso> Acción <Nombre> [(Lista Parámetros)] donde los parámetros son opcionales

Ejemplo: Un rectángulo es un objeto caracterizado 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 y un algoritmo con el desarrollo de este.

- En particular se destaca el método **constructor**, que no es más que el método que se ejecuta cuando el objeto es creado. Este constructor suele tener el mismo nombre de la clase/objeto, pero, aunque es una práctica común, el método constructor no necesariamente tiene que llamarse igual a la clase (al menos, no en pseudocódigo). Es un método que recibe cero o más parámetros y lo usual es que inicialicen los valores de los atributos del objeto.

- En lenguajes como Java, C# y C++ se puede definir más de un método constructor, que normalmente se diferencian entre sí por la cantidad de parámetros que reciben.

- 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.

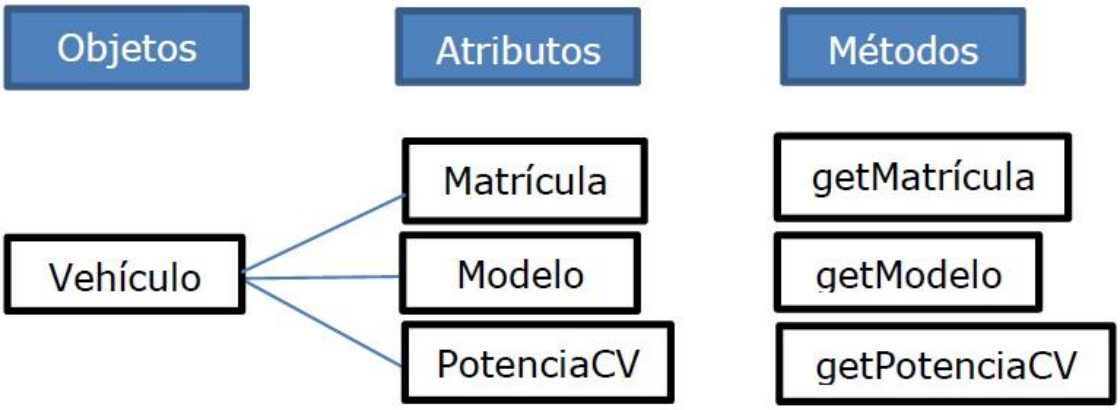


Ilustración: ejemplo de atributos y métodos

ARTÍCULO DE INTERÉS

Se recomienda realizar la siguiente lectura sobre Simula, el primer lenguaje orientado a objetos:

http://es.wikipedia.org/wiki/Simula

7. CREACIÓN DE OBJETOS Y MÉTODOS CONSTRUCTORES

Cada objeto o instancia de una clase debe ser creada explícitamente a través de un método u operación especial denominado **Constructor**. Los atributos de un objeto toman valores iniciales dados por el constructor. Por convención el método constructor tiene el mismo nombre de la clase (aunque esto no es obligatorio en pseudocódigo) y no se le asocia un modo de acceso (**es público**). En Java sí se le asocia un modo de acceso, aunque normalmente es público.

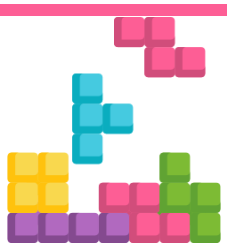
Algunos lenguajes proveen un **método constructor por defecto** para cada clase y/o permiten la definición de más de un método constructor.

Métodos destructores de objetos:

Los objetos que ya no son utilizados en un programa ocupan inútilmente espacio de memoria, el cual es conveniente recuperar en un momento

dado. Según el lenguaje de programación utilizado esta tarea es dada al programador o es tratada automáticamente por el procesador o soporte de ejecución del lenguaje. En Java esta tarea la realiza el recolector de basura (Garbage Collector) de forma automática.

En la **notación algorítmica** NO se tendrá en cuenta ese problema de administración de memoria, por lo tanto, no se definirán formas para destruir objetos. En cambio, al utilizar lenguajes de programación sí que se deben conocer los métodos destructores suministrados por el lenguaje y utilizarlos a fin de eliminar objetos una vez que no sean útiles.



EJEMPLO PRÁCTICO

Ejemplo de constructor de la clase `Vehiculo` en Java:

```
public class Vehiculo {
private String matricula;
private String modelo;
private int potenciaCV;

public Vehiculo(String matricula, String modelo, int pCV) {
	this.matricula=matricula;
	this.modelo=modelo;
	this.potenciaCV=pCV;
}}
```

8. MENSAJE

Es la petición de un objeto a otro para solicitar la ejecución de alguno de sus métodos o para obtener el valor de un atributo público.

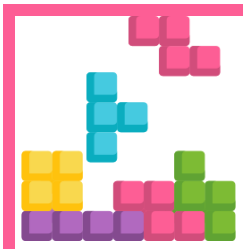
Estructuralmente, un mensaje consta de 3 partes:

- **Identidad del receptor***:* Nombre del objeto que contiene el método a ejecutar.
- **Nombre del método a ejecutar**: Solo los métodos declarados públicos.
- **Lista de Parámetros** que recibe el método (cero o más parámetros)

Su sintaxis algorítmica es*:*

```
<Variable_Objeto>.<Nombre_Método> ( [<Lista de Parámetros> ] );
```

Cuando el objeto receptor recibe el mensaje, comienza la ejecución del algoritmo contenido dentro del método invocado, recibiendo y/o devolviendo los valores de los parámetros correspondientes, si los tiene, ya que son opcionales: (**[ ]**).



EJEMPLO PRÁCTICO

Definición de la Clase Rectángulo: IMPLEMENTACIÓN

Rectángulo
Privado Real Largo Privado Real Ancho
Constructor Acción Rectángulo(lar, anc) Público Función Área: Real Público Función Perímetro: Real

```
Clase Rectángulo;
	// Atributos
	Privado:
		Real Largo, Ancho;
	// Métodos
	// método constructor
	Acción Rectángulo(Real lar, anc);
		Largo = lar;
		Ancho = anc;
	FAcción;

	Público Función Área: Real
	// Retorna el área o superficie ocupada por el rectángulo
		retornar(Largo * Ancho);
	FFunción Área;

	Público Función Perímetro: Real
	// Retorna el perímetro del rectángulo
		retornar(2 * (Largo + Ancho));
	FFunción Perímetro;
FClase Rectángulo;

// Uso de la clase rectángulo
Acción Principal
	Rectángulo R;	// se declara una variable de tipo objeto
			// Rectángulo, a la cual llamaremos R.
	Real L, A;	// se declaran las variables reales L y A
			// para largo y ancho del rectángulo.
	Escribir("Suministre a continuación los valores para el largo
y el ancho");
	Leer(L); Leer(A);
	R.Rectángulo(L, A);
	Escribir("Resultados de los cálculos:");
	Escribir("Área: "+ R.Área + " - Perímetro " + R.Perímetro);
FAcción Principal;
```

9. REPRESENTACIÓN EN NOTACIÓN ALGORÍTMICA DE UNA CLASE

Las clases son el elemento principal dentro del enfoque orientado a objetos. En este lenguaje las declaraciones forman parte de su propio grupo, el grupo [Clases]. Cada una de las declaraciones de clase debe tener el siguiente formato:

```
Clase <Identificador> [Hereda de: <Clases>]
        Atributos
        Público:
                [Constantes]; [Variables]; o [Estructuras];
        Privado:
                [Constantes]; [Variables]; o [Estructuras];
        Protegido:
                [Constantes]; [Variables]; o [Estructuras];
        Operaciones
        Público:
                [Métodos]
        Privado:
                [Métodos]
        Protegido:
                [Métodos]
FClase <Identificador>
```

En el caso de la especificación de los Atributos o de los Métodos los modos de acceso Público (+), Privado (-) o Protegido (#) pueden omitirse, solamente en el caso en el que los grupos [Constantes], [Estructuras] y [Variables] son vacíos, es decir, no se estén declarando Atributos o Métodos.

