Ingeniería de Software II

UML – Leguaje Unificado de Modelado

Universidad Autónoma de Entre Ríos
Facultad de Ciencia y Tecnología

Lic. Paolo Orundés Cardinali
orundescardinali.paolo@uader.edu.ar

El Lenguaje Unificado de Modelado

Temario

- Historia
- ¿Qué es UML? ¿Por qué usar UML?
- Conceptos básicos
- Diagrama de Clases
- Diagrama de Casos de Uso
- Notas
- Diagrama de Objetos
- Diagrama de Paquetes
- Diagrama de Estados
- Diagrama de Secuencia
- Diagrama de Actividad
- Diagrama de Interacción en Visión General
- Diagrama de Comunicación
- Diagrama de Componentes
- Diagrama de Despliegue

Historia

UML es una notación que se originó como resultado de la unificación de la técnica de modelado de objetos (OMT por sus siglas en inglés [Rumbaugh et al., 1991], Booch [Booch 1941] e ingeniería de software orientada a objetos (OOSE, por sus siglas en inglés [Jacobson et al., 1992]).

El UML también está influido por otras notaciones orientadas a objetos, como las presentadas por Shlaer/Mellor [Mellor y Shlaer, 1998], Coad/Yourdon [Coad el al.,1995], Wirfs-Brock [Wirfs-Brock et al., 1990] y Martin/Odell [Martin y Odell, 1992].

UML ha sido diseñado para un amplio rango de aplicaciones. Por lo tanto, proporciona construcciones para un amplio rango de sistemas y actividades (sistemas en tiempo real, sistemas distribuidos, análisis, diseño del sistema, entregas). El desarrollo de sistemas se enfoca en tres modelos diferentes del sistema:

- **El modelo funcional**, representado en UML, con diagramas de caso de uso, describe la funcionalidad del sistema desde el punto de vista del usuario.
- **El modelo de objetos**, representado en UML con diagramas de clase, describe la estructura de un sistema desde el punto de vista de objetos, atributos, asociaciones y operaciones.
- **El modelo dinámico**, representado en UML con diagramas de secuencia, diagramas de estado y diagramas de actividad, describen el comportamiento interno del sistema. Los diagramas de secuencia describen el comportamiento como una secuencia de mensajes intercambiados entre un conjunto de objetos, mientras que los diagramas de gráfica de estado describen el comportamiento desde el punto de vista de estados de un objeto individual y las transiciones posibles entre estados.

En resumen:

Cuando comienza POO no existía una metodología propia para trabajar con dicho paradigma. Por lo que se empezaron a trabajar con las metodologías existentes, ellas eran

- Desciende de tres metodologías que se utilizan
    - Rumbaugh – Objet Modeling Technique
    - Jacobson – Objetc-Oriented Software Enigneerieng
    - Booch – Booch Method
- Una empresa llamada Rational Software combinó las tres metodologías en un proceso largo.
- Existen varias versiones
- UML es un lenguaje que puede usarse con diferentes metodologías pero no es una metodología en sí mismo.

¿Qué es UML? ¿Por qué usar UML?

UML es un lenguaje estándar para escribir proyectos de software. Puede utilizarse para visualizar, especificar, construir y documentar los componentes de un sistema que involucra una gran cantidad de software.

UML es apropiado para modelar desde sistemas de información de empresas hasta aplicaciones distribuidas basadas en la web, e incluso para sistemas empotrados de tiempo real, muy exigentes y críticos. Es un lenguaje muy expresivo, que cubre todas las vistas necesarias para desarrollar y luego implementar tales sistemas.

UML es sólo un lenguaje y por tanto sólo una parte de un método de desarrollo de software. UML es independiente del proceso, aunque para utilizarlo óptimamente se debería usar en un proceso que fuese dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

En general el UML es un lenguaje para visualizar, especificar, construir y documentar los componentes de un sistema con gran cantidad de *software*.

Como lenguaje, el UML proporciona un vocabulario y las reglas para combinarlas palabras de ese vocabulario con el objetivo de posibilitar la comunicación. Un lenguaje de modelado es un lenguaje cuyo vocabulario y reglas se centran en la representación conceptual y física del sistema. Un lenguaje de modelado como UML es, por tanto, un lenguaje estándar para los proyectos *software*.

El modelado proporciona una comprensión de un sistema. Nunca es suficiente un único modelo. Más bien, para comprender cualquier cosa, a menudo se necesitan múltiples modelos conectados entre sí, excepto en los casos más triviales. Para sistemas con gran cantidad de *software*, se requiere un lenguaje que cubra las diferentes vistas de la arquitectura de un sistema mientras evoluciona a través del ciclo de vida del desarrollo del *software*.

El vocabulario y las reglas de un lenguaje como el UML indican cómo crear y leer modelos bien formados, pero no dicen que modelos se deben crear ni cuando se deberían crear. Esta es la tarea del proceso del desarrollo del *software*. Un proceso bien definido guiará a su usuario al decidir que componentes producir, qué actividades y personal se emplea para crearlos y gestionarlos, y cómo usar estos componentes para medir y controlar el proyecto de forma global.

Para muchos programadores, la distancia entre pensar en una implementación y transformarla en código es casi nula. Lo piensas, lo codificas. De hecho algunas cosas se modelan mejor directamente en código. En estos casos, el **programador todavía está haciendo mentalmente algo de modelado**, si bien lo hace de forma completamente mental. No obstante, esta manera de proceder plantea algunos problemas:

- Primero, la comunicación de estos modelos conceptuales se torna complicada y está sujeta a errores, salvo que las dos partes hablen el mismo lenguaje. Normalmente los proyectos y las organizaciones desarrollan su propio lenguaje, y es difícil comprender lo que está pasando para alguien ajeno al grupo.
- Segundo, hay algunas cuestiones en un sistema de *software* que no se pueden entender a menos que se construyan sobre modelos que trasciendan el lenguaje de programación textual. Por ejemplo el significado de una jerarquía de clases puede inferirse, pero no capturarse completamente inspeccionando el código de todas las clases en la jerarquía.
- Tercero, si el desarrollador que escribió el código no dejó documentación escrita de los modelos que hacía en su cabeza, esa información se perderá para siempre, o como

mucho, será sólo parcialmente reproducible a partir de la implementación una vez que el desarrollador se haya marchado.

Al escribir modelos en UML se afronta el tercer problema: **un modelo explícito facilita la comunicación**.

Algunas cosas se modelan mejor textualmente; otras se modelan mejor de forma gráfica. En realidad, en todos los sistemas interesantes hay estructuras que trascienden lo que puede ser representado mediante un lenguaje de comunicación. UML es uno de estos lenguajes gráficos. Así afronta el segundo problema mencionado anteriormente.

UML es algo más que un simple montón de símbolos gráficos. Más bien, detrás de cada símbolo en la notación UML hay una semántica bien definida. De manera que un desarrollador puede escribir un modelo en UML y otro desarrollador, o incluso otra herramienta, puede interpretar ese modelo sin ambigüedad. Así afronta el primer problema mencionado anteriormente.

En este contexto, especificar significa construir modelos precisos, no ambiguos y completos. En particular, UML cubre la especificación de todas las decisiones de análisis, diseño e implementación que deben realizarse al desarrollar y desplegar un sistema con gran cantidad de *software*.

UML no es un lenguaje de programación visual, pero sus modelos pueden conectarse a gran variedad de lenguajes de programación. Esto significa que es posible establecer correspondencias desde un modelo UML a un lenguaje de programación como C++, Java o Visual Basic, o incluso tablas en una base de datos relacional o al almacenamiento persistente en una base de datos orientada a objetos.

Las cosas que se expresan mejor gráficamente también se representan gráficamente en UML, mientras que las cosas que se representan mejor textualmente se plasman con el lenguaje de programación.

Esta correspondencia permite la ingeniería directa: la generación de código a partir de un modelo UML en un lenguaje de programación. Lo contrario también es posible: se puede reconstruir un modelo en UML a partir de una implementación. Pero este proceso no es automático, a menos que se codifique esa información en la implementación, la información se pierde cuando se pasan de los modelos al código. La ingeniería inversa requiere, por lo tanto, herramientas que la soporten e intervención humana. La combinación de estas dos vías de generación de código y de ingeniería inversa produce una ingeniería “de ida y vuelta”, entendiendo por eso la posibilidad de trabajar en una vista gráfica o textual, mientras que las herramientas mantienen la consistencia de las dos vistas.

Además de esta correspondencia directa, UML es lo suficientemente expresivo y no ambiguo como para permitir la ejecución directa de modelos, la simulación de sistemas y la instrumentación de sistemas en ejecución.

Hoy en día, una organización de software produce toda clase de componentes además de código ejecutable. Estos componentes incluyen (sin limitarse a ello):-

- Requisitos.

- Arquitectura.
- Diseño.
- Código fuente.
- Planificación de proyectos.
- Pruebas.
- Prototipos.
- Versiones.

Dependiendo de la cultura de desarrollo, algunos de estos componentes se tratan más o menos formalmente que otros. Tales componentes no son sólo los entregables de un proyecto, también son críticos en el control, la medición y comunicación que requiere un sistema durante su desarrollo y después de su implementación.

