C.F.G.S. DESARROLLO DE APLICACIONES WEB

MÓDULO: Entornos de Desarrollo

# Unidad 5

## Elaboración de diagramas de clase.

**ÍNDICE DE CONTENIDOS**

## OBJETIVOS

Los objetivos generales de aprendizaje de esta unidad son generar diagramas de clases, valorando su importancia en el desarrollo de aplicaciones y empleando las herramientas disponibles en el entorno.

## 1.- Introducción a la orientación a objetos

La construcción de software es un proceso cuyo objetivo es dar solución a problemas utilizando una herramienta informática y tiene como resultado la construcción de un programa informático. Como en cualquier otra disciplina en la que se obtenga un producto final de cierta complejidad, si queremos obtener un producto de calidad, es preciso realizar un proceso previo de análisis y especificación del proceso que vamos a seguir, y de los resultados que pretendemos conseguir.

- **El enfoque estructurado**.

Sin embargo, cómo se hace es algo que ha ido evolucionando con el tiempo, en un principio se tomaba el problema de partida y se iba sometiendo a un proceso de división en subproblemas más pequeños reiteradas veces, hasta que se llegaba a problemas elementales que se podía resolver utilizando una función. Luego las funciones se hilaban y entretejían hasta formar una solución global al problema de partida. Era, pues, un proceso centrado en los procedimientos, se codificaban mediante **funciones** que actuaban sobre **estructuras de datos**, por eso a este tipo de programación se le llama **programación estructurada**. Sigue una filosofía en la que se intenta aproximar qué hay que hacer, para así resolver un problema.

- **Enfoque orientado a objetos**.

La orientación a objetos ha roto con esta forma de hacer las cosas. Con este nuevo **paradigma** el proceso se centra en simular los elementos de la realidad asociada al problema de la forma más cercana posible.

La **abstracción** que permite representar estos elementos se denomina objeto, y tiene las siguientes características:

- Está formado por un conjunto de **atributos**, que son los datos que le caracterizan y
- Un conjunto de **operaciones** que definen su comportamiento. Las operaciones asociadas a un objeto actúan sobre sus atributos para modificar su estado. Cuando se indica a un objeto que ejecute una operación determinada se dice que se le pasa un **mensaje**.

Las aplicaciones orientadas a objetos están formadas por un conjunto de objetos que interaccionan enviándose mensajes para producir resultados. Los objetos similares se abstraen en clases, se dice que un objeto es una instancia de una clase.

Cuando se ejecuta un programa orientado a objetos ocurren tres sucesos:

- Primero, los objetos se crean a medida que se necesitan.
- Segundo. Los mensajes se mueven de un objeto a otro (o del usuario a un objeto) a medida que el programa procesa información o responde a la entrada del usuario.
- Tercero, cuando los objetos ya no se necesitan, se borran y se libera la memoria.

## 2.- Conceptos de orientación a objetos.

Como hemos visto la orientación a objetos trata de acercarse al contexto del problema lo más posible por medio de la simulación de los elementos que intervienen en su resolución y basa su desarrollo en los siguientes conceptos:

- **Abstracción**: Permite capturar las características y comportamientos similares de un conjunto de objetos con el objetivo de darles una descripción formal. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad, o el problema que se quiere atacar.

- **Encapsulación**: Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la **cohesión** de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.

- **Modularidad**: Propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. En orientación a objetos es algo consustancial, ya que los objetos se pueden considerar los módulos más básicos del sistema.

- **Principio de ocultación**: Aísla las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas. Reduce la propagación de efectos colaterales cuando se producen cambios.
- **Polimorfismo**: Consiste en reunir bajo el mismo nombre comportamientos diferentes. La selección de uno u otro depende del objeto que lo ejecute.

- **Herencia**: Relación que se establece entre objetos en los que unos utilizan las propiedades y comportamientos de otros formando una jerarquía. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.

- **Recolección de basura**: Técnica por la cual el entorno de objetos se encarga de destruir automáticamente los objetos, y por tanto desvincular su memoria asociada, que hayan quedado sin ninguna referencia a ellos.

### 2.1.- Ventajas de la orientación a objetos.

Este paradigma tiene las siguientes **ventajas** con respecto a otros:

1. Permite desarrollar software en mucho menos tiempo, con menos coste y de mayor calidad gracias a la **reutilización** porque al ser completamente modular facilita la creación de código reusable dando la posibilidad de reutilizar parte del código para el desarrollo de una aplicación similar.

2. Se consigue aumentar la calidad de los sistemas, haciéndolos más **extensibles** ya que es muy sencillo aumentar o modificar la funcionalidad de la aplicación modificando las operaciones.

3. El software orientado a objetos es más **fácil de modificar y mantener** porque se basa en criterios de modularidad y encapsulación en el que el sistema se descompone en objetos con unas responsabilidades claramente especificadas e independientes del resto.

4. La tecnología de objetos facilita la adaptación al entorno y el cambio haciendo aplicaciones **escalables**. Es sencillo modificar la estructura y el comportamiento de los objetos sin tener que cambiar la aplicación.

### 2.2.- Clases, atributos, métodos.

Los objetos de un sistema se abstraen, en función de sus características comunes, en clases. Una clase está formada por un conjunto de procedimientos y datos que resumen características similares de un conjunto de objetos. La clase tiene dos propósitos: definir **abstracciones** y favorecer la **modularidad**.

Una clase se describe por un conjunto de elementos que se denominan **miembros** y que son:

- **Nombre**.
- **Atributos**: conjunto de características asociadas a una clase. Pueden verse como una relación binaria entre una clase y cierto dominio formado por todos los posibles valores que puede tomar

cada atributo. Cuando toman valores concretos dentro de su dominio definen el **estado** del objeto. Se definen por su nombre y su tipo, que puede ser simple o compuesto como otra clase.

- **Protocolo**: Operaciones (métodos, mensajes) que manipulan el estado.
    - Un ***método*** es el procedimiento o función que se invoca para actuar sobre un objeto.
    - Un ***mensaje*** es el resultado de cierta acción efectuada por un objeto.