¿SABÍAS QUE…?

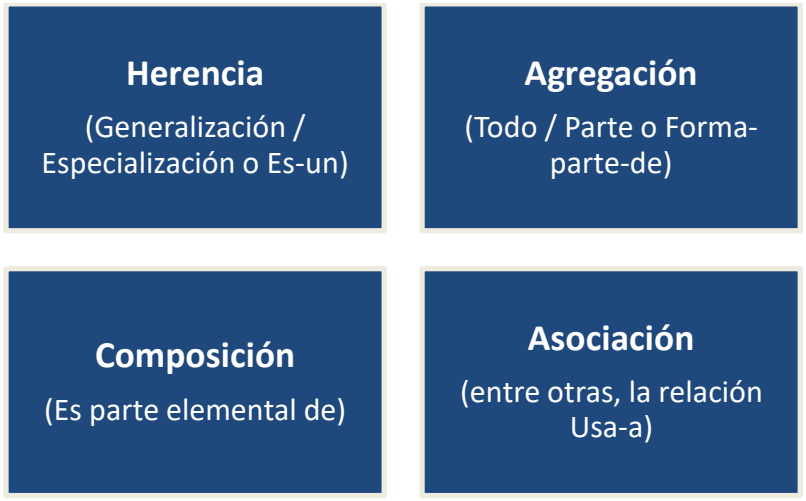
Otra forma de especificar los modos de acceso es colocándolo antes del nombre DE CADA UNO de los atributos o métodos.

10. RELACIONES ENTRE CLASES

Las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan **relacionar entre sí**, de manera que puedan compartir 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:



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

Es un tipo de jerarquía de clases, en la que 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 incorporar atributos y métodos propios, que a su vez serán heredados por sus hijas.

Representación:

- **En la notación algorítmica**: Se coloca el nombre de la clase padre después de la frase Hereda de del encabezado de la clase y se usan sus atributos y métodos públicos o protegidos. Ejemplo: Clase Punto3D Hereda de Punto2D.

- **En el diagrama de clases**: La herencia se representa mediante una relación de generalización/especificación, que se denota de la siguiente forma:

clase **hija** → clase **padre subclase** → **superclase**

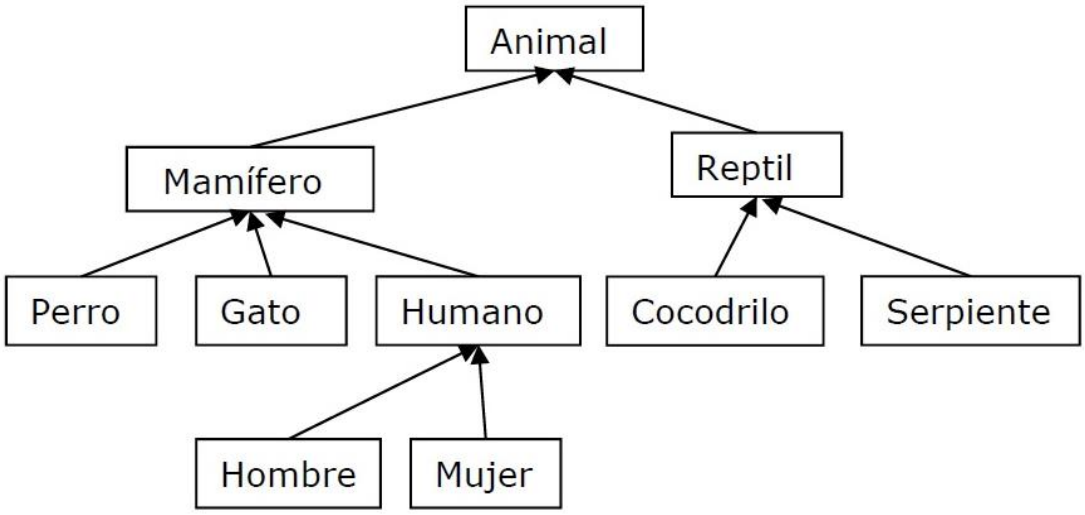


Ilustración: relación de Herencia entre la Clase Animal y sus clases hijas

¿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**.

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

Es una relación que representa a los **objetos compuestos por otros objetos**. Indica Objetos que a su vez están formados por otros. El objeto en el nivel superior de la jerarquía es el **todo** y los que están en los niveles inferiores son sus **partes** o **componentes**.

Representación en el Diagrama de Clase

La relación *forma parte de*, no es más que una **asociación**, que se denota:

La **multiplicidad** es el rango de cardinalidad permitido que puede asumir la asociación, se denota LI..LS. Se puede usar * en el límite superior para representar una cantidad ilimitada (ejemplo: 3..*).

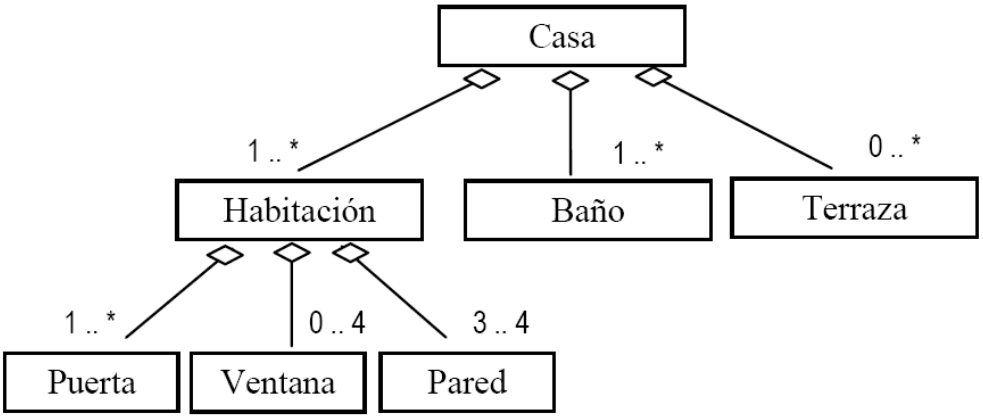


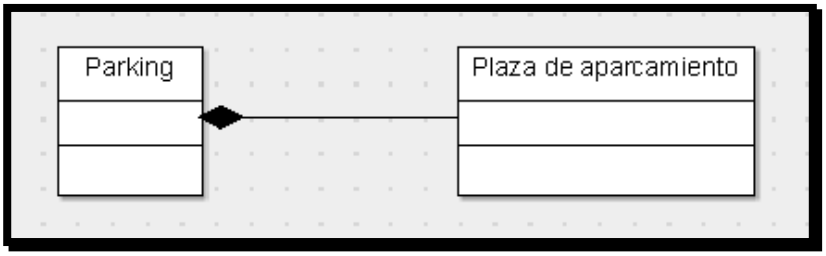
Ilustración: Objeto Casa descrito en términos de sus componentes

10.3 Relación de composición

Un componente es **parte esencial** de un elemento. Este tipo de relación es más fuerte que el caso de agregación, hasta el punto de que, si el componente es eliminado o desaparece, la clase mayor deja de existir.

Representación en el Diagrama de Clases

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



- La clase Parking **agrupa varios elementos** del tipo *Plaza_de_aparcamiento.*

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

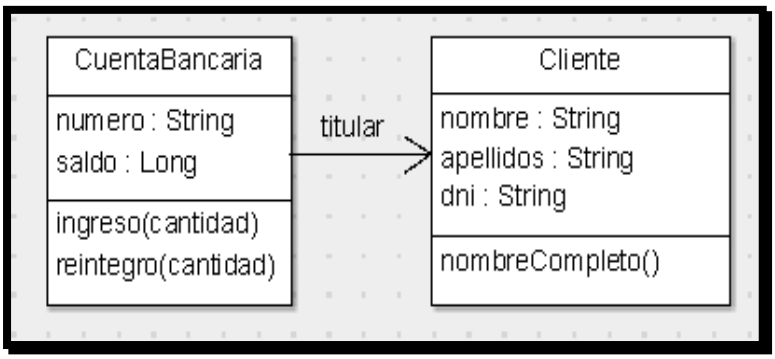
10.4 Relación de asociación («uso», usa, cualquier otra relación)

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.