UML cubre la documentación de la arquitectura de un sistema y todos sus detalles. UML también proporciona un lenguaje para expresar requisitos y pruebas. Finalmente, UML proporciona un lenguaje para modelar las actividades de planificación de proyectos y gestión de versiones.

En resumen

- UML es el Lenguaje Unificado de Modelado (*Unified Modeling Language*)
    - Es un lneguaje visual, usamos diagramas para mostrar lo que deseamos.
- Es un lenguaje gráfico capaz de expresar
    - Requisitos de Software
    - Arquitectura del Software
    - Diseño del Software
- Que sirve para
    - Comunicarse entre desarrolladores
    - Comunicarse con los clientes
    - Usar herramientas de generación automática de código
- http://www.uml.org

- OMG
    - Object Managment Group
    - Es un consorcio internacional de estandarización de tecnologías orientadas a objetos
    - http://www.omg.org/

- Entre otras cosas estandariza el estándar UML

Consiste en un conjunto integrado de diagramas definidos para ayudar a los desarrolladores de software y de sistemas a realizar las tareas de análisis y diseño:

- Especificación
- Visualización
- Diseño Arquitectónico
- Construcción
- Simulación y pruebas
- Documentación

Características generales

- Extensible
- Flexible
- Escalable

Conceptos básicos

Modelo

Un modelo UML es una representación abstracta de un sistema que se construye con diagramas UML y documentación adicional. UML es el acrónimo de Lenguaje Unificado de Modelado.

Los modelos UML se utilizan para visualizar la arquitectura, diseño e implementación de sistemas de software complejos.

Tienen las siguientes características:

- Un patrón sobre el cual algo que se producirá está basado
- Un diseño o tipo
- Modelar es construir un plan basándonos en un patrón
- En UML lo podemos considerar como una forma visual de describir un negocio y sus reglas
- Algunos creen que modelar no tiene ningún valor, pero
    - Nos ayuda a comunicar a diseños
    - Clasifica problemas complejos
    - Ayuda a los diseños sean cercanos a las implementaciones de la realidad
    - Ahorra tiempo y dinero al permitir trabajar más eficientemente
    - Ayuda a definir y entender los objetivos
    - Comprendemos el negocio y sus procesos
- No hay que exagerar el análisis al punto que ya no se es productivo: “Parálisis de análisis”

Diagrama

Un diagrama UML es una forma de visualizar sistemas y software utilizando el Lenguaje Unificado de Modelado (UML), es decir:

- Es una visualización de diferentes elementos de modelado descritos en UML
- Cada diagrama se usa para un propósito específico
- Cada diagrama tiene símbolos especiales para lograr ese propósito
- Es la representación de un proceso o un sistema

Composición de los diagramas

Los diagramas en UML tienen una composición en dos grandes partes:

- Notación
  - Elementos que trabajan entre sí adentro de un diagrama
  - Conectores, símbolos.
- Object Management Group
  - Se encarga de hacer las especificaciones del lenguaje
    - Diagramas
    - Objetos

Abstracción

La abstracción en UML es el proceso de simplificar y generalizar detalles complejos para centrarse en las características esenciales de un sistema. Es un principio clave de la ingeniería de software, es decir:

- La técnica de hacer un modelo de tus ideas del mundo es el uso de la **abstracción**: Por ejemplo, un mapa es un modelo del mundo, no el mundo en miniatura
- En los diagramas UML se muestra una abstracción del sistema, no todo el sistema, con el objetivo de que sea fácil de entender

Puntos de vista

Los puntos de vista en UML son las diferentes vistas del sistema que admite este lenguaje de modelado unificado. Cada vista proporciona una perspectiva única del sistema.

Ejemplos de vistas de UML: vista del usuario, vista estructural, vista de comportamiento, vista de implementación, vista de interacción de rastreo

En resumen:

- UML permite crear diagramas que reflejan diferentes **puntos de vista** del mismo sistema: Por ejemplo, hay mapas físicos, mapas políticos, mapas históricos (todos sobre el mismo mundo)
- Esto permite mostrar ciertos aspectos y ocultar otros para que sean más fáciles de comprender

Tipos de diagramas

Los tipos de diagramas es una clasificación según las funciones que representan del modelo. Hay dos grandes tipos de diagramas

- **Diagramas Estructurales:** Muestran los elementos de construcción del sistema. Características que no cambian con el tiempo
- **Diagramas de Comportamiento:** Muestra como el sistema responde a las peticiones o evoluciona con el tiempo.
    - **Diagramas de Interacción:** Engloba a ciertos diagramas de comportamiento que muestran el intercambio de mensajes dentro de un grupo de objetos que cooperan (colaboración) para obtener un objetivo

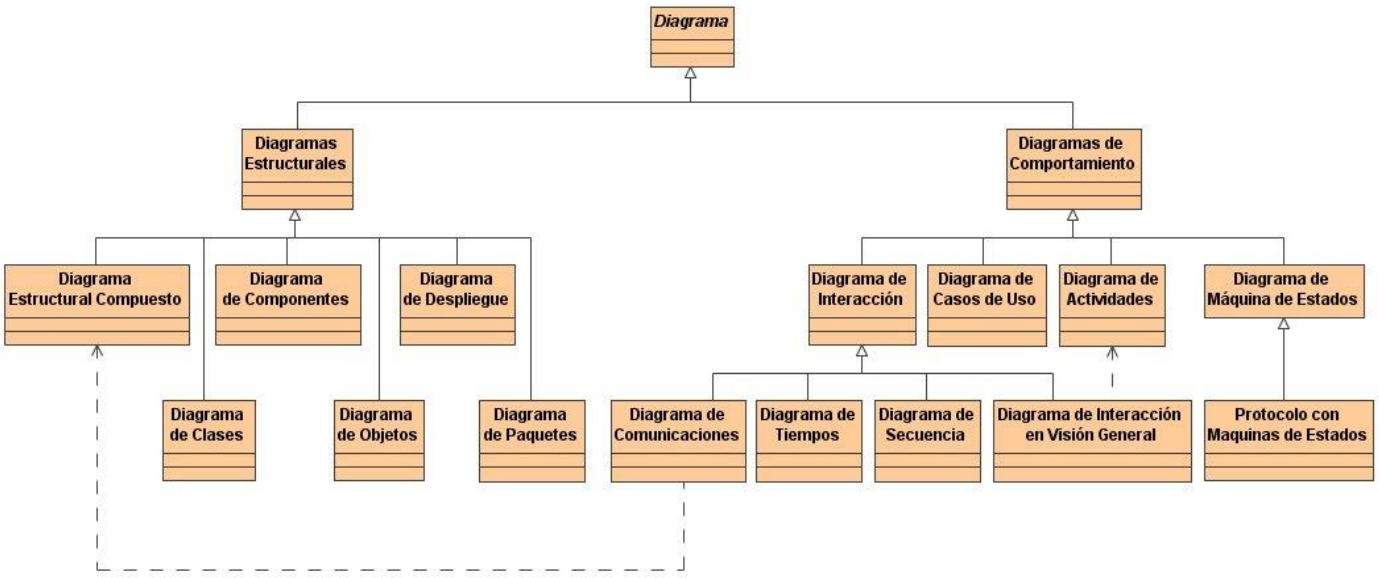


Diagrama de clases

- Entendiendo el diagrama de clases
- Clases
- Relaciones
- Clase abstracta
- Objeto
- Atributos
- Operaciones
- Visibilidad
- Estereotipos
- Asociación
- Multiplicidad
- Asociación reflexiva
- Clase asociada
- Agregación
- Composición
- Generalización
- Realización
- Dependencia
- Notas
- Restricciones
- Interfaz proveedor
- Interfaz requerida
- Template

El objetivo principal de este modelo es la representación de los aspectos estáticos del sistema, utilizando diversos mecanismos de abstracción (clasificación, generalización, agregación).

El diagrama de clases recoge las clases existentes y sus correspondientes asociaciones. En este diagrama se representa la estructura y el comportamiento de cada uno de los objetos del sistema y sus relaciones con los demás objetos, pero no muestra ninguna información temporal.

Con el fin de facilitar la comprensión del diagrama, se pueden incluir paquetes como elementos del mismo, donde cada uno de ellos agrupa un conjunto de clases que tienen algún tipo de relación. Este diagrama no refleja los comportamientos temporales de las clases, aunque para mostrarlos se puede utilizar un diagrama de transición de estados.

Los elementos básicos del diagrama son: **clases y relaciones**.

Clases

Una clase describe un conjunto de objetos con propiedades (atributos) similares y un comportamiento común. Los objetos son instancias de las clases.

No existe un procedimiento inmediato que permita localizar las clases del diagrama de clases. Éstas suelen corresponderse con sustantivos que hacen referencia al ámbito del sistema de información y que se encuentran en los documentos de las especificaciones de requisitos y los casos de uso.

Dentro de la estructura de una clase se definen los atributos, que representan los datos asociados a los objetos instanciados por esa clase, y las operaciones o métodos, que representan las funciones o procesos propios de los objetos de una clase, caracterizando a dichos objetos.