    Los métodos determinan como actúan los objetos cuando reciben un mensaje, es decir, cuando se requiere que el objeto realice una acción descrita en un método se le envía un mensaje. El conjunto de mensajes a los cuales puede responder un objeto se le conoce como *protocolo del objeto*.

Por ejemplo, si tenemos un objeto icono, tendrá como atributos el tamaño, o la imagen que muestra, y su protocolo puede constar de mensajes invocados por el clic del botón de un ratón cuando el usuario pulsa sobre el icono. De esta forma los mensajes son el único conducto que conectan al objeto con el mundo exterior.

Los valores asignados a los atributos de un objeto concreto hacen a ese objeto ser **único**. La clase define sus características generales y su comportamiento.

**2.3.- Visibilidad.**

El principio de ocultación es una propiedad de la orientación a objetos que consiste en aislar el estado de manera que sólo se puede cambiar mediante las operaciones definidas en una clase. Este aislamiento protege a los datos de que sean modificados por alguien que no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.
Da lugar a que las clases se dividan en dos partes:

1. **Interfaz**: captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.
2. **Implementación**: comprende como se representa la abstracción, así como los mecanismos que conducen al comportamiento deseado.

Existen distintos niveles de ocultación que se implementan en lo que se denomina **visibilidad**. Es una característica que define el tipo de acceso que se permite a atributos y métodos y que podemos establecer como:

- **Público**: Se pueden acceder desde cualquier clase y cualquier parte del programa.
- **Privado**: Sólo se pueden acceder desde operaciones de la clase.
- **Protegido**: Sólo se pueden acceder desde operaciones de la clase o de **clases derivadas** en cualquier nivel.

Como norma general a la hora de definir la visibilidad tendremos en cuenta que:

- El estado debe ser privado. Los atributos de una clase se deben modificar mediante métodos de la clase creados a tal efecto.
- Las operaciones que definen la funcionalidad de la clase deben ser públicas.
- Las operaciones que ayudan a implementar parte de la funcionalidad deben ser privadas (si no se utilizan desde clases derivadas) o protegidas (si se utilizan desde clases derivadas).

**2.4.- Objetos. Instanciación.**

Cada vez que se construye un objeto en un programa informático a partir de una clase se crea lo que se conoce como **instancia** de esa clase. Cada instancia en el sistema sirve como modelo de un objeto del contexto del problema relevante para su solución, que puede realizar un trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema, sin revelar cómo se implementan estas características.

Un objeto se define por:

- **Su estado**: es la concreción de los atributos definidos en la clase a un valor concreto.
- **Su comportamiento**: definido por los métodos públicos de su clase.
- **Su tiempo de vida**: intervalo de tiempo a lo largo del programa en el que el objeto existe.

Comienza con su creación a través del mecanismo de **instanciación** y finaliza cuando el objeto se destruye.

La encapsulación y el ocultamiento aseguran que los datos de un objeto están ocultos, con lo que no se pueden modificar accidentalmente por funciones externas al objeto.

**Citas para pensar**

Mientras que un objeto es una entidad que existe en el tiempo y el espacio, una clase representa sólo una abstracción, "la esencia" del objeto, si se puede decir así. *Grady Booch*

Ejemplo de objetos:

1. **Objetos físicos**: aviones en un sistema de control de tráfico aéreo, casas, parques.
2. **Elementos de interfaces gráficas de usuario**: ventanas, menús, teclado, cuadros de diálogo.
3. **Animales**: animales vertebrados, animales invertebrados.
4. **Tipos de datos definidos por el usuario**: Datos complejos, Puntos de un sistema de coordenadas.
5. **Alimentos**: carnes, frutas, verduras.

Existe un caso particular de clase, llamada **clase abstracta**, que, por sus características, no puede ser instanciada. Se suelen usar para definir métodos genéricos relacionados con el sistema que no serán traducidos a objetos concretos, o para definir métodos de base para clases derivadas.

## 3.- UML.

**Citas para pensar**

Una empresa de software con éxito es aquella que produce de manera consistente software de calidad que satisface las necesidades de los usuarios. El modelado es la parte esencial de todas las actividades que conducen a la producción de software de calidad.

**UML** (*Unified Modeling Language o Lenguaje Unificado de Modelado*) es un conjunto de herramientas que permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos. Se ha convertido en el estándar de facto de la industria, debido a que ha sido concebido por los autores de los tres métodos más usados de orientación a objetos: Grady Booch, Ivar Jacobson y Jim Rumbaugh.
Las raíces técnicas de UML son:

- OMT - Object Modeling Technique (Rumbaugh et al.)
- Método-Booch (G. Booch)
- OOSE - Object-Oriented Software Engineering (I. Jacobson)

UML permite a los desarrolladores y desarrolladoras visualizar el producto de su trabajo en esquemas o diagramas estandarizados denominados **modelos** que representan el sistema desde diferentes perspectivas.

**¿Por qué es útil modelar?**

- Porque permite utilizar un lenguaje común que facilita la comunicación entre el equipo de desarrollo.

- Con UML podemos documentar todos los **artefactos** de un proceso de desarrollo (**requisitos**, arquitectura, pruebas, versiones,...) por lo que se dispone de documentación que trasciende al proyecto.
- Hay estructuras que trascienden lo representable en un lenguaje de programación, como las que hacen referencia a la **arquitectura del sistema**, utilizando estas tecnologías podemos incluso indicar qué módulos de software vamos a desarrollar y sus relaciones, o en qué nodos hardware se ejecutarán cuando trabajamos con sistemas distribuidos.
- Permite especificar todas las decisiones de análisis, diseño e implementación, construyéndose modelos precisos, no ambiguos y completos.

Además UML puede conectarse a lenguajes de programación mediante **ingeniería directa** e **inversa**, como veremos.

### 3.1.- Tipos de diagramas UML.

UML describe el sistema mediante una serie de modelos que ofrecen diferentes puntos de vista. Pero ¿qué tenemos que hacer para representar un modelo?, ¿en qué consiste exactamente?.
Utilizaremos diagramas, que son unos **grafos** en los que los nodos definen los elementos del diagrama, y los arcos las relaciones entre ellos.