El ejemplo más común de esta relación es de objetos que son utilizados por los humanos para alguna función, como Lápiz (se usa para escribir), tenedor (se usa para comer), silla (se usa para sentarse), etc. Otro ejemplo son los objetos Cajero y Cuenta.

El Cajero "*usa a*" la cuenta para hacer las transacciones de consulta y retirada y verificar la información del usuario.

Representación en el Diagrama de Clases



- 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.

Ilustración: ejemplo relación de asociación

ENLACE DE INTERÉS

Puede ampliar información y ver más ejemplos sobre las relaciones de asociación en la POO visitando la web:

http://www.cristalab.com/tutoriales/programacion-orientada-a-objetos-asociacion-vs-composicion-c89337l/

10.5 Software de modelado

A la hora de saber qué y cómo programar, se debe tener claro cómo se quiere que sea el sistema. Para ello se dispone del Lenguaje Unificado de Modelado (UML, por sus siglas en inglés, Unified Modeling Language).

UML 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.

ENLACE DE INTERÉS

Para ampliar información sobre el lenguaje UML se recomienda visitar las siguientes referencias web:

https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado

https://msdn.microsoft.com/es-es/library/bb972214.aspx

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

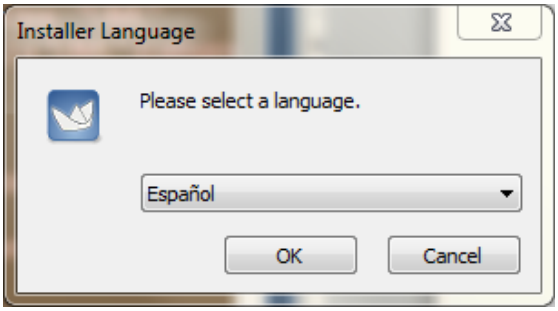
De entre ellas cabe destacar:

- ArgoUML
- Papyrus – Plugin para Eclipse

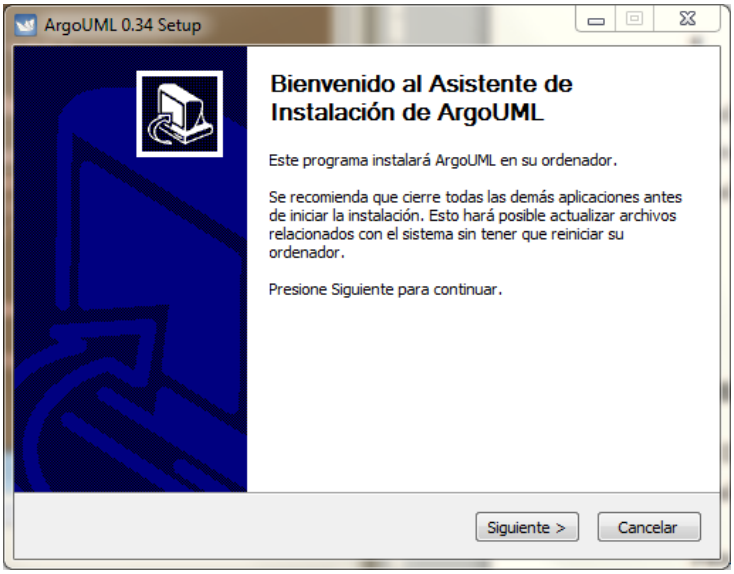
Instalación ArgoUML

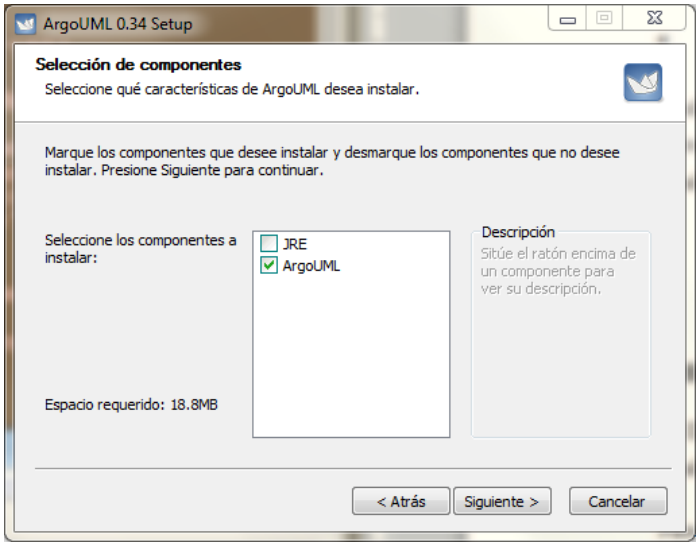
Se realizará la descarga de la última versión de ArgoUML https://argouml.uptodown.com/windows y se procederá a instalarla en el sistema. Es una herramienta sencilla de modelado, compatible con cualquier sistema y está disponible en diversos lenguajes.

Se seleccionará el lenguaje del software ArgoUML:

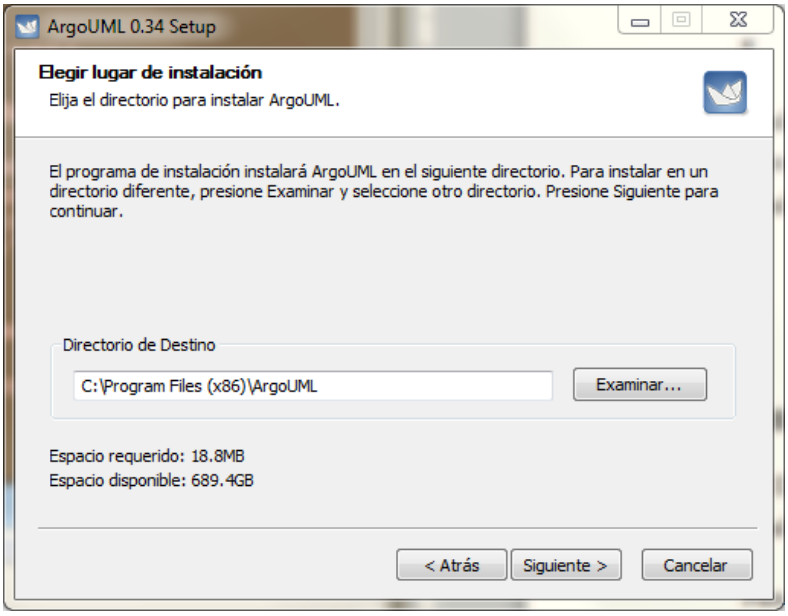


Se siguen los pasos que indica el asistente de instalación de ArgoUML:

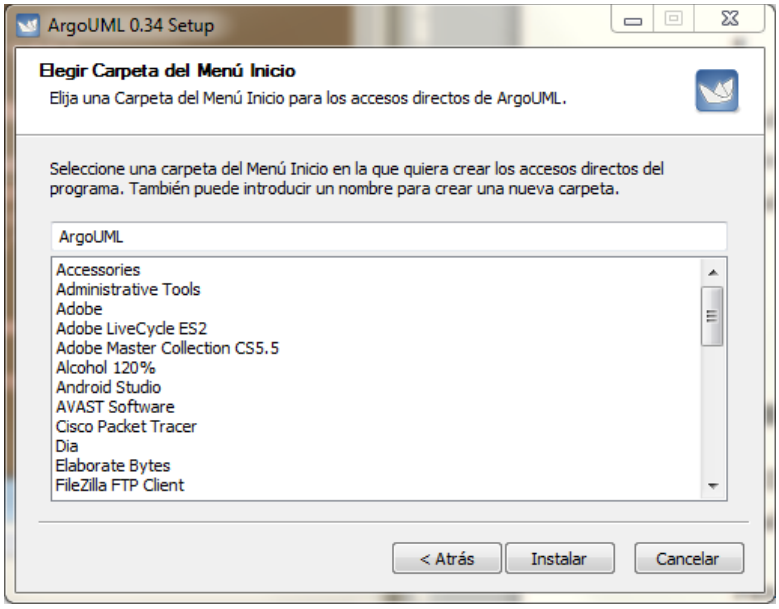




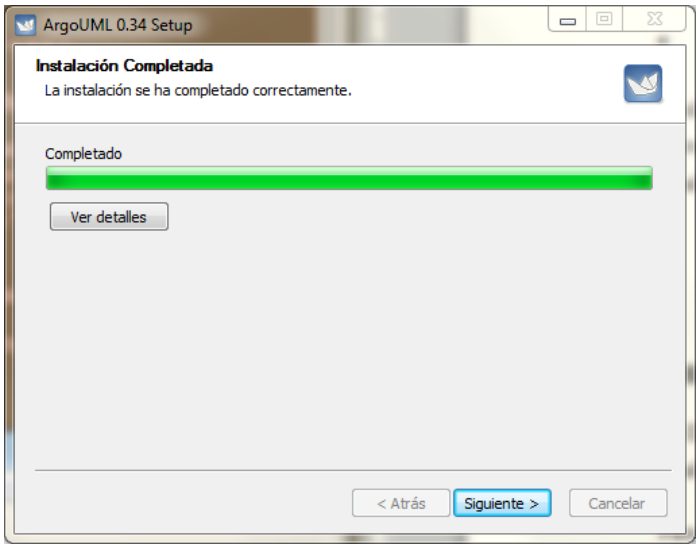
Se indica el directorio en el cual se instalará el software. Se recomienda dejar el que se propone por defecto:



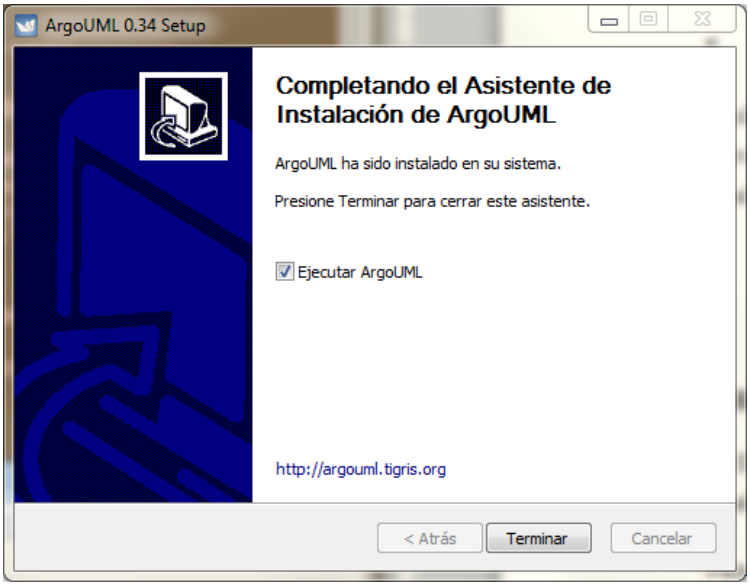
Pulsamos en el botón `Siguiente >`

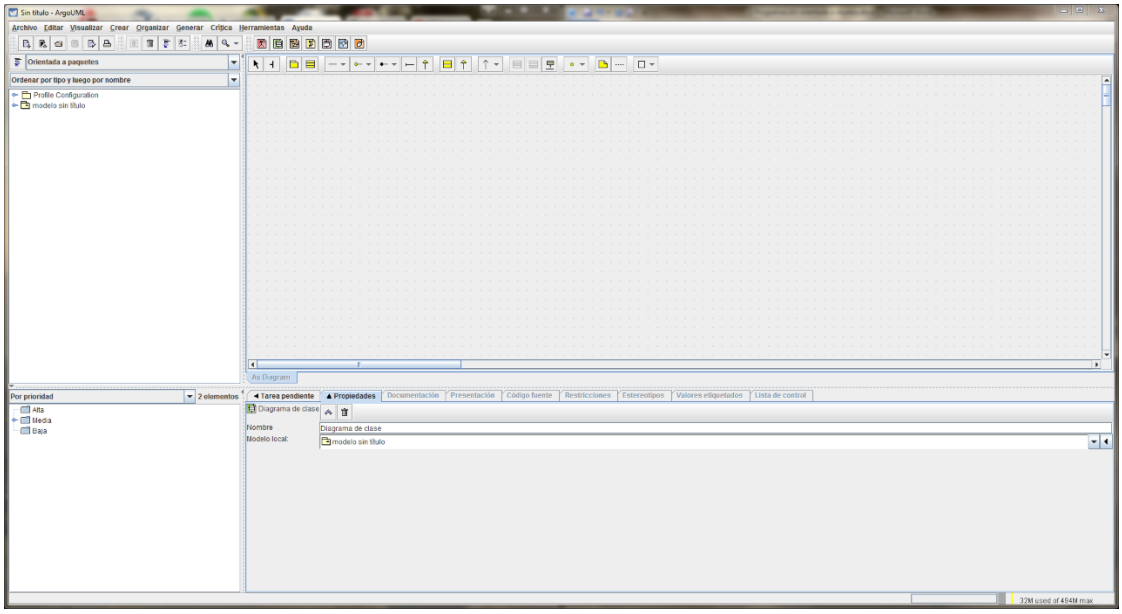


Se pulsará sobre el botón `Instalar`



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:





ENLACE DE INTERÉS

Manual y curso básico de ArgoUML:

http://www.lawebdelprogramador.com/cursos/UML/6589-Manual-Basico-de-ArgoUML.html

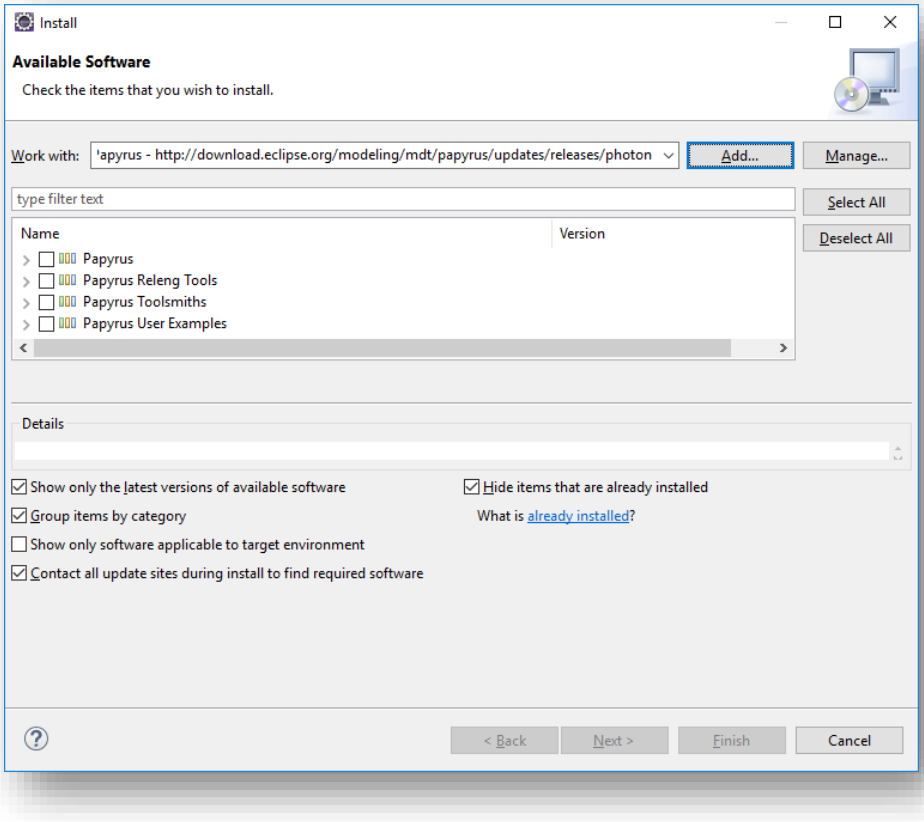
Instalación Plugin Papyrus en Eclipse

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

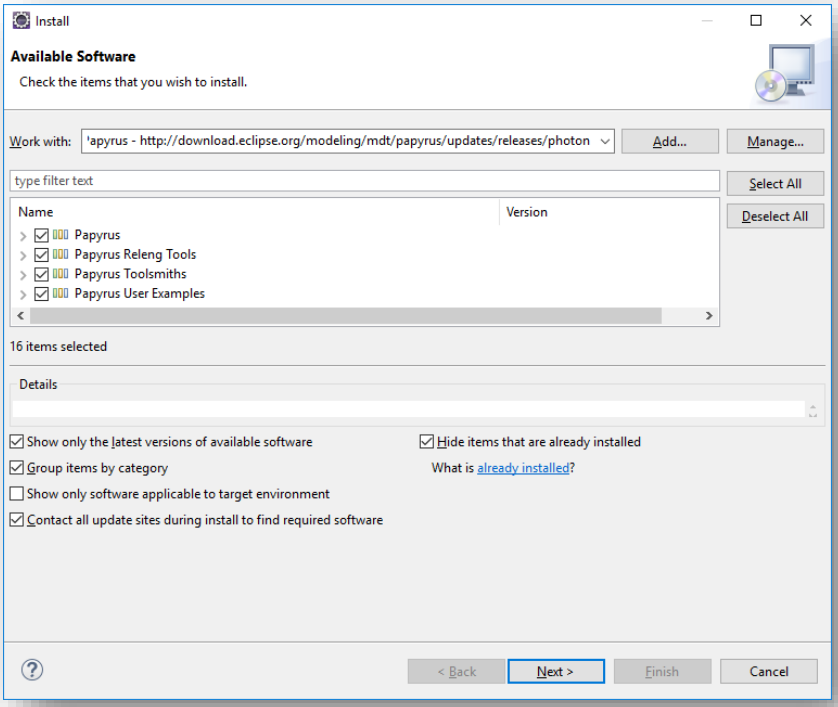
Para ello se deben seguir los siguientes pasos:

Se abre el IDE Eclipse, y se selecciona la opción `Help -> Install New Software`. Se tiene que añadir la url de descarga de Papyrus:

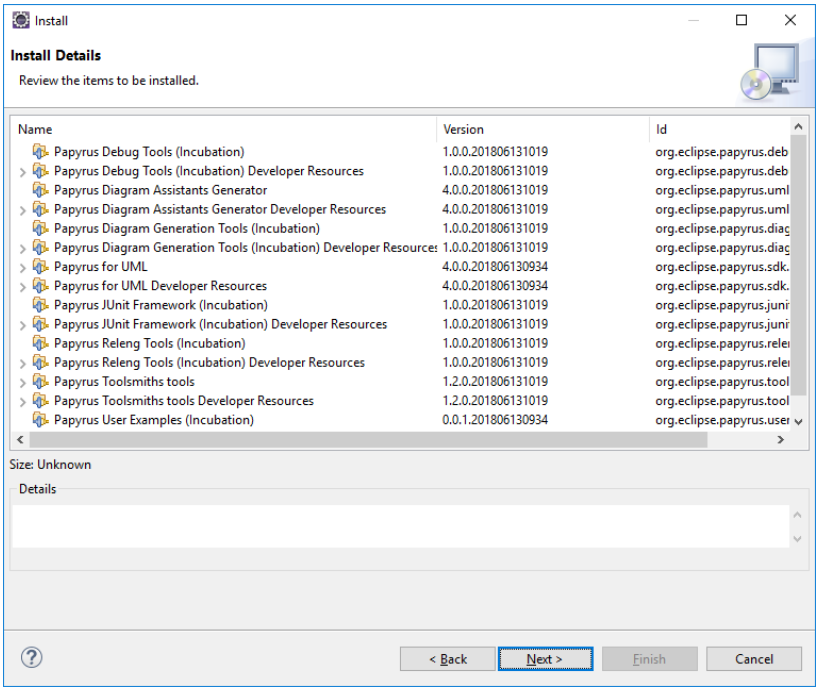
http://download.eclipse.org/modeling/mdt/papyrus/updates/releases/photon

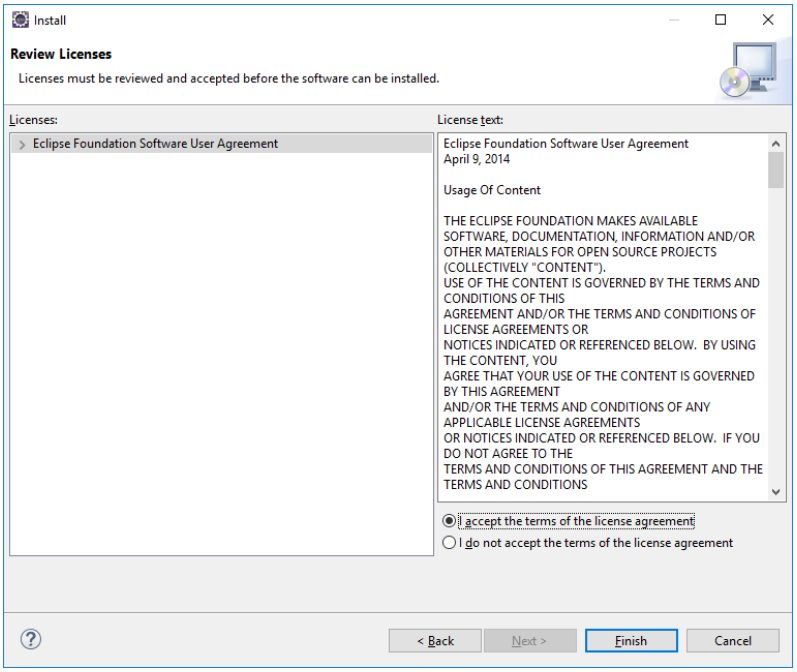


A continuación, se seleccionan todos los ítems y se pulsa sobre el botón `Next >`

