

Elaboración de diagramas de clases

Objetivos

- Aprender los conceptos básicos de la orientación a objetos.
- Comprender los fundamentos del lenguaje UML.
- Conocer los tipos de diagramas que se pueden construir en el lenguaje UML.
- Saber cómo representar clases empleando el lenguaje UML.
- Reconocer y diferenciar los tipos de relaciones que se pueden establecer entre las clases.
- Conocer la manera de representar las diferentes relaciones entre clases empleando el lenguaje UML.
- Distinguir los tipos de clases de análisis de interfaz, de control y de entidad.
- Utilizar herramientas para la elaboración de diagramas de clases.
- Generar código a partir de un diagrama de clases.
- Crear un diagrama de clases a partir de código mediante ingeniería inversa.

Contenidos

- 5.1. Conceptos básicos de la orientación a objetos
- 5.2. El lenguaje UML
- 5.3. Clases, atributos, métodos y visibilidad
- 5.4. Relaciones entre clases
- 5.5. Tipos de clases de análisis
- 5.6. Herramientas para la creación de diagramas de clases
- 5.7. Generación de código a partir de diagramas de clases
- 5.8. Generación de diagramas de clases a partir de código (ingeniería inversa)

Introducción

Para crear una aplicación informática, es necesario llevar a cabo una serie de tareas que ya fueron descritas en las unidades previas. Durante las etapas de análisis y diseño, se deben obtener modelos gráficos que representen la información que es necesario manejar en la aplicación y las operaciones que se deben realizar sobre dicha información. La creación de estos modelos se debe realizar siguiendo ciertas normas y el estándar a nivel internacional para ello es el lenguaje UML (*unified modeling language*).

Una aplicación orientada a objetos consta de una serie de clases relacionadas entre sí, a partir de las cuales, se crean objetos que se envían mensajes. Con relación a ello, el lenguaje UML establece las normas para crear muchos tipos diferentes de diagramas, de los cuales los más relevantes son los diagramas de clases, a los que se dedica esta unidad. Así, se estudiarán **cómo se representan las clases y sus relaciones** siguiendo el lenguaje UML y cómo crear estos diagramas empleando varias herramientas informáticas. También se verá la manera de generar código a partir de un diagrama de clases, y viceversa.

■ 5.1. Conceptos básicos de la orientación a objetos

Como se indicó en el Apartado 1.2, el *paradigma orientado a objetos* supuso en cierto modo un cambio de visión en relación con el desarrollo de software. Básicamente, se pasó de prestar atención a los procesos que se tenían que realizar en una aplicación a dar mayor relevancia a los datos sobre los que se debía operar.

En el paradigma orientado a objetos, una aplicación consta de objetos que interactúan entre ellos para llevar a cabo las funciones encomendadas al software. Como se indicó en la Unidad 1, un objeto es una instancia de una clase, y toda clase tiene una serie de propiedades o atributos y realiza un conjunto de operaciones llamadas *métodos*. Se puede definir un *mensaje* como una solicitud de un objeto para que otro objeto o él mismo ejecute uno de sus métodos.

Para que un lenguaje de programación pueda ser calificado como orientado a objetos precisa tener una serie de características, que se enumeran a continuación (Coad y Yourdon, 1991):

- **Abstracción:** cada objeto en el sistema funciona como un agente abstracto que puede realizar trabajo, informar y cambiar su estado, así como comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características. El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real.
- **Encapsulamiento:** hace referencia a reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad. Esto permite aumentar la cohesión de los componentes del sistema.

- **Polimorfismo:** comportamientos diferentes asociados a objetos distintos pueden compartir el mismo nombre; al llamarlos por este nombre, se utilizará el comportamiento correspondiente al objeto que se esté usando. Esto hace posible definir varios métodos con el mismo nombre y la misma interfaz en diferentes clases, y también definir varios métodos con el mismo nombre en una misma clase, pero con interfaz distinta (distinto número y/o tipo de parámetros y/o tipo de valor devuelto).
- **Herencia:** las clases con atributos o métodos comunes se relacionan entre sí formando jerarquías de clasificación en las que las subclases (clases hijas) poseen por herencia los atributos y métodos de la superclase o de las superclases (clase padre o clases padres). La herencia posibilita el polimorfismo y el encapsulamiento, lo que permite crear clases como tipos especializados de clases preexistentes.
- **Modularidad:** es la propiedad que permite dividir una aplicación en partes más pequeñas llamadas **módulos**, cada uno de los cuales debe ser lo más independiente posible de los otros módulos, es decir, el acoplamiento entre los módulos debe ser el menor posible.
- **Ocultamiento:** cada objeto está aislado del exterior y expone su interfaz a otros objetos, la cual especifica cómo interactuar con los objetos de la clase. Esta propiedad protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, de manera que solo los métodos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, y así se evitan efectos secundarios e interacciones inesperadas.
- **Recolección de basura:** es la técnica por la cual el entorno se encarga de destruir automáticamente los objetos que hayan quedado sin ninguna referencia a ellos, desvinculando la memoria asociada. Gracias a esta característica el programador no necesita preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un objeto y la liberará cuando nadie la esté usando. En algunos lenguajes orientados a objetos que surgieron como extensión de lenguajes estructurados, como es el caso de C++ y Object Pascal, esta característica no existe y debe anularse la asignación de la memoria expresamente.

Por tanto, se puede considerar que un lenguaje de programación orientado a objetos es el que cumple todas estas características, si bien se podría decir que la última de estas propiedades (*recolección de basura*) no es del todo necesaria para considerar a lenguajes como C++ u Object Pascal como orientados a objetos.

5.2. El lenguaje UML

El modelo orientado a objetos surgió como parte de las metodologías de análisis y diseño orientado a objetos, que comenzaron su andadura durante los años 80 del siglo pasado. A partir del año 1996, se integraron varios de los métodos de análisis y diseño orientado a objetos, lo que dio lugar a la aparición del *lenguaje unificado de modelado* (UML), cuya última versión es la 2.5.1, que data del año 2015.

Los principios básicos que se deben seguir a la hora de diseñar el modelo de un sistema son los siguientes:

- La elección acerca de qué modelos crear tiene una enorme influencia sobre cómo se acomete un problema y cómo se da forma a una solución.
- Todo modelo puede ser expresado con diferentes niveles de precisión.
- Los mejores modelos están ligados a la realidad.
- Un único modelo o vista no es suficiente. Cualquier sistema no trivial se aborda mejor mediante un pequeño conjunto de modelos casi independientes y desde múltiples puntos de vista.

El lenguaje UML sigue estos principios. Cabe afirmar que UML es un lenguaje gráfico que se emplea para visualizar, especificar, construir y documentar un sistema por medio de diferentes tipos de diagramas que se utilizan en las diferentes tareas de desarrollo del software. Estos modelos se corresponden con diferentes niveles de abstracción y presentan diferente nivel de detalle, o dicho