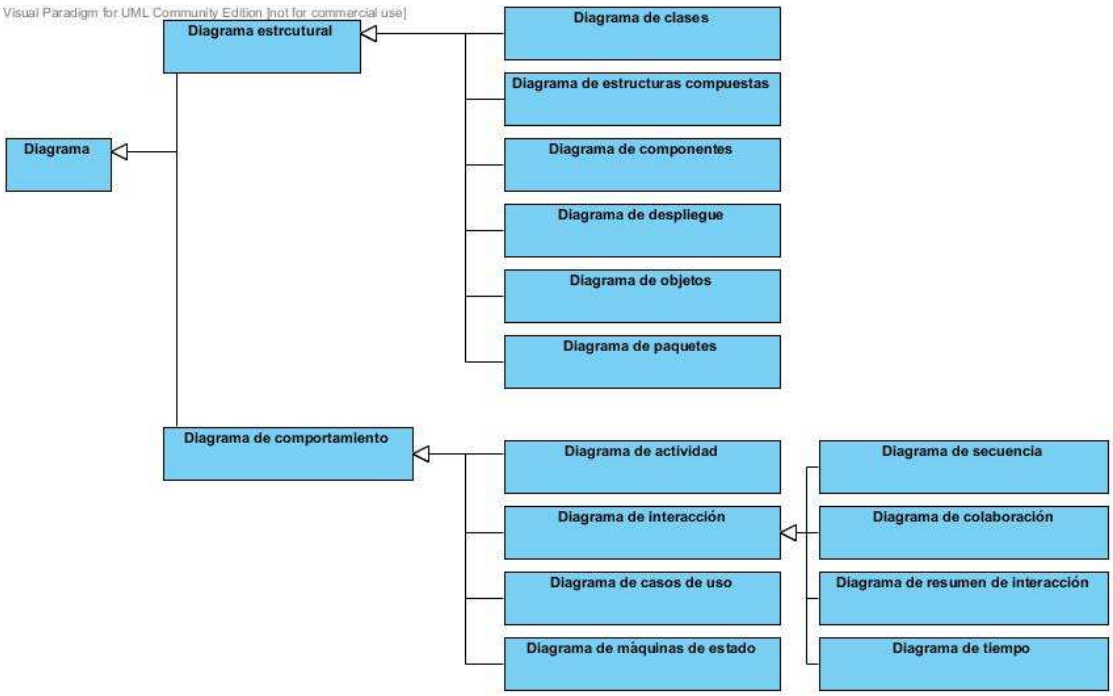
UML define un sistema como una **colección de modelos** que describen sus diferentes perspectivas. Los modelos se implementan en una serie de diagramas que son representaciones gráficas de una colección de elementos de modelado, a menudo dibujado como un grafo conexo de arcos (relaciones) y vértices (otros elementos del modelo).
Un diagrama UML se compone de cuatro tipos de elementos:

- **Estructuras**: Son los nodos del grafo y definen el tipo de diagrama.
- **Relaciones**: Son los arcos del grafo que se establecen entre los elementos estructurales.
- **Notas**: Se representan como un cuadro donde podemos escribir comentarios que nos ayuden a entender algún concepto que queramos representar.
- **Agrupaciones**: Se utilizan cuando modelamos sistemas grandes para facilitar su desarrollo por bloques.

Se clasifican en:

- **Diagramas estructurales**: Representan la visión estática del sistema. Especifican clases y objetos y como se distribuyen físicamente en el sistema.

- **Diagramas de comportamiento**: muestran la conducta en tiempo de ejecución del sistema, tanto desde el punto de vista del sistema completo como de las instancias u objetos que lo integran. Dentro de este grupo están los diagramas de interacción.




En la imagen aparecen todos los diagramas organizados según su categoría. Se han destacado aquellos que perteneces al estándar UML 2.0, más novedoso. En total se describen trece diagramas para modelar diferentes aspectos de un sistema, sin embargo no es necesario usarlos todos, dependerá del tipo de aplicación a generar y del sistema, es decir, se debe generar un diagrama sólo si es necesario.

**Citas para pensar**

Un 80% de las aplicaciones se pueden modelar con el 20% de los diagramas UML.

Una Clasificación un poco mas detallada seria:

■ **Diagramas estructurales.**

- **Diagramas de clases**: Muestra los elementos del modelo estático abstracto, y está formado por un conjunto de clases y sus relaciones. Tiene una prioridad ALTA.

- **Diagrama de objetos**: Muestra los elementos del modelo estático en un momento concreto, habitualmente en casos especiales de un diagrama de clases o de comunicaciones, y está formado por un conjunto de objetos y sus relaciones. Tiene una prioridad ALTA.

- **Diagrama de componentes**: Especifican la organización lógica de la implementación de una aplicación, sistema o empresa, indicando sus componentes, sus interrelaciones, interacciones y sus interfaces públicas y las dependencias entre ellos. Tiene una prioridad MEDIA.

- **Diagramas de despliegue**: Representan la configuración del sistema en tiempo de ejecución. Aparecen los nodos de procesamiento y sus componentes. Exhibe la ejecución de la arquitectura del sistema. Incluye nodos, ambientes operativos sea de hardware o software, así como las interfaces que las conectan, es decir, muestra como los componentes de un sistema se distribuyen entre los ordenadores que los ejecutan. Se utiliza cuando tenemos sistemas distribuidos. Tiene una prioridad MEDIA.

- **Diagrama integrado de estructura** (UML 2.0): Muestra la estructura interna de una clasificación (tales como una clase, componente o caso típico), e incluye los puntos de interacción de esta clasificación con otras partes del sistema. Tiene una prioridad BAJA.

- **Diagrama de paquetes**: Exhibe cómo los elementos del modelo se organizan en paquetes, así como las dependencias entre esos paquetes. Suele ser útil para la gestión de sistemas de mediano o gran tamaño. Tiene una prioridad BAJA.