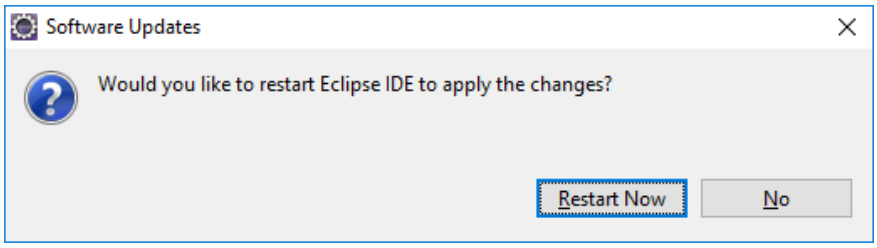


Se pulsa de nuevo en el botón `Next >`



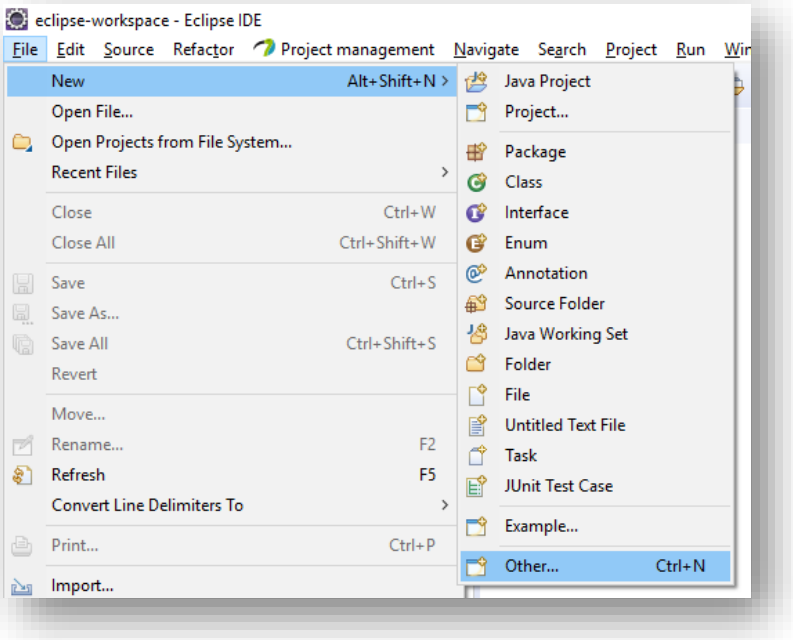


Se aceptan los términos de la licencia y se pulsa el botón `Finish`

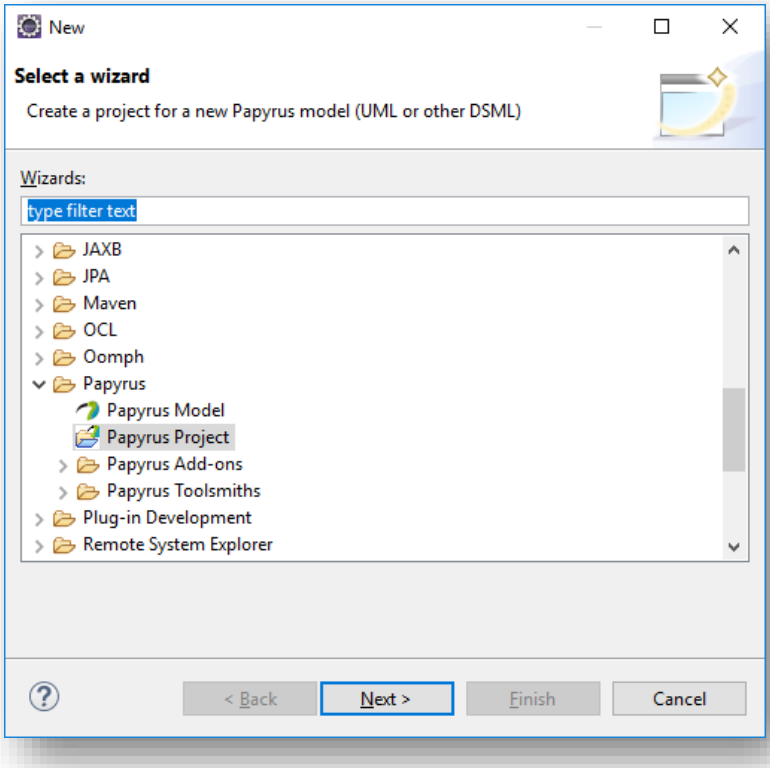


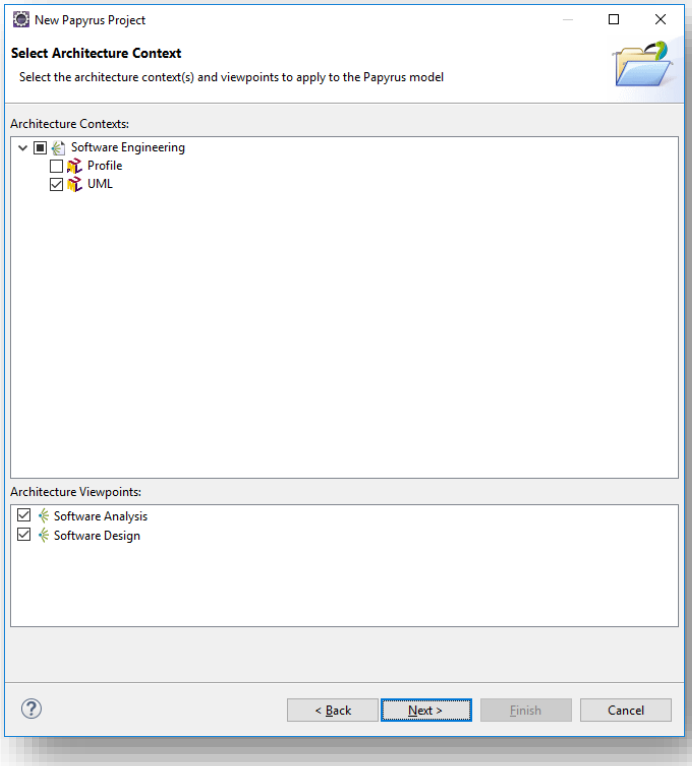
Una vez que se instala hay que pulsar el botón `Restart Now` para finalizar la instalación.