El diagrama de clases permite representar clases abstractas. Una *clase abstracta* es una clase que no puede existir en la realidad, pero que es útil conceptualmente para el diseño del modelo orientado a objetos. Las clases abstractas no son instanciables directamente sino en sus descendientes. Una clase abstracta suele ser situada en la jerarquía de clases en una posición que le permita ser un depósito de métodos y atributos para ser compartidos o heredados por las subclases de nivel inferior.

Las clases y en general todos los elementos de los diagramas, pueden estar clasificados de acuerdo a varios criterios, como por ejemplo su objetivo dentro de un programa. Esta clasificación adicional se expresa mediante un *estereotipo*.

Algunos de los autores de métodos orientados a objetos, establecen una clasificación de todos los objetos que pueden aparecer en un modelo. Los tipos son: Objeto entidad, objeto interfaz o límite y objeto de control:

- **Objetos entidad** (Entity Objects).
  - Representan información que tiene una identidad única y persiste a lo largo del tiempo.
  - Normalmente contienen datos y lógica de negocio.
  - Ejemplo: En un sistema de gestión de estudiantes, un objeto Estudiante con atributos como nombre, matrícula y fecha de nacimiento.

```
Clase Estudiante
    Atributos:
        nombre
        matrícula
        fechaNacimiento

    Método Constructor(nombre, matrícula, fechaNacimiento):
        this.nombre = nombre
        this.matrícula = matrícula
        this.fechaNacimiento = fechaNacimiento

```

- **Objetos límite o interfaz** (Boundary Objects).
  - Actúan como intermediarios entre el sistema y el usuario u otros sistemas externos.
  - Manejan la entrada y salida de datos.
  - Ejemplo: Una clase FormularioRegistro que se encarga de capturar los datos del estudiante a través de una interfaz web

```
Clase FormularioRegistro
    Método capturarDatos():
        // Simular entrada de datos desde una interfaz
        retornar nuevo Estudiante("Juan Pérez", "12345", "2000-05-15")

```

- **Objetos de control** (Control Objects).
  - Manejan la lógica de negocio y la coordinación del flujo de trabajo.
  - Actúan como intermediarios entre los objetos entidad y los objetos de límite.
  - Ejemplo: Una clase GestorEstudiantes que valida y guarda los datos del estudiante en una base de datos.

```
Clase GestorEstudiantes
    Método registrarEstudiante(estudiante):
        // Lógica para validar y guardar en base de datos
        Imprimir "Estudiante registrado: " + estudiante.nombre + ", Matrícula: " + estudiante.matrícula
```

Funcionando en conjunto

```
// Crear el formulario y capturar los datos del estudiante
formulario = nuevo FormularioRegistro()
estudiante = formulario.capturarDatos()

// Registrar al estudiante usando el objeto de control
gestor = nuevo GestorEstudiantes()
gestor.registrarEstudiante(estudiante)
```

Donde:

- Estudiante: Objeto entidad que almacena los datos del estudiante.
- FormularioRegistro: Objeto límite que captura los datos desde una interfaz.
- GestorEstudiantes: Objeto de control que coordina el proceso de registro.

En función de la herramienta empleada, también se puede añadir información adicional a las clases para mostrar otras propiedades de las mismas, como son las reglas de negocio, responsabilidades, manejo de eventos, excepciones...

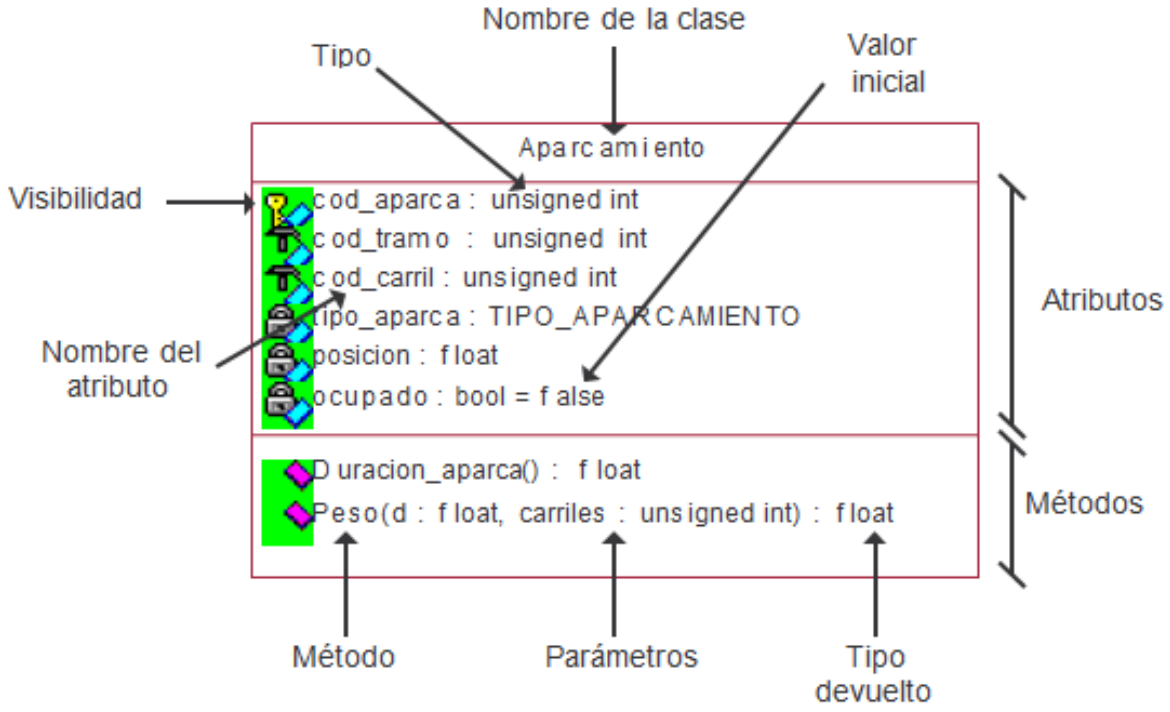
Dentro de las clases es importante destacar un caso especial, las *plantillas* o *templates*. Éstas son clases que están parametrizadas, es decir, que necesitan de un parámetro a la hora de formarse.

En resumen

- Uno de los diagramas más comunes
- Es un diagrama estático
- Usamos un clasificador que es una figura rectangular
- El clasificador para la clase colocamos el nombre, los atributos y las operaciones
- Como se relacionan los componentes, pero no “que y como” lo hacen

La notación de una clase se realiza de la siguiente manera

- El nombre de la clase aparece centrado en la parte superior
- En la parte central los atributos
- En la parte inferior los métodos
- Atención: los métodos abstractos se escriben en Itálica o Cursiva



Relaciones

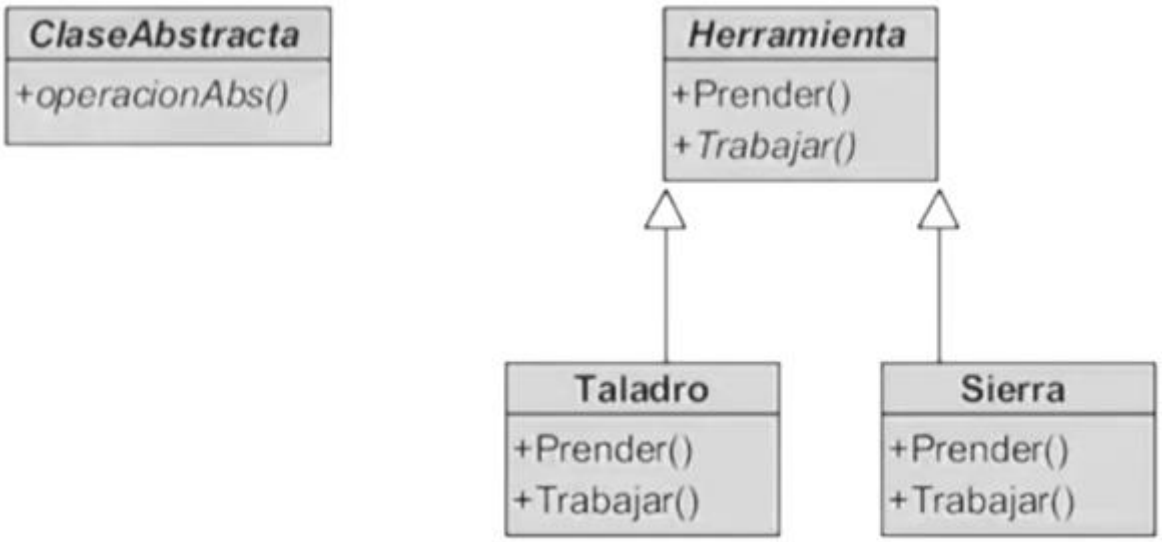
Los tipos más importantes de relaciones estáticas entre clases son los siguientes:

- **Asociación**. Las relaciones de asociación representan un conjunto de enlaces entre objetos o instancias de clases. Es el tipo de relación más general, y denota básicamente una dependencia semántica. Por ejemplo, un Carril pertenece a un Tramo. Cada asociación puede presentar elementos adicionales que doten de mayor detalle al tipo de relación:
  - **Rol**, o nombre de la asociación, que describe la semántica de la relación en el sentido indicado. Por ejemplo, la asociación entre Carril y Tramo recibe el nombre de *pertenece a*, como rol en ese sentido.
  - **Multiplicidad**, que describe la cardinalidad de la relación, es decir, especifica cuántas instancias de una clase están asociadas a una instancia de la otra clase. Los tipos de multiplicidad son: uno a uno, uno a muchos, muchos a muchos.
- **Herencia**. Las jerarquías de generalización / especialización se conocen como herencia. La herencia es el mecanismo que permite a una clase de objetos incorporar atributos y métodos de otra clase, añadiéndolos a los que ya posee. Con la herencia se refleja una relación *es-un* entre clases. La clase de la cual se hereda se denomina *superclase*, y la que hereda *subclase*.
- **Agregación**. La agregación es un tipo de relación jerárquica entre un objeto que representa la totalidad de ese objeto y las partes que lo componen. Permite el agrupamiento físico de estructuras relacionadas lógicamente. Los objetos *son-parte-de* otro objeto completo.
- **Composición**. La composición es una forma de agregación donde la relación de propiedad es más fuerte, e incluso coinciden los tiempos de vida del objeto completo y las partes que lo componen.
- **Dependencia**. Una relación de dependencia se utiliza entre dos clases o entre una clase y una interfaz, e indica que una clase requiere de otra para proporcionar alguno de sus servicios

Clase abstracta

Una **Clase Abstracta** es una clase que **no puede ser instanciada directamente**, sino que **sirve como base para otras clases derivadas** que heredan su comportamiento y atributos. Entonces:

- No puede ser instanciada
- Otras clases heredan de ella
- Se denota igual que las otras clases, pero con el nombre en itálica o cursiva
- Puede obtener operaciones abstractas



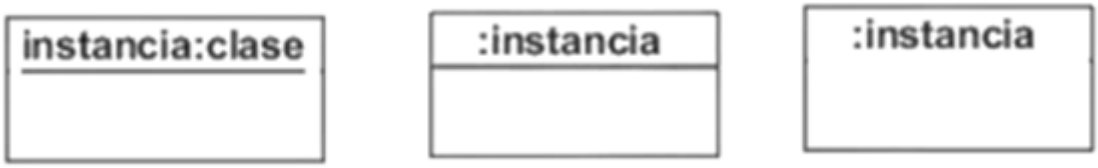
Características principales:

- Se utiliza para **definir atributos y métodos comunes** a múltiples clases.
- Puede contener **métodos abstractos (sin implementación)** que las **subclases están obligadas a implementar**.
- Permite la **herencia y el polimorfismo**.
- **No se pueden crear objetos** directamente a partir de una clase abstracta.

Objeto

Un **objeto** representa una **instancia concreta de una clase**, con **atributos específicos** que contienen **valores reales** y que pueden **ejecutar comportamientos definidos por la clase**.

- Es posible describir un objeto, aunque no es frecuente
- Se pueden colocar objetos anónimos



El segundo y el tercero son objetos anónimos, no son útiles desde el punto de vista de la programación, pero son válidos para UML

Diferencia entre Clase y Objeto

Clase	Objeto
Es una **plantilla o modelo** que define atributos y métodos.	Es una **instancia concreta** de una clase.
Define **propiedades genéricas**.	Contiene **valores específicos** para los atributos.
No ocupa memoria.	Ocupa espacio en memoria al ser creado.

Importancia del Objeto en UML:

- Permite **modelar el comportamiento del sistema en tiempo de ejecución**.
- Ayuda a **entender la relación entre instancias específicas** y cómo interactúan.
- Es clave en la **programación orientada a objetos (POO)**.

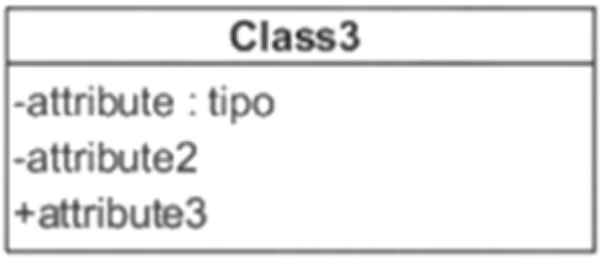
Visibilidad

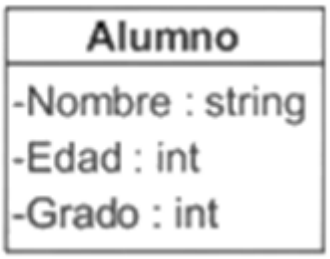
- **+ público:** Accesible desde cualquier clase
- **- privado:** Accesible solo desde la propia clase
- **# protegido**: Accesible desde la propia clase y las clases derivadas (los objetos que no formen parte de la cadena de herencia son privados, en cambio los que si formen parte de la cadena de herencia son públicos)
- **~ paquete:** Accesible desde clases dentro del mismo paquete (en algunos lenguajes como Java).

Atributos

Los **atributos** representan las **propiedades o características** de una clase. Son las **variables o datos** que describen el **estado interno de un objeto**.

- Se componen de la siguiente manera: Visibilidad nombre :tipo





Elementos clave de un atributo:

- Nombre: Identifica el atributo (por ejemplo, nombre, edad).
- Visibilidad: Define el nivel de acceso (+ público, - privado, # protegido, ~ paquete).
- Tipo de dato: Especifica el tipo de valor que almacena (por ejemplo, String, int, float).
- Valor por defecto (opcional): Valor inicial que tendrá el atributo.
- Multiplicidad (opcional): Indica si el atributo puede contener uno o varios valores (por ejemplo, 0..1, 1..*).
- Modificadores (opcional): Como static o final.

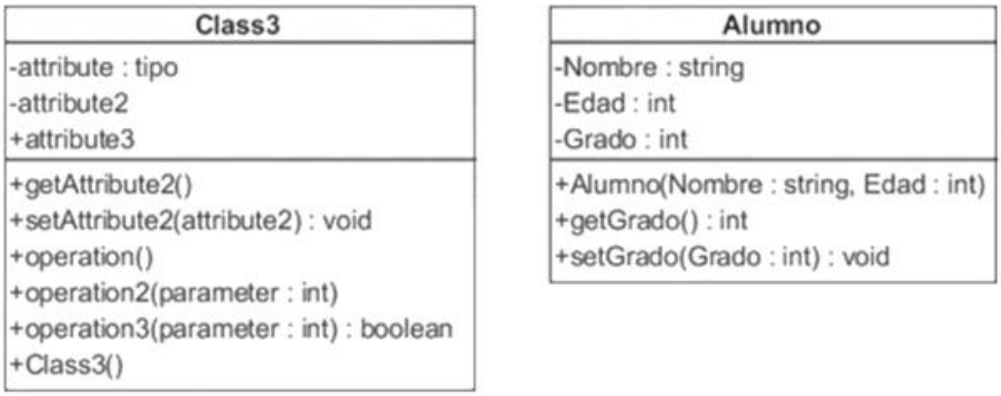
¿Para qué se utilizan los atributos en UML?

- Definen el estado interno de un objeto.
- Permiten modelar la información que la clase necesita almacenar.
- Son esenciales para identificar la estructura de datos en un sistema orientado a objetos.

Operaciones

Las **operaciones** representan **los métodos o funciones** que pertenecen a una clase. Son las acciones o comportamientos que una clase puede realizar o ejecutar sobre sus atributos u otros objetos.

- Visibilidad nombre(parámetros): tipo de retorno



- Se puede escribir en forma larga o completa “con parámetros”, forma corta “sin parámetros”.
- Class3() es un constructor que no recibe ni devuelve.
- Get y setter funciones de interface

Características clave de una operación:

- **Nombre**: Identifica la operación (por ejemplo, calcularPromedio()).
- **Visibilidad**: Controla el acceso a la operación (+ público, - privado, # protegido, ~ paquete).
- **Parámetros**: Pueden recibir uno o más parámetros con su tipo de dato (por ejemplo, agregarNota(nota: float)).
- **Valor de retorno**: Especifica el tipo de dato que devuelve la operación (por ejemplo, :float).
- **Modificadores**: Como static o abstract, que indican comportamientos especiales.
- **Excepciones**: En algunos casos, se pueden indicar excepciones que la operación podría lanzar.

Diferencia entre Operaciones y Métodos:

- En UML se habla de **operaciones**, que son **definiciones abstractas** del comportamiento de una clase.
- En el código fuente, estas operaciones se implementan como **métodos**.

Estereotipos

Un estereotipo es una extensión del modelo estándar que permite agregar significado adicional a los elementos de UML, como clases, interfaces o componentes.

Los estereotipos ayudan a personalizar UML para diferentes dominios o necesidades específicas, como diseño de software, bases de datos o sistemas embebidos.