■ **Diagramas de comportamiento.**

- **Diagramas de casos de uso**: Representan las acciones a realizar en el sistema desde el punto de vista de los usuarios. En él se representan las acciones, los usuarios y las relaciones entre ellos. Sirven para especificar la funcionalidad y el comportamiento de un sistema mediante su interacción con los usuarios y/u otros sistemas. Tiene una prioridad MEDIA.

- **Diagramas de estado de la máquina**: Describen el comportamiento de un sistema dirigido por eventos. En él aparecen los estados que pueden tener un objeto o interacción, así como las transiciones entre dichos estados. Se lo denomina también diagrama de estado, diagrama de estados y transiciones o diagrama de cambio de estados. Tiene una prioridad MEDIA.

- **Diagrama de actividades**: Muestran el orden en el que se van realizando tareas dentro de un sistema. En él aparecen los procesos de alto nivel de la organización. Incluye flujo de datos, o un modelo de la lógica compleja dentro del sistema. Tiene una prioridad ALTA.

- **Diagramas de interacción.**
  - **Diagramas de secuencia**: Representan la ordenación temporal en el paso de mensajes. Modela la secuencia lógica, a través del tiempo, de los mensajes entre las instancias. Tiene una prioridad ALTA.
  - **Diagramas de comunicación/colaboración (UML 2.0)**: Resaltan la organización estructural de los objetos que se pasan mensajes. Ofrece las instancias de las clases, sus interrelaciones, y el flujo de mensajes entre ellas. Comúnmente enfoca la organización estructural de los objetos que reciben y envían mensajes. Tiene una prioridad BAJA.
  - **Diagrama de interacción**: Muestra un conjunto de objetos y sus relaciones junto con los mensajes que se envían entre ellos. Es una variante del diagrama de actividad que permite mostrar el flujo de control dentro de un sistema o proceso organizativo. Cada nodo de actividad dentro del diagrama puede representar otro diagrama de interacción. Tiene una prioridad BAJA.
  - **Diagrama de tiempos**: Muestra el cambio en un estado o una condición de una instancia o un rol a través del tiempo. Se usa normalmente para exhibir el cambio en el estado de un objeto en el tiempo, en respuesta a eventos externos. Tiene una prioridad BAJA

### 3.2.- Herramientas para la elaboración de diagramas UML.

Lo que nos permite conocer a un buen desarrollador es que siempre hace un buen esquema inicial de cada proyecto, y eso puede hacerse en miles de soportes, desde una libreta a un servilleta, cualquier cosa que te permita hacer un pequeño dibujo. El uso de herramientas, además de facilitar la elaboración de los diagramas, tiene otras ventajas, como la integración en entornos de desarrollo, con lo que podremos generar el código base de nuestra aplicación desde el propio diagrama.

La herramienta más simple que se puede utilizar para generar diagramas es lápiz y papel, hoy día, sin embargo, podemos acceder a **herramientas CASE** que facilitan en gran manera el desarrollo de los diagramas UML. Estas herramientas suelen contar con un entorno de ventanas tipo **WYSIWYG**, permiten documentar los diagramas e integrarse con otros entornos de desarrollo incluyendo la generación automática de código y procedimientos de ingeniería inversa.

Podemos encontrar, entre otras, las siguientes herramientas:

- **Rational Systems Developer de IBM**: Herramienta propietaria que permite el desarrollo de proyectos software basados en la metodología UML. Desarrollada en origen por los creadores de UML ha sido recientemente absorbida por IBM. Ofrece versiones de prueba, y software libre para el desarrollo de diagramas UML.

- **Visual Paradigm for UML (VP-UML):** Incluye una versión para uso no comercial que se distribuye libremente sin más que registrarse para obtener un archivo de licencia. Incluye diferentes módulos para realizar desarrollo UML, diseñar bases de datos, realizar actividades de ingeniería inversa y diseñar con Agile. Es compatible con los IDE de Eclipse, Visual Studio .net, IntellijDEA y NetBeans. **Multiplataforma**, incluye instaladores para Windows y Linux.

- **ArgoUML**: se distribuye bajo licencia Eclipse. Soporta los diagramas de UML 1.4, y genera código para java y C++. Para poder ejecutarlo se necesita la plataforma java. Admite ingeniería directa e inversa.

### 3.3.- Diagramas de clases.

Empecemos por los diagramas estructurales, entre ellos el más importante es el diagrama de clases, fíjate, representa la estructura estática del sistema y las relaciones entre las clases.
Dentro de los diagramas estructurales, y de todos en general, es el más importante porque representa los elementos estáticos del sistema, sus atributos y comportamientos, y como se relacionan entre ellos. Contiene las clases del **dominio del problema**, y a partir de éste se obtendrán las clases que formarán después el programa informático que dará solución al problema.

En un diagrama de clases podemos encontrar los siguientes elementos:

- **Clases**: recordemos que son abstracciones del dominio del sistema que representan elementos del mismo mediante una serie de características, que llamaremos atributos, y su comportamiento, que serán métodos. Los atributos y métodos tendrán una visibilidad que determinará quién puede acceder al atributo o método. Por ejemplo una clase puede representar a un coche, sus atributos serán la cilindrada, la potencia y la velocidad, y tendrá dos métodos, uno para acelerar, que subirá la velocidad, y otro para frenar que la bajará.

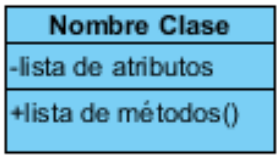
- **Relaciones**: en el diagrama representan relaciones reales entre los elementos del sistema a los que hacen referencia las clases. Pueden ser de asociación, agregación y herencia. Por ejemplo si tengo una clase persona, puedo establecer una relación conduce entre persona y coche.

- **Notas**: Se representan como un cuadro donde podemos escribir comentarios que nos ayuden a entender algún concepto que queramos representar.

- **Elementos de agrupación**: Se utilizan cuando hay que modelar un sistema grande, entonces las clases y sus relaciones se agrupan en **paquetes**, que a su vez se relacionan entre sí.

### 3.3.1.- Creación de clases

Una clase se representa en el diagrama como un rectángulo divido en tres filas, arriba aparece el nombre de la clase, a continuación los atributos con su visibilidad y después los métodos con su visibilidad que está representada por el signo menos "–" para los atributos (privados) y por el signo más "+" para los métodos (públicos).




**Citas para pensar**

"Una clase es una descripción de un conjunto de objetos que manifiestan los mismos atributos, operaciones, relaciones y la misma semántica."

*(Object Modelling and Design [Rumbaught et al., 1991])*

"Una clase es un conjunto de objetos que comparten una estructura y un comportamiento comunes."

[*Booch G., 1994]*

### 3.3.2.- Atributos.

Forman la parte estática de la clase. Son un conjunto de variables para las que es preciso definir:

- Su **nombre**.
- Su **tipo**, puede ser un tipo simple, que coincidirá con el tipo de dato que se seleccione en el lenguaje de programación final a usar, o compuesto, pudiendo incluir otra clase.

Además se pueden indicar otros datos como un valor inicial o su visibilidad. La visibilidad de un atributo se puede definir como:

- **Público:** Se pueden acceder desde cualquier clase y cualquier parte del programa.
- **Privado:** Sólo se pueden acceder desde operaciones de la clase.
- **Protegido:** Sólo se pueden acceder desde operaciones de la clase o de clases derivadas en cualquier nivel.
- **Paquete:** Se puede acceder desde las operaciones de las clases que pertenecen al mismo paquete que la clase que estamos definiendo. Se usa cuando el lenguaje de implementación es Java.

### 3.3.3.- Métodos.

Representan la funcionalidad de la clase, es decir, qué puede hacer. Para definir un método hay que indicar como mínimo su nombre, parámetros, el tipo que devuelve y su visibilidad. También se debe incluir una descripción del método que aparecerá en la documentación que se genere del proyecto.

Existe un caso particular de método, el **constructor** de la clase, que tiene como característica que no devuelve ningún valor. El constructor tiene el mismo nombre de la clase y se usa para ejecutar las acciones necesarias cuando se instancia un objeto de la clase. Cuando haya que destruir el objeto se podrá utilizar una función para ejecutar las operaciones necesarias cuando el objeto deje de existir, que dependerán del lenguaje que se utilice.

## 3.4.- Relaciones entre clases.

Una **relación** es una conexión entre dos clases que incluimos en el diagrama cuando aparece algún tipo de relación entre ellas en el dominio del problema.
Se representan como una línea continua.

Los mensajes "navegan" por las relaciones entre clases, es decir, los mensajes se envían entre objetos de clases relacionadas, normalmente en ambas direcciones, aunque a veces la definición del problema hace necesario que se navegue en una sola dirección, entonces la línea finaliza en punta de flecha.

Las relaciones se caracterizan por su **cardinalidad**, que representa cuantos objetos de una clase se pueden involucrar en la relación, y pueden ser:

- De herencia.
- De composición.
- De agregación.

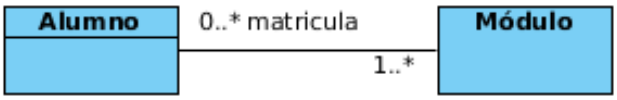
### 3.4.1.- Cardinalidad o multiplicidad de la relación.

Un concepto muy importante es la **cardinalidad de una relación**, representa cuantos objetos de una clase se van a relacionar con objetos de otra clase.

En una relación hay dos cardinalidades, una para cada extremo de la relación y pueden tener los siguientes valores:

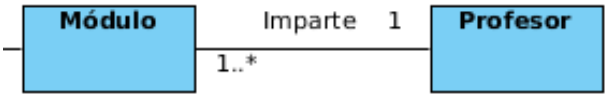
| Significado de las cardinalidades. | |
|---|---|
| **Cardinalidad** | **Significado** |
| **1** | Uno y sólo uno |
| **0..1** | Cero o uno |
| **N..M** | Desde N hasta M |
| ***** | Cero o varios |
| **0..*** | Cero o varios |
| **1..*** | Uno o varios (al menos uno) |

Por ejemplo, si tengo la siguiente relación:




Quiere decir que los alumnos se matriculan en los módulos, en concreto, que un alumno se puede matricular en uno a más módulos y que un módulo puede tener ningún alumno, uno o varios.

O esta otra:




En la que un profesor puede impartir uno o varios módulos, mientras que un módulo es impartido sólo por un profesor.

### 3.4.2.- Relación de herencia

La **herencia** es una propiedad que permite a los objetos ser construidos a partir de otros objetos, es decir, la capacidad de un objeto para utilizar estructuras de datos y métodos presentes en sus antepasados.

El objetivo principal de la herencia es la ***reutilización*****,** poder utilizar código desarrollado con anterioridad. La herencia supone una **clase base** y una jerarquía de clases que contiene las **clases derivadas**. Las clases derivadas pueden heredar el código y los datos de su clase base, añadiendo su propio código especial y datos, incluso cambiar aquellos elementos de la clase base que necesitan ser diferentes, es por esto que los atributos, métodos y relaciones de una clase se muestran en el nivel más alto de la jerarquía en el que son aplicables.

**Tipos**:

- **Herencia simple**: Una clase puede tener sólo un ascendente. Es decir una subclase puede heredar datos y métodos de una única clase base.
- **Herencia múltiple**: Una clase puede tener más de un ascendente inmediato, adquirir datos y métodos de más de una clase.

**Representación**:

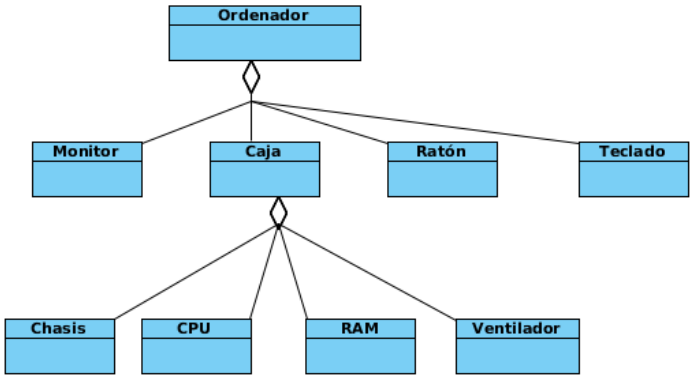
En el diagrama de clases se representa como una asociación en la que el extremo de la clase base tiene un triángulo.

### 3.4.3.- Agregación y composición.

Muchas veces una determinada entidad existe como un conjunto de otras entidades. En este tipo de relaciones un objeto componente se integra en un objeto compuesto. La orientación a objetos recoge este tipo de relaciones como dos conceptos: la agregación y la composición.

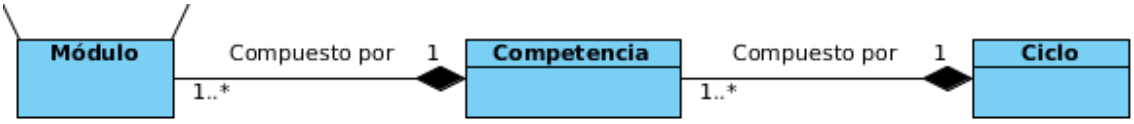
La **agregación** es una asociación binaria que representa una relación todo-parte (pertenece a, tiene un, es parte de). Los elementos parte pueden existir sin el elemento contenedor y no son propiedad suya. Por ejemplo, un centro comercial tiene clientes o un equipo tiene unos miembros. El tiempo de vida de los objetos no tiene porqué coincidir.

En el siguiente caso, tenemos un ordenador que se compone de piezas sueltas que pueden ser sustituidas y que tienen entidad por sí mismas, por lo que se representa mediante relaciones de agregación. Utilizamos la agregación porque es posible que una caja, ratón o teclado o una memoria RAM existan con independencia de que pertenezcan a un ordenador o no.




La **composición** es una agregación fuerte en la que una instancia '**parte**' está relacionada, como máximo, con una instancia **'todo**' en un momento dado, de forma que cuando un objeto 'todo' es eliminado, también son eliminados sus objetos 'parte'. Por ejemplo: un rectángulo tiene cuatro vértices, un centro comercial está organizado mediante un conjunto de secciones de venta...

Para modelar la estructura de un ciclo formativo vamos a usar las clases Módulo, Competencia y Ciclo que representan lo que se puede estudiar en Formación Profesional y su estructura lógica. Un ciclo formativo se compone de una serie de competencias que se le acreditan cuando supera uno o varios módulos formativos.




Dado que si eliminamos el ciclo las competencias no tienen sentido, y lo mismo ocurre con los módulos hemos usado relaciones de composición. Si los módulos o competencias pudieran seguir existiendo sin su contenedor habríamos utilizado relaciones de agregación.

Estas relaciones se representan con un rombo en el extremo de la entidad contenedora. En el caso de la agregación es de color blanco y para la composición negro. Como en toda relación hay que indicar la cardinalidad.

#### 3.4.4.- Atributos de enlace

Es posible que tengamos alguna relación en la que sea necesario añadir algún tipo de información que la complete de alguna manera. Cuando esto ocurre podemos añadir atributos a la relación.

### 3.5.- Paso de los requisitos de un sistema al diagrama de clases.

Empezamos identificando objetos que serán las clases del diagrama examinando el planteamiento del problema. Los objetos se determinan subrayando cada nombre o cláusula nominal e introduciéndola en una tabla simple. Los sinónimos deben destacarse. Pero, ¿qué debemos buscar una vez que se han aislado todos los nombres? Buscamos sustantivos que puedan corresponder con las siguientes categorías:

- **Entidades externas** (por ejemplo: otros sistemas, dispositivos, personas) que producen o consumen información a usar por un sistema computacional.

- **Cosas** (por ejemplo: informes, presentaciones, cartas, señales) que son parte del dominio de información del problema.

- **Ocurrencias o sucesos** (por ejemplo: una transferencia de propiedad o la terminación de una serie de movimientos en un robot) que ocurren dentro del contexto de una operación del sistema.

- **Papeles o roles** (por ejemplo: director, ingeniero, vendedor) desempeñados por personas que interactúan con el sistema.

- **Unidades organizacionales** (por ejemplo: división, grupo, equipo) que son relevantes en una aplicación.

- **Lugares** (por ejemplo: planta de producción o muelle de carga) que establecen el contexto del problema y la función general del sistema.

- **Estructuras** (por ejemplo: sensores, vehículos de cuatro ruedas o computadoras) que definen una clase de objetos o, en casos extremos, clases relacionadas de objetos.

Cuando estemos realizando este proceso debemos estar pendientes de no incluir en la lista cosas que no sean objetos, como operaciones aplicadas a otro objeto, por ejemplo, "inversión de imagen" producirá un objeto en el ámbito del problema, pero en la implementación dará origen a un método. También es posible detectar dentro de los sustantivos **atributos** de objetos, cosa que también indicaremos en la tabla.
Cuando tengamos la lista completa habrá que estudiar cada objeto potencial para ver si, finalmente, es incluido en el diagrama.
Para ayudarnos a decidir podemos utilizar los siguientes criterios:

1. La información del objeto es necesaria para que el sistema funcione.
2. El objeto posee un **conjunto de atributos** que podemos encontrar en cualquier ocurrencia del objeto. Si sólo aparece un atributo normalmente se rechazará y será añadido como atributo de otro objeto.
3. El objeto tiene un **conjunto de operaciones** identificables que pueden cambiar el valor de sus atributos y son comunes a cualquier ocurrencia del objeto.
4. Es una entidad externa que consume o produce información esencial para la producción de cualquier solución en el sistema.

El objeto se incluye si cumple todos (o casi todos) los criterios.

Se debe tener en cuenta que la lista no incluye todo, habrá que añadir objetos adicionales para completar el modelo y también, que diferentes descripciones del problema pueden provocar la toma de diferentes decisiones de creación de objetos y atributos.

### 3.5.1.- Obtención de atributos y operaciones.

Definen al objeto en el contexto del sistema, es decir, el mismo objeto en sistemas diferentes tendría diferentes atributos, por lo que debemos buscar en el enunciado o en nuestro propio conocimiento, características que tengan sentido para el objeto en el contexto que se analiza. Deben contestar a la pregunta "*¿Qué elementos (compuestos y/o simples) definen completamente al objeto en el contexto del problema actual?*"

- **Operaciones**

  Describen el comportamiento del objeto y modifican sus características de alguna de estas formas:

  - Manipulan los datos.

- Realizan algún cálculo.
- Monitorizan un objeto frente a la ocurrencia de un suceso de control.

Se obtienen analizando verbos en el enunciado del problema.

- **Relaciones**

  Por último habrá que estudiar de nuevo el enunciado para obtener cómo los objetos que finalmente hemos descrito se relacionan entre sí. Para facilitar el trabajo podemos buscar mensajes que se pasen entre objetos y las relaciones de composición y agregación. Las relaciones de herencia se suelen encontrar al comparar objetos semejantes entre sí, y constatar que tengan atributos y métodos comunes.

  Cuando se ha realizado este procedimiento no está todo el trabajo hecho, es necesario revisar el diagrama obtenido y ver si todo cumple con las especificaciones. No obstante siempre se puede refinar el diagrama completando aspectos del ámbito del problema que no aparezcan en la descripción recurriendo a entrevistas con los clientes o a nuestros conocimientos de la materia.

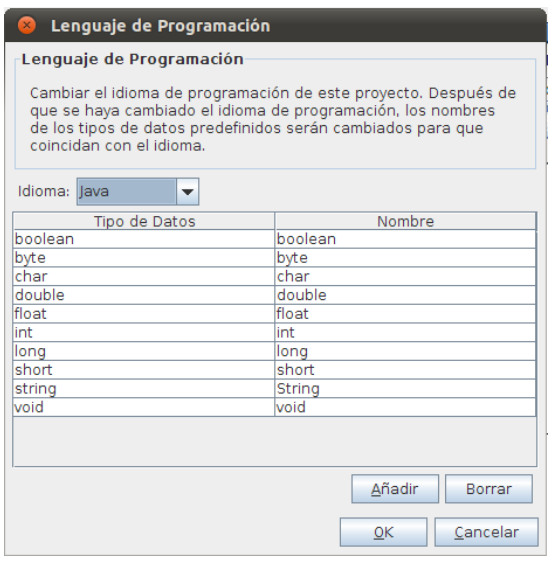
### 3.6.- Generación de código a partir del diagrama de clases.

La Generación Automática de Código consiste en la creación utilizando herramientas CASE de código fuente de manera automatizada. El proceso pasa por establecer una correspondencia entre los elementos formales de los diagramas y las estructuras de un lenguaje de programación concreto. El diagrama de clases es un buen punto de partida porque permite una traducción bastante directa de las clases representadas gráficamente, a clases escritas en un lenguaje de programación específico como Java o C++.

Normalmente las herramientas de generación de diagramas UML incluyen la facilidad de la generación, o actualización automática de código fuente, a partir de los diagramas creados.

#### 3.6.1.- Elección del lenguaje de programación. Orientaciones para el lenguaje java.

El lenguaje final de implementación de la aplicación influye en algunas decisiones a tomar cuando estamos creando el diagrama ya que el proceso de traducción es inmediato. Si existe algún problema en los nombres de clases, atributos o tipos de datos porque no puedan ser



utilizados en el lenguaje final o no existan la generación dará un fallo y no se realizará.
Por ejemplo, si queremos utilizar la herramienta de generación de código tendremos que asegurarnos de utilizar tipos de datos simples apropiados, es decir, si usamos Java el tipo de dato para las cadenas de caracteres será String en lugar de string o char*.

**3.7.- Generación de la documentación.**

Como en todos los diagramas UML, podemos hacer las anotaciones que consideremos necesarias abriendo la especificación de cualquiera de los elementos, clases o relaciones, o bien del diagrama en sí mismo en la pestaña "Specification".
La ventana del editor cuenta con herramientas para formatear el texto y darle un aspecto bastante profesional, pudiendo añadir elementos como imágenes o hiperenlaces.
También se puede grabar un archivo de voz con la documentación del elemento usando el icono Grabar.

- **Generar informes**

  Cuando los modelos están completos podemos generar un informe en varios formatos diferentes (HTML, PDF o Word) con la documentación que hemos escrito.
  El resultado es un archivo (.html, .pdf o .doc) en el directorio de salida que hayamos indicado con la documentación de los diagramas seleccionados.

## 4.- Ingeniería Inversa.

La **ingeniería inversa** se define como el proceso de analizar código, documentación y comportamiento de una aplicación para identificar sus componentes actuales y sus dependencias y para extraer y crear una abstracción del sistema e información del diseño.
El sistema en estudio no es alterado, sino que se produce un conocimiento adicional del mismo.

Tiene como caso particular la reingeniería que es el proceso de extraer el código fuente de un archivo ejecutable.

La ingeniería inversa puede ser de varios tipos:

- **Ingeniería inversa de datos**: Se aplica sobre algún código de bases datos (aplicación, código SQL, etc.) para obtener los modelos relacionales o sobre el modelo relacional para obtener el diagrama entidad-relación.
- **Ingeniería inversa de lógica o de proceso**: Cuando la ingeniería inversa se aplica sobre el código de un programa para averiguar su lógica (reingeniería), o sobre cualquier documento de diseño para obtener documentos de análisis o de requisitos.

- **Ingeniería inversa de interfaces de usuario**: Se aplica con objeto de mantener la lógica interna del programa para obtener los modelos y especificaciones que sirvieron de base para la construcción de la misma, con objeto de tomarlas como punto de partida en procesos de ingeniería directa que permitan modificar dicha interfaz.

**GLOSARIO**

**Abstracción:**

Aislar un elemento de su contexto o del resto de elementos que le acompañan para disponer de ciertas características que necesitamos excluyendo las no pertinentes. Con ello capturamos algo en común entre las diferentes instancias con objeto de controlar la complejidad del software.

**Arquitectura del sistema:**

Conjunto de decisiones significativas acerca de la organización de un sistema software, la selección de los elementos estructurales a partir de los cuales se compone el sistema, y las interfaces entre ellos. Junto con su comportamiento, tal y como se especifica en las colaboraciones entre esos elementos, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente mayores y el estilo arquitectónico que guía esta organización, estos elementos y sus interfaces, sus colaboraciones y su composición.

**Artefacto:**

Un artefacto es una información que es utilizada o producida mediante un proceso de desarrollo de software. Pueden ser artefactos un modelo, una descripción o un software.

**Cardinalidad:**

Número de elementos de un conjunto.

**CASE** (Computer Aid Software Engineering)

Ingeniería del Software Asistida por Ordenador. Hace referencia a una serie de herramientas software para la ayuda al desarrollo del software, en todos los aspectos del ciclo de vida del software.

**Clase base:**

Cuando se utiliza el concepto de herencia es la clase que contiene los métodos y atributos que van a ser heredados por la clase derivada.

**Clase derivada:**

Cuando se utiliza la herencia es la clase que hereda los atributos y métodos de la clase base.

**Cohesión:**

En informática, la cohesión hace referencia a la forma en que agrupamos unidades de software (módulos, subrutinas...) en una unidad mayor. Por ejemplo: la forma en que se agrupan funciones en una biblioteca de funciones o la forma en que se agrupan métodos en una clase, etc. En orientación a objetos logramos una alta cohesión cuando una clase hace referencia a una única entidad, el objetivo es lograr el principio de única responsabilidad por el que una clase se dedica a una responsabilidad únicamente y, a su vez, esa responsabilidad está cubierta por una única clase.

**Dominio del problema:**

Se define como dominio a un área de conocimiento o actividad caracterizada por un conjunto de conceptos y terminología comprendida por los practicantes de ese dominio. El dominio del problema es aquel sobre el que se define el problema a resolver por el sistema que se va a generar.

**Encapsulación:**

Reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción, permite mejorar la cohesión y se implementa a través del principio de ocultación.

**Escalabilidad:**

Propiedad deseable de un sistema, red o proceso que le permite hacerse más grande sin rehacer su diseño y sin disminuir su rendimiento.

**Estructura de datos:**

Es el conjunto de una colección de datos y de las funciones que modifican esos datos, que recrean una entidad con sentido, en el contexto de un problema.

**Extensibilidad:**

Principio de diseño en el desarrollo de sistemas informáticos que tiene en cuenta el futuro crecimiento del sistema. Mide la capacidad de extender un sistema y el esfuerzo necesario para conseguirlo.

**Función:**

Conjunto de sentencias escritas en un lenguaje de programación que operan sobre un conjunto de parámetros y producen un resultado.

**Grafo:**

Deriva del griego, significa trazar. Objeto combinatorio formado por un conjunto de nodos y un subconjunto de líneas seleccionadas del conjunto de líneas que unen cada par de nodos entre sí, denominadas arcos. Cuando dos nodos del grafo están unidos por un arco se dice que existe una relación entre los nodos.

**Herencia:**

Propiedad del paradigma de orientación a objetos que permite que un objeto sea construido a partir de otro, reutilizando sus propiedades y métodos y aportando los suyos propios. Puede ser simple o compuesta según se herede de un único objeto o de varios.

**Ingeniería directa:**

En el contexto del desarrollo de software, la transformación de un modelo en código a través de su traducción a un determinado lenguaje de programación.

**Ingeniería inversa:**

En el contexto del desarrollo de software, la transformación del código en un modelo a través de su traducción desde un determinado lenguaje de programación.

**Instancia:**

Objeto de una clase, creado en tiempo de ejecución con un estado concreto.

**Modelo:**

Representación gráfica o esquemática de una realidad, sirve para organizar y comunicar de forma clara los elementos que involucran un todo. Esquema teórico de un sistema o de una realidad compleja que se elabora para facilitar su comprensión y el estudio de su comportamiento.

**Multiplataforma:**

Referido a un software que puede ejecutarse en diferentes plataformas. Por ejemplo una aplicación que pueda ejecutarse en Windows, Linux, Mac OS-X y Power PC. Se entiende por plataforma la combinación de software y hardware que se usa para ejecutar aplicaciones, puede estar formado por un sistema operativo, una arquitectura o ambos.

**Ocultación:**

Relacionado con la ingeniería del software, consiste en el aislamiento del estado, es decir, de los datos miembro de un objeto, de manera que sólo se puede cambiar mediante las operaciones definidas para ese objeto. Este aislamiento protege a los datos de que sean modificados por alguien que no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

**Paquete:**

Mecanismo de propósito general para organizar elementos en grupos.

**Paradigma:**

Modelo o patrón aplicado a cualquier disciplina científica u otro contexto epistemológico.

**Polimorfismo:**

La palabra proviene del griego y significa varias formas. En el paradigma de orientación a objetos hace referencia al comportamiento diferente que adopta un objeto según el tipo que tenga, o de un método según sean sus parámetros.

**Programación estructurada:**

Paradigma de programación que postula que una programa informático no es más que una sucesión de llamadas a funciones, bien sean del sistema o definidas por el usuario.

**Requisito:**

Condiciones que debe cumplir un proyecto software. Suelen venir definidos por el cliente. Permiten definir los objetivos que debe cumplir un proyecto software.

**Reutilización:**

Utilizar artefactos existentes durante la construcción de nuevo software. Esto aporta calidad y seguridad al proyecto, ya que el código reutilizado ya ha sido probado.

**Wysiwyg:**

Siglas de **"What You See Is What You Get".** En español **"Lo que ves es lo que obtienes"**.

Hace referencia a aplicaciones para el desarrollo de proyectos con componentes gráficos que se crean seleccionando los componentes de una paleta y "dibujándolos" sobre un lienzo que forma la base del proyecto gráfico.