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

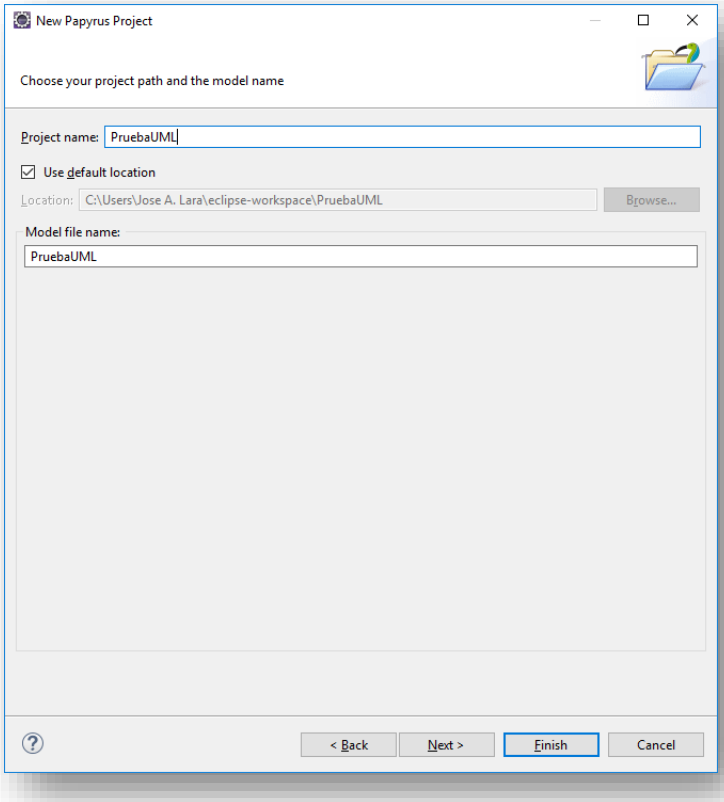


Se selecciona `Papyrus Project` y se pulsa `Next >`

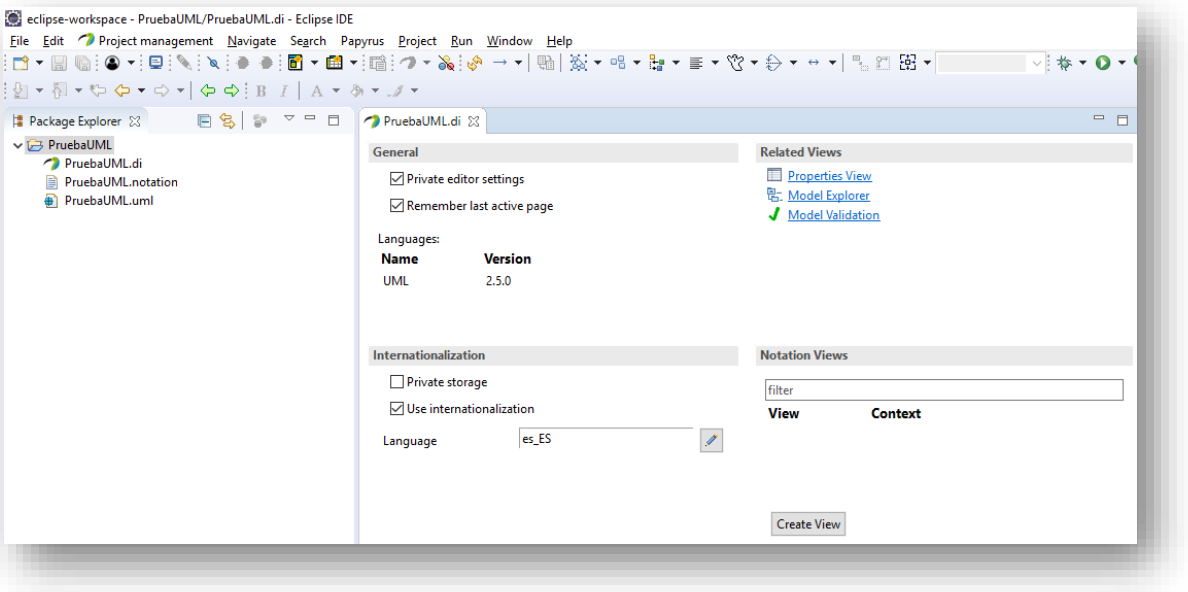




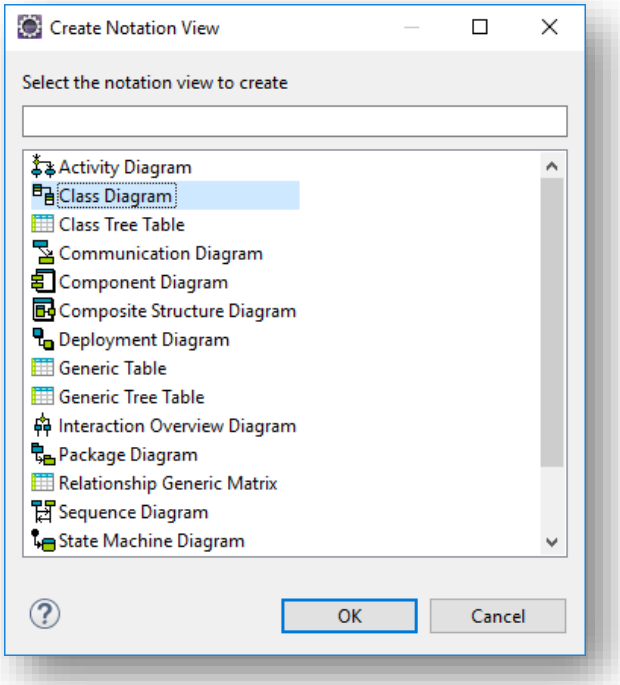
Se selecciona la opción UML y se pulsa `Next >`

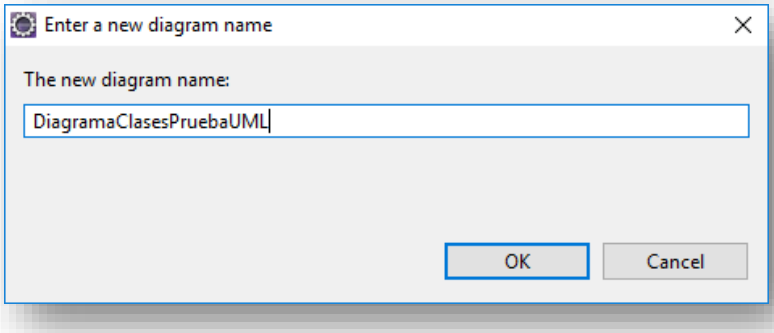


Finalmente se pulsa sobre `Finish`

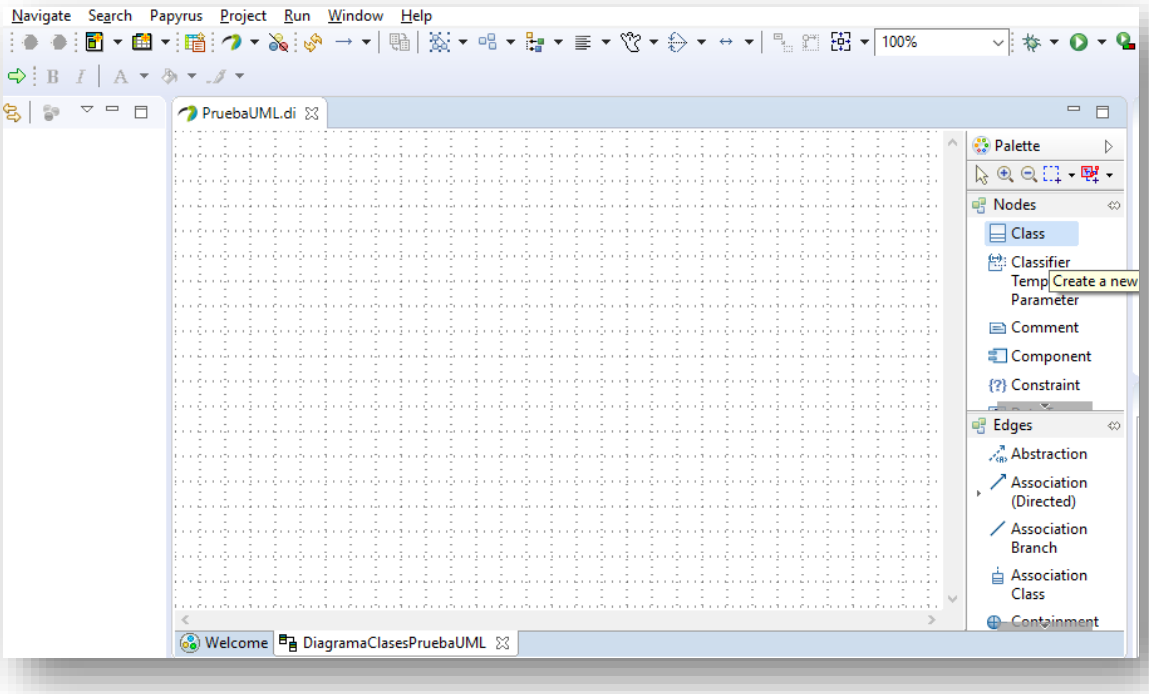


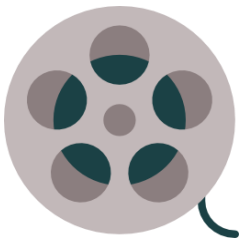
Se pulsa sobre el botón que aparece en la parte de abajo `Create View`**, y se selecciona el tipo de diagrama que se quiere crear. Se le da un nombre**





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





VIDEO DE INTERÉS

En el siguiente enlace podrá ver un vídeo tutorial de Papyrus:

https://www.youtube.com/watch?v=aMiqJXWfAtQ

11. FUNDAMENTOS DEL ENFOQUE ORIENTADO A OBJETO

El Enfoque Orientado a Objeto se basa en **cuatro principios** que constituyen la base de todo desarrollo orientado a objetos.
Estos principios son:

Otros elementos a 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 **especialización** es lo contrario de la generalización.

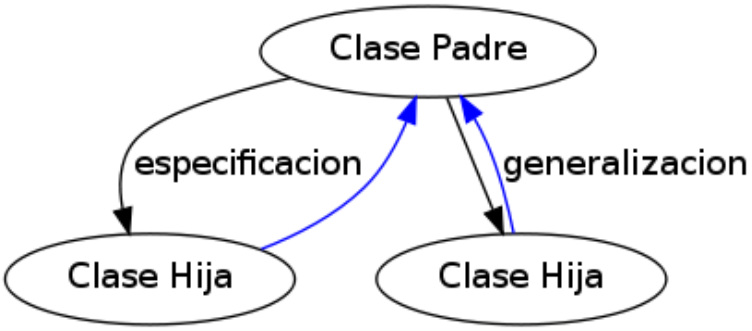


Ilustración: generalización y especificación

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**.

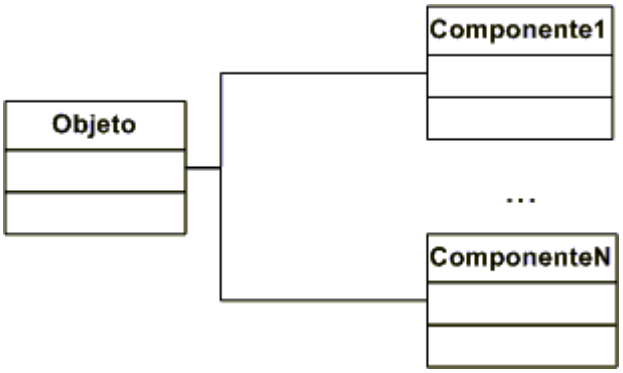


Ilustración: Agregación

Ejemplo: Mediante agregación se puede definir por ejemplo un computador, por 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.

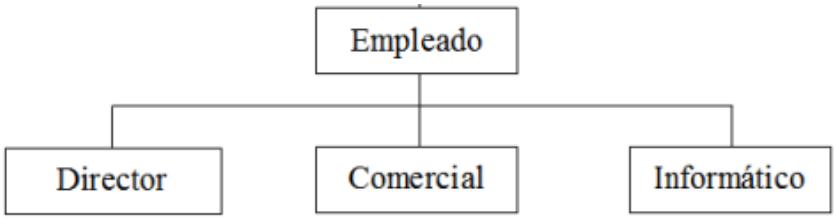


Ilustración: clasificación

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.

ENLACE DE INTERÉS

Puedes ampliar información sobre el encapsulamiento en la Programación Orientada a Objetos visitando la web:

http://www.mastermagazine.info/termino/4880.php

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: Por 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 a la clase invocada, sin embargo, el nombre del método es el mismo.

Mamifero.Comer ≠ Ave.Comer ≠ Pez.Comer

Ejemplo: Otro ejemplo de polimorfismo es 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.

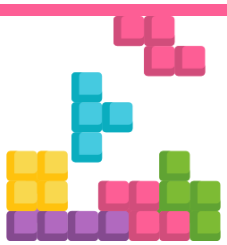
Ejemplo: Otro ejemplo de sobrecarga se puede dar cuando se tiene un método definido originalmente en la clase padre, que ha sido adaptado o modificado en la clase hija. Por ejemplo, un método `Comer` para la clase `Animal` y otro `Comer` que ha sido adaptado para la clase `Ave`, quien está heredando de la clase `Animal`.

Cuando se desea indicar que se está invocando (o llamando) a un método sobrecargado y que pertenece a otra clase (por ejemplo, a la clase padre) lo indicamos con la siguiente sintaxis:

```
Clase_Madre::nombre_método;
```

Para el ejemplo, la llamada en la clase hija `Ave` del método sobrecargado `Comer` de la clase madre `Animal` sería:

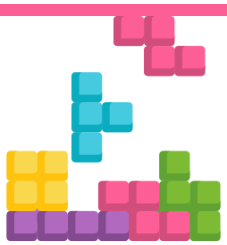
```
Animal::Comer;
```



EJEMPLO PRÁCTICO

Ejemplo 1, Definición de la Clase Rectángulo

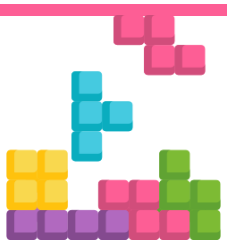
```
Clase Rectángulo;
        // Atributos
        Privado:
                Real Largo, Ancho;
        // Métodos
        Constructor Rectángulo(Real lar, anc);
                Largo = lar;
                Ancho = anc;
        FConstructor;

        Público Función Área: Real
        // Retorna el área o superficie ocupada por el rectángulo
                retornar(Largo * Ancho);
        FFunción Área;

        Público Función Perímetro: Real
        // Retorna el perímetro del rectángulo
                retornar(2 * (Largo + Ancho));
        FFunción Perímetro;
FClase Rectángulo;

Clase Principal
        Acción Usa_Rectángulo
        Rectángulo R;   // se declara una variable de tipo objeto Rectángulo, a la cual llamaremos R

        Real L, A;      // se declaran las variables reales L y A para leer el largo y ancho dado por
                                // el usuario
        Escribir("Suministre a continuación los valores para el largo y el ancho");
        Leer(L, A);
        R.Rectángulo(L, A);
        Escribir("Resultados de los cálculos:");
        Escribir("Área: "+ R.Área + " - Perímetro " + R.Perímetro);
        FAcción Usa_Rectángulo;
FClase Principal;
```



EJEMPLO PRÁCTICO

Ejemplo 2, Clase Punto3D que se deriva de la clase Punto2D

```
Clase Punto3D Hereda de Punto2D;
        Protegido Real Z; // coordenada Z del punto
        Público:
                Acción Punto3D
                // Crea un punto con coordenadas (0,0,0) reutilizando el constructor de un
                        // punto de 2D
                        Punto2D;
                        Z = 0;
                FAcción;
                Acción Punto3D(Real CX, CY, CZ)
                // Crea un punto de tres coordenadas reutilizando el constructor de un punto
                // de 2D
                        Punto2D(CX, CY);
                        Z = CZ;
                FAcción;
                // otros métodos de la clase
                Función CoordenadaZ: Real
                // devuelve el valor de la coordenada Z del punto que hace la llamada
                        retornar(Z);
                Ffunción;
                Acción CambiarZ(Real NZ)
                // modifica el valor de la coordenada Z del punto que hace la llamada
                        Z = NZ;
                Ffunción;
                Función Distancia3D(Real CX, CY, CZ): Real
                // devuelve la distancia entre el punto actual y otro con coordenadas CX, CY
                // y CZ
                        Real D; //contiene la distancia
                        D = RestaC(X, CX) + RestaC(Y, CY) + RestaC(Z, CZ);
                        D = D ∧ (1/2);
                        retornar(D);
                Ffunción;
                Función Distancia3D(Ref Punto3D P): Real
                // devuelve la distancia entre el punto actual y otro punto P
                        retornar(Distancia3D(P.X, P.Y, P.Z));
                Ffunción;
FClase Punto3D
```



EJEMPLO PRÁCTICO

Ejemplo 3, Clase Cuenta Bancaria

```
Clase CuentaBancaria
        // atributos
        Privado Entero Saldo;
        Privado String NroCuenta;
        Privado String Titular;

        // métodos
        Acción CuentaBancaria(Entero montoInicial; String num, nombre)
        // asigna a los atributo de la clase sus valores iniciales
                Saldo = montoInicial;
                NroCuenta = num;
                Titular = nombre;
        Facción;

        Público Acción depositar(Entero cantidad)
        // incrementa el saldo de la cuenta
                Saldo = Saldo + cantidad;
        Facción;

        Público Acción retirar(Entero cantidad)
        // disminuye el saldo de la cuenta
                Saldo = Saldo - cantidad;
        Facción;

        Público Función obtenerSaldo: Entero
        // permite conocer el monto disponible o saldo de la cuenta
                Retornar(saldo);
        FFunción;
FinClase CuentaBancaria
```

COMPRUEBA LO QUE SABES

Analizando el ejemplo 2:

- ¿Qué atributos y métodos hereda la clase Punto3D de la clase Punto2D?
- ¿Qué atributos y métodos hereda la clase Punto2D de la clase Punto3D?
- ¿Quiénes pueden usar los atributos y métodos de Punto2D?

ENLACE DE INTERÉS

Para ampliar información sobre más fundamentos del enfoque orientado a objetos se recomienda visitar:

http://fpsalmon.usc.es/genp/doc/cursos/poo/modelo.html

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. A continuación, se han identificado los fundamentos básicos de este enfoque y uno de los principales conceptos del paradigma: la herencia.