En resumen:

- Nos permiten extender UML
- Lo usamos en elementos que no forman parte de UML, pero que son similares a alguno de UML
- Nos permiten definir nuestros propios elementos
- También ayudan a definir roles
- Otra función es la de especializar elementos de UML
- Las dependencias también se pueden estereotipar
- Se escriben de la forma <<nombre>>
- El nombre indica como se está especializando

<<NOMBRE>> es una dependencia estereotipada

Interfaces y enumeraciones no son parte de UML

¿Para qué se usan los estereotipos?

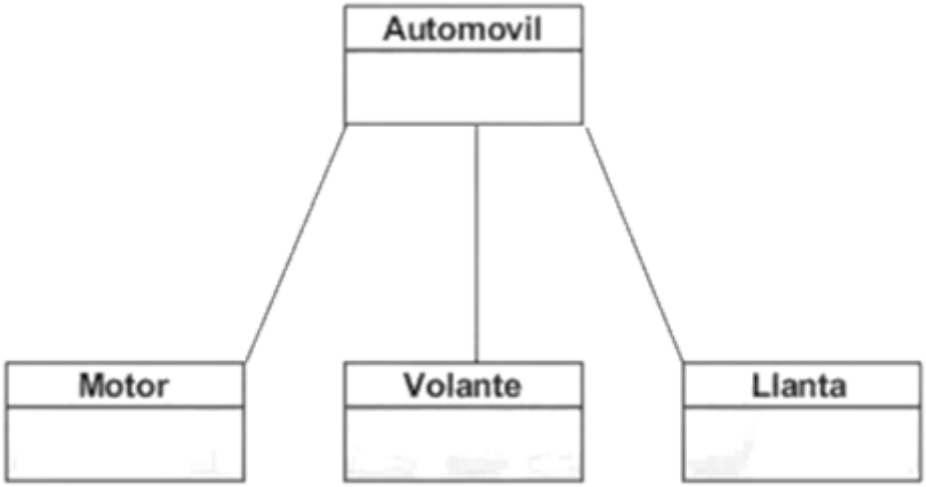
- **Distinguir responsabilidades** (entidad, control, límite).
- Extender UML para **dominios específicos** (por ejemplo, modelado de bases de datos).
- Mejorar la **comprensión del modelo** por parte del equipo de desarrollo.

Asociación

Una **asociación** representa una **relación estructural** entre **dos o más clases**, que indica cómo los objetos de una clase están **conectados o interactúan** con los objetos de otra clase.

- Muestra una relación entre clases
- Se lee generalmente como: tiene un
- Se muestra como una línea y a veces como una flecha abierta
- Se puede indicar por medio de un verbo el tipo de asociación
- < o > indican la dirección al ser colocados junto al verbo

Cuando no tiene sentido se lee como MAESTRO TIENE UN ALUMNO o viceversa



Una clase puede tener varias clases asociadas

Tipos de asociaciones:

- Asociación simple: Conexión entre dos clases (como Estudiante y Curso).
- Asociación reflexiva: Una clase se asocia consigo misma (por ejemplo, Empleado tiene un Jefe que también es un Empleado).
- Asociación n-aria: Involucra más de dos clases.
- Agregación (relación "parte-todo", más débil).
- Composición (relación "parte-todo", más fuerte).

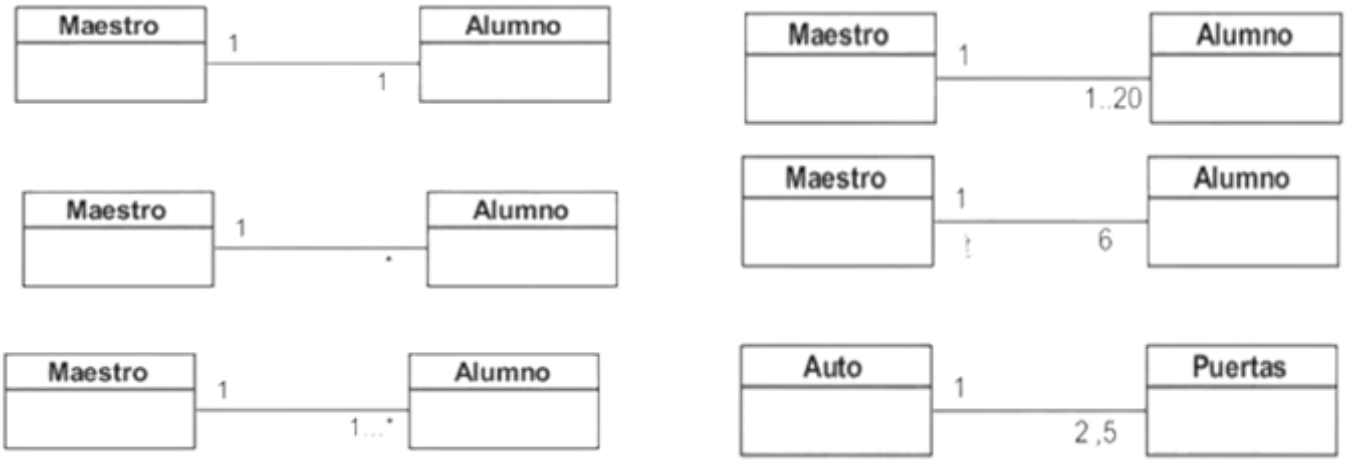
¿Para qué se usa una asociación en UML?

- Modelar la **interacción entre objetos**.
- Representar **relaciones lógicas o físicas** entre entidades del sistema.
- Definir **restricciones** sobre la cantidad de instancias que pueden estar conectadas.

Multiplicidad

La **multiplicidad** define **cuántas instancias de una clase pueden estar asociadas** con una instancia de otra clase.

- También se le llega a llamar cardinalidad
- Indica el número de objetos en una asociación



¿Para qué sirve la multiplicidad?

- Controla la cantidad de instancias permitidas en una relación.
- Permite definir restricciones y reglas de negocio.
- Ayuda a entender la estructura y el diseño del sistema.

Asociación reflexiva

Es un tipo de asociación en la que una **clase está relacionada consigo misma**. Esto permite modelar relaciones jerárquicas o auto-referenciadas, como una estructura de árbol, jerarquías organizacionales o listas enlazadas.

Explicación:

- Un Empleado puede supervisar a varios empleados (multiplicidad *).
- Un Empleado puede tener un único supervisor (multiplicidad 0..1).

¿Cuándo usar una asociación reflexiva?

- Cuando necesitas modelar una jerarquía o una estructura recursiva.

- Cuando una entidad puede tener una relación directa o indirecta con otras instancias de la misma clase.

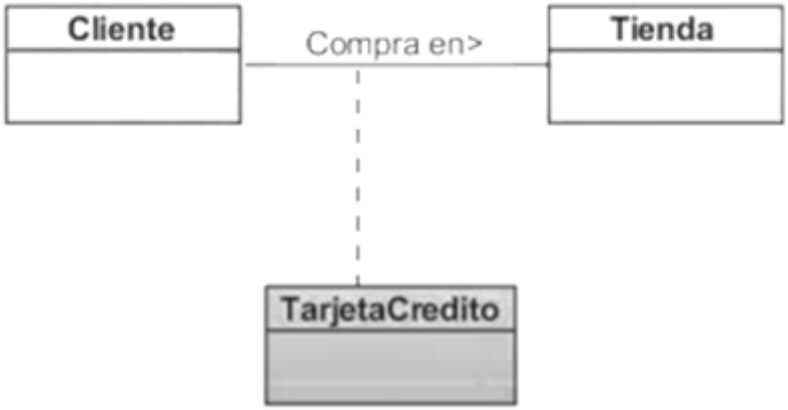
Clase asociada

Una clase asociada (también llamada clase de asociación) es una clase que **almacena información adicional** sobre una asociación entre dos o más clases.

Se utiliza cuando la relación entre dos clases tiene **atributos o comportamientos propios**, que no pueden ser representados directamente en ninguna de las clases involucradas.

En resumen:

- Cuando la asociación en si se lleva a cabo por medio de una clase que tiene sus propios atributos y operaciones
- El cliente compra en la tienda por medio de la tarjeta de crédito



Clase asociada tarjeta de crédito. Se lee como Cliente compra en tienda por Tarjeta de Crédito (línea punteada)

Explicación:

La clase Inscripción es una clase asociada que almacena información sobre la fecha de inscripción y la nota obtenida, datos que no pertenecen ni a Estudiante ni a Curso, sino a la relación entre ambos.

¿Cuándo usar una clase asociada?

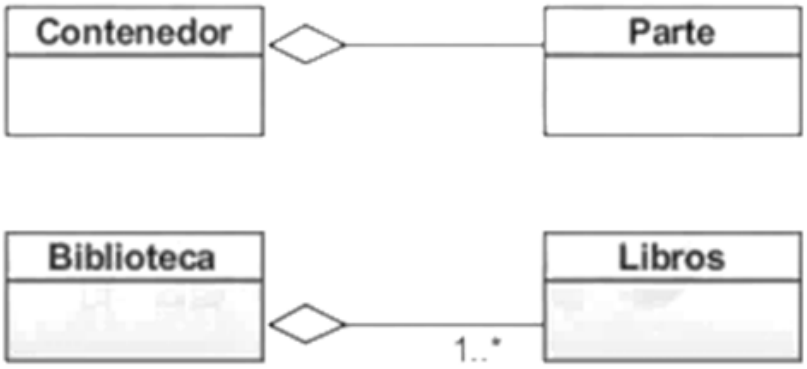
- Cuando la relación entre dos clases tiene atributos adicionales.
- Para modelar acciones o eventos que ocurren en la interacción entre las clases.
- Para evitar la redundancia de datos en las clases principales.

Agregación

La **agregación** es un tipo de **asociación débil** que representa una **relación "parte-todo"**, donde una clase contiene o agrupa a otras, pero las **partes pueden existir independientemente** del todo.

Es decir:

- Es parte de
- Está hecho de



- Se representa con un **diamante en blanco**.
- Tienen existencia por separado (si uno deja de existir, el otro puede existir)
- Libros es parte de Biblioteca o Biblioteca está hecho de Libros

Características clave de la agregación:

- Es una **relación débil**, lo que significa que las **partes no dependen de la existencia del todo**.
- Se representa con un **rombo blanco** en el extremo de la clase que actúa como **contenedor** o **todo**.
- Las **partes pueden ser compartidas** por otros objetos.

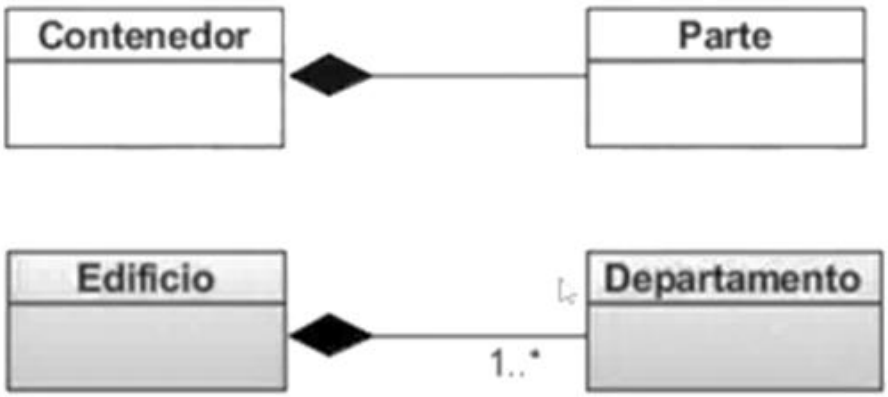
¿Cuándo usar una agregación?

- Cuando los objetos pueden ser **compartidos por múltiples contenedores**.
- Cuando deseas modelar una **relación jerárquica débil**.

Composición

La **composición** es un tipo de **asociación fuerte** que representa una **relación "parte-todo"**, donde las **partes no pueden existir independientemente** del todo.

- Si el contenedor se destruye, se destruyen las partes
- Regla de no compartir -> En la composición la parte puede pertenecer a un solo contenedor



- Se representa con un **rombo lleno**
- Regla de no CONVERTIR

- Parte puede pertenecer a un solo contenedor. Ej.: Departamento solo puede pertenecer a un Edificio

Características clave de la composición:

- Es una **relación fuerte**, lo que significa que, si el **todo se destruye, las partes también se destruyen**.
- Se representa con un **rombo negro** en el extremo de la clase que actúa como el **todo**.
- Las **partes son exclusivas** de ese todo y **no pueden ser compartidas** por otros objetos.

¿Cuándo usar una composición?

- Cuando las **partes no pueden existir sin el todo**.
- Para modelar una **relación de dependencia total**.

Diferencia entre Agregación y Composición:

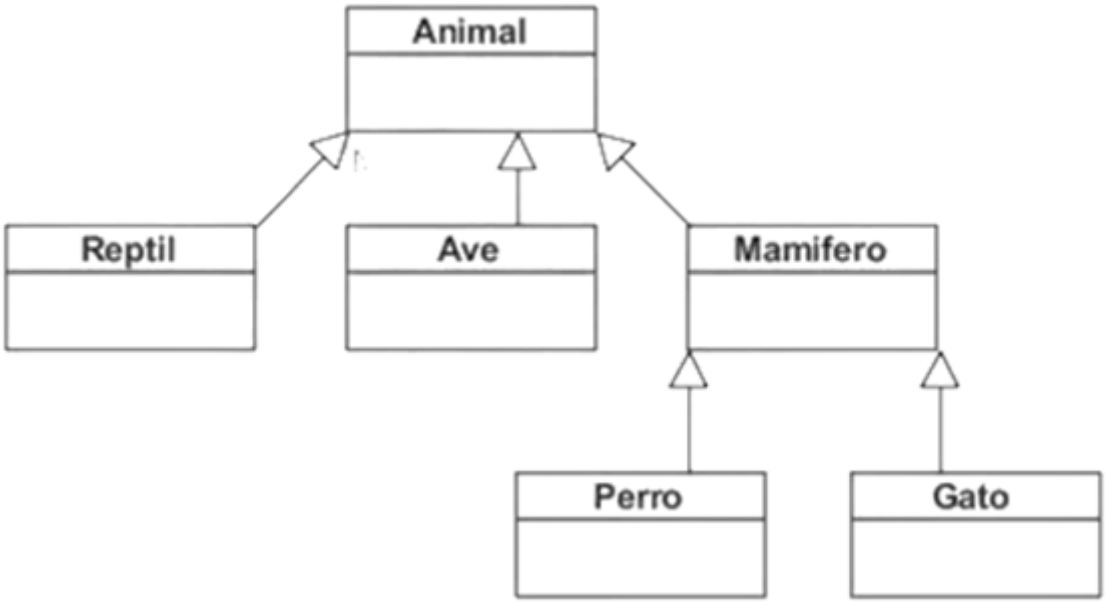
Característica	Agregación	Composición
Relación	Débil ("tiene una")	Fuerte ("es parte de")
Independencia	Las partes pueden existir sin el todo.	Las partes dependen del todo.
Representación en UML	Rombo blanco ◇	Rombo negro ◆

Generalización

La generalización representa una **relación jerárquica** entre una clase padre (superclase) y una o más clases hijas (subclases).

Esta relación indica que las subclases heredan atributos, métodos y comportamientos de la superclase.

- Describe la herencia
- Es un



Explicación:

- Animal es la superclase, que puede tener atributos como nombre y edad, y métodos como respirar(), moverse(), alimentarse().
- Reptil, Ave y Mamífero son subclases que heredan los atributos y comportamientos de Animal, pero especializan ciertas características, como la forma de desplazarse o reproducirse.
- Perro y Gato son subclases de Mamífero. Estas heredan los atributos y métodos tanto de Animal como de Mamífero, pero también pueden tener comportamientos específicos como ladrar() o maullar().

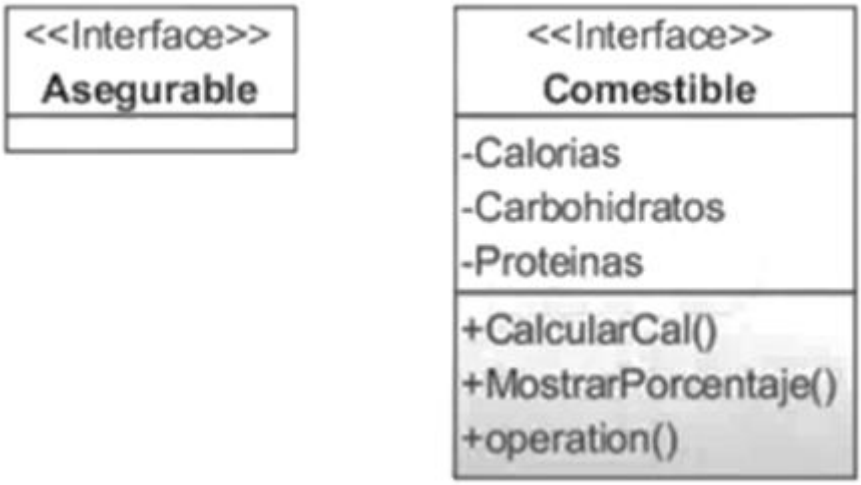
Características clave de la generalización:

- **Permite reutilizar código y comportamientos comunes**.
- Las subclases pueden **extender o especializar** las características heredadas.
- Se representa con una **flecha vacía con punta triangular** que apunta hacia la superclase.

Realización

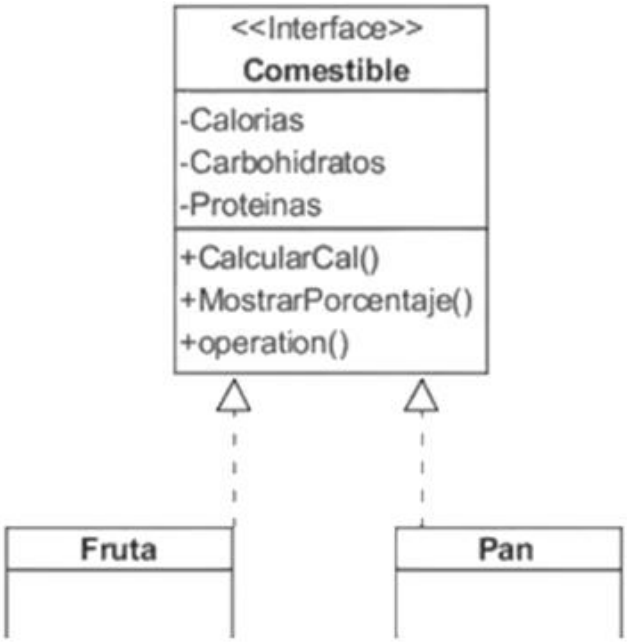
La **realización** es una relación que indica que una **clase concreta o componente implementa el comportamiento definido en una interfaz**.

- Muestra una implementación
- Muestra la relación entre una interfaz y una clase o componente



* Algunos lenguajes no permitan interfaces que tengan atributos

- Implementa



Características clave:

- La **interfaz** define un **conjunto de métodos o comportamientos**, pero **no implementa su lógica**.
- La **clase concreta** que realiza la interfaz debe **proveer la implementación de todos los métodos definidos**.
- Se representa con una **línea discontinua con una flecha triangular vacía** que apunta hacia la interfaz.

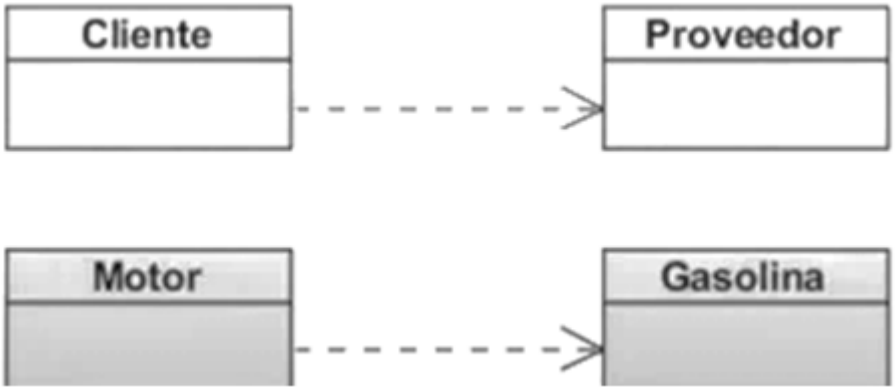
¿Cuándo usar una realización en UML?

- Cuando deseas **definir un contrato o comportamiento común** que diferentes clases deben implementar.
- Para **garantizar la consistencia** en la implementación de ciertas funcionalidades.
- En sistemas orientados a **interfaces y polimorfismo**.

Dependencia

Una **dependencia** representa una **relación débil y temporal entre dos elementos**, donde **un cambio en un elemento puede afectar el comportamiento o la implementación del otro**.

- Muestra una relación en la que una clase usa a otra clase de alguna manera
- Usa a
- Cambios en el proveedor afectan al cliente, cuidado con eso



CUIDADO: un cambio en el proveedor afecta al cliente a nivel sistema.

Características clave de la dependencia:

- Se representa con una **línea discontinua con una flecha** que apunta desde el **elemento dependiente hacia el elemento del que depende**.
- Indica que **un elemento usa o necesita otro** para funcionar correctamente, pero **no tiene una relación directa o permanente**.
- Se suele utilizar para **representar el uso de clases, métodos o bibliotecas externas**.

¿Cuándo usar una dependencia en UML?

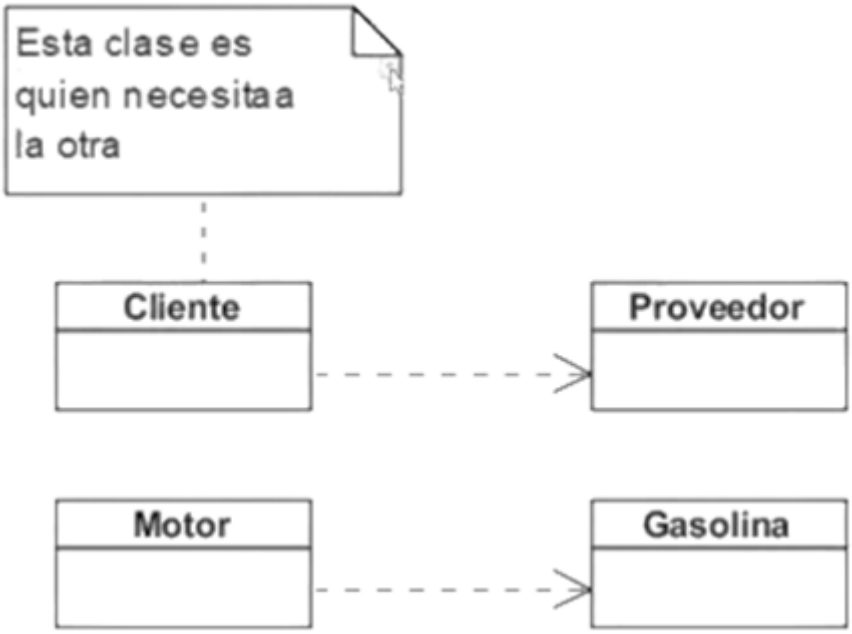
- Cuando una clase **utiliza temporalmente otra clase**, como al **invocar un método o acceder a una función**.
- Para modelar el uso de **bibliotecas externas o servicios web**.
- Para mostrar la relación entre **módulos o componentes** en un sistema.

Notas

Las **notas** son elementos gráficos utilizados para **agregar comentarios, aclaraciones o información adicional** en un diagrama.

Ventajas del uso de notas en UML:

- Mejoran la **comunicación entre desarrolladores y analistas**.
- Permiten **documentar el modelo sin afectar su estructura**.
- Facilitan la **comprensión del diagrama** para otros equipos o futuros mantenedores del sistema.
- Nos sirven para dar indicaciones o aclaraciones sobre algo
- Usamos un ancla para unirla al elemento



Características clave de las notas en UML:

- **No afectan la lógica del modelo**, solo aportan información adicional para **mejorar la comprensión del diagrama**.
- Se representan con un **rectángulo con una esquina doblada**, similar a una "nota adhesiva".
- Pueden contener **texto libre**, como explicaciones, advertencias o detalles sobre decisiones de diseño.
- Pueden estar **conectadas a uno o varios elementos del diagrama** mediante una **línea discontinua**.

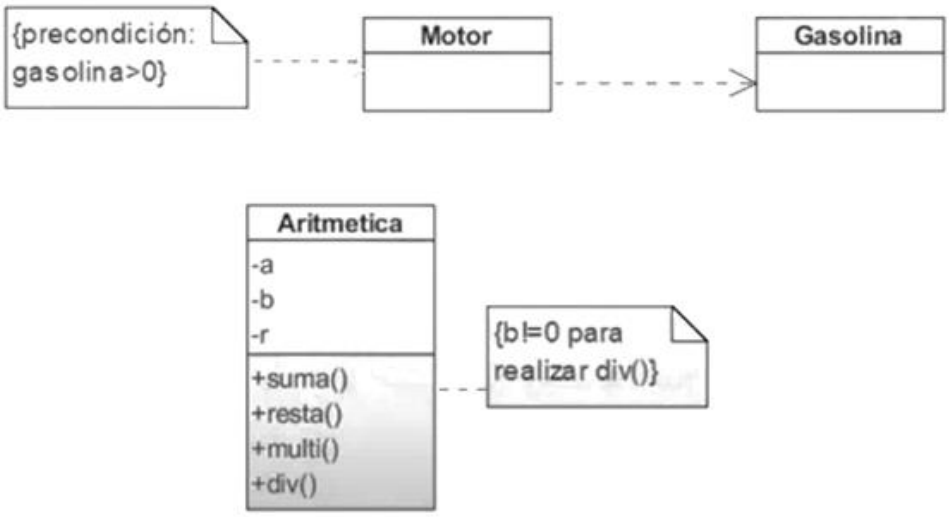
¿Para qué se utilizan las notas?

- **Aclarar decisiones de diseño.**
- Documentar **requisitos adicionales o restricciones**.
- Incluir **detalles de implementación** o advertencias para otros desarrolladores.
- Añadir **explicaciones sobre relaciones complejas** entre clases o componentes.

Restricciones

Las **restricciones** son **reglas o condiciones** que **limitan o controlan el comportamiento** de los elementos del modelo, como clases, atributos, relaciones o asociaciones. Muestra limitaciones o condiciones de un objeto en un diagrama

- Podemos indicar precondiciones o postcondiciones
- Pueden colocarse en notas, o en algunos casos en el mismo elemento
- La multiplicidad es un tipo de restricción



Características clave de las restricciones:

- Se utilizan para **garantizar la coherencia y la integridad del sistema**.
- Se expresan entre **llaves** `{}`.
- Pueden definir **condiciones lógicas**, **restricciones de valores**, **cardinalidad** o **reglas de negocio**.
- Pueden aplicarse a **clases, atributos, operaciones o relaciones**.

Tipos de restricciones en UML:

- **Restricciones estructurales:**
  - Definen reglas sobre **valores de atributos** o **relaciones entre clases**.
  - Ejemplo: {saldo >= 0} para una cuenta bancaria.
- **Restricciones de cardinalidad:**
  - Controlan la **cantidad mínima o máxima de instancias** en una relación.
  - Ejemplo: {1..*} indica que debe haber **al menos una instancia**.
- **Restricciones de integridad:**
  - Aseguran que ciertas **operaciones solo se ejecuten bajo condiciones específicas**.
  - Ejemplo: {soloAdministradorPuedeModificar}.

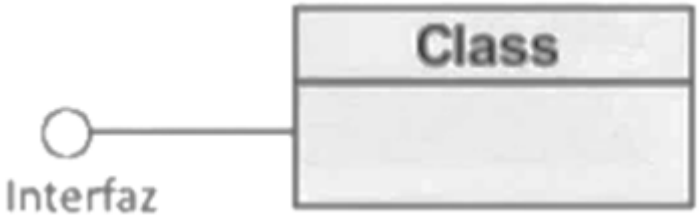
En conclusión

Las restricciones en UML son esenciales para **definir reglas de negocio, mantener la integridad del modelo** y **evitar errores lógicos** en el diseño del sistema.

Interfaz proveedor

Una **Interfaz Proveedor** representa un **punto de acceso** que **expone los servicios o funcionalidades** que una clase o componente ofrece a otros elementos del sistema.

- La clase provee una implementación a la interfaz
- Lollipop



La clase provee una implementación a la interfaz proveída (círculo completo)

Características clave de la Interfaz Proveedor:

- **Define un contrato** con métodos que otras clases o componentes pueden invocar.
- Se representa con un **círculo (llamado "bola de lollipop")** conectado a la clase o componente que la implementa.
- La clase que **implementa la interfaz** es responsable de **proveer la lógica** para los métodos definidos en ella.

¿Cuándo se utiliza una Interfaz Proveedor?

- Para garantizar la modularidad y el principio de bajo acoplamiento.
- Permite que múltiples clases implementen la misma interfaz, facilitando la extensibilidad.
- Facilita el uso de inyección de dependencias y la sustitución de implementaciones sin afectar el resto del sistema.

Interfaz requerida

Una **Interfaz Requerida** representa una **dependencia o necesidad** que una clase o componente tiene para funcionar, es decir, **los servicios que necesita consumir** de otra clase o componente.

- La clase requiere de la interfaz para poder hacer alguna funcionalidad
- Requiere de otra clase que la implemente

La implementación de la interfaz requerida (medio círculo) la realiza otra clase

Características clave de la Interfaz Requerida:

- Se representa con un **semicírculo o "socket"** que apunta hacia la interfaz que necesita.
- Indica que el **componente no puede operar por sí solo**, sino que **depende de otro componente que implemente esa interfaz**.
- Facilita el **desacoplamiento** y la **modularidad** en sistemas complejos.

¿Cuándo usar una Interfaz Requerida en UML?

- Cuando un **componente necesita consumir servicios** que otro componente proporciona.
- Para **desacoplar módulos** y facilitar la **sustitución de implementaciones**.
- En **arquitecturas orientadas a servicios** o **microservicios**.

Diferencia entre Interfaz Proveedor y Requerida:

Interfaz Proveedor	Interfaz Requerida
Define **lo que una clase ofrece** (implementa).	Define **lo que una clase necesita** para funcionar.
Representada con una **"bola de lollipop" (◯)**.	Representada con un **"socket" (○)**.
Ejemplo: `Servicio` implementado por `Proveedor`.	Ejemplo: `Cliente` necesita `Servicio`.

Dentro del diagrama de clases

- En el primer ejemplo vemos que Class2 tiene una interface requerida y Class1 tiene una interface proveedor.
- Class1 implementa la interface. Class2 necesita de una instancia de Class1
- Si no hay demasiado espacio en el diagrama, podemos utilizar la figura 2.

En resumen

Una interfaz provista es una afirmación que el clasificador contenido provee a las operaciones a las operaciones definidas por el elemento de la interfaz nombrada y se define dibujando un vínculo de realización entre la clase y la interfaz. Una interfaz requerida es un estado que el clasificador puede comunicar con algún otro clasificador que provee operaciones definidas por el elemento de la interfaz.

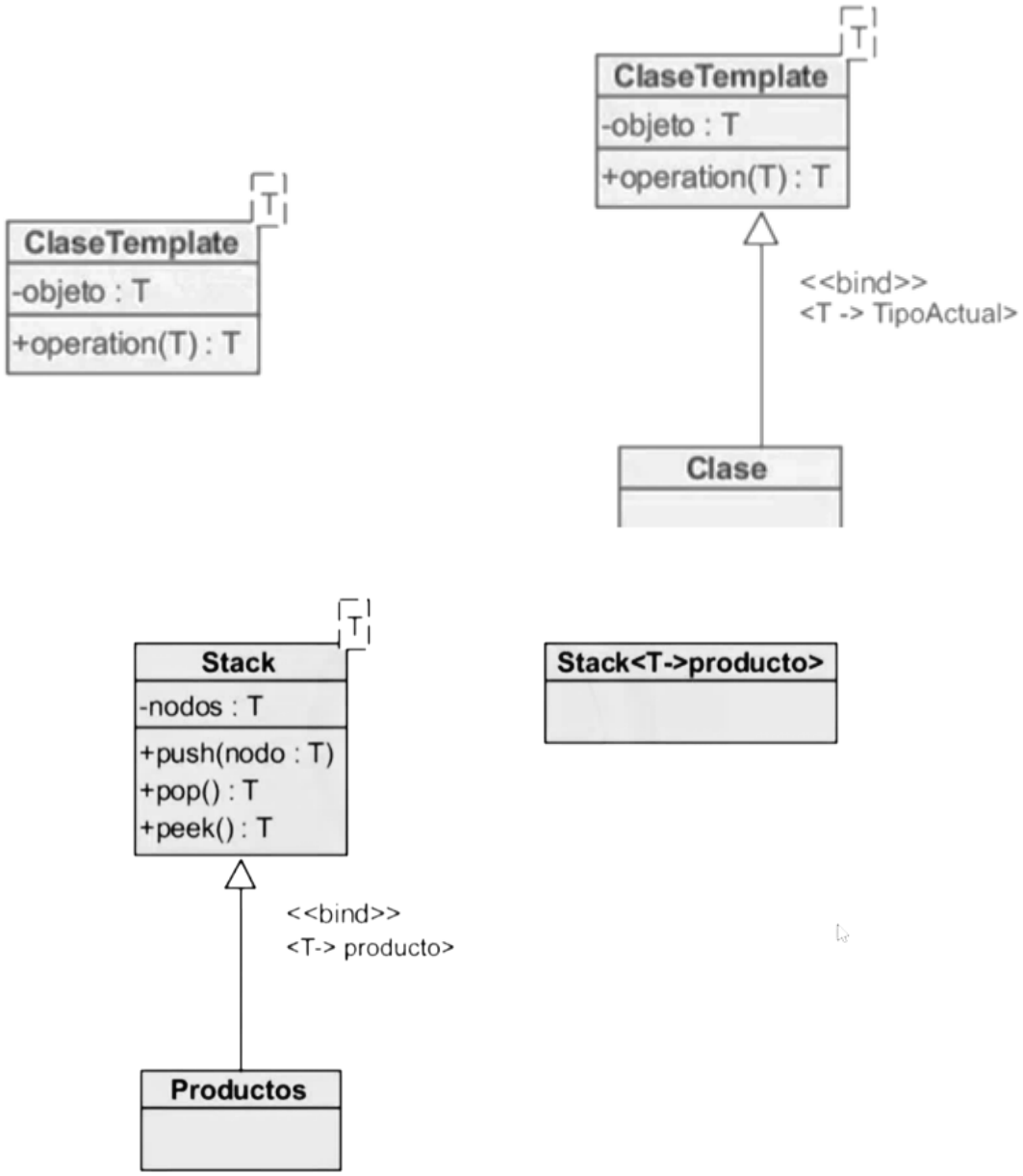
Una interfaz provista se muestra como un “pelota en un palo” adjuntada al borde de un elemento clasificador. Una interfaz requerida se muestra como una “copa en un palo” adjuntada al borde de un elemento clasificador.

Template

El concepto de **Template (Plantilla)** se refiere a la **definición genérica de una clase, interfaz o método**, que puede ser **parametrizada con tipos de datos específicos** cuando se utiliza.

Permite **reutilizar una estructura o lógica común** para diferentes tipos de datos, sin necesidad de duplicar código. Es similar a los **genéricos en lenguajes de programación**, como en **Java**, **C++** o **C#**.

- Elemento parametrizado que puede ser usado para generar otros elementos del modelo
- Funciona similar a los tipos genéricos en C# y otros lenguajes <T>
- <T -> TipoActual



Aplicaciones comunes:

- Permite **evitar duplicación de código**.
- Aumenta la **flexibilidad y reutilización** de clases o métodos.
- Se usa en **estructuras de datos genéricas**, como listas, pilas o colas.

Analogía entre algunos lenguajes:

Lenguaje	Palabra clave
Java	`class<T>`
C++	`template<class T>`
C#	`class<T>`
Python	No es nativo, pero se puede simular con `typing`

Tabla de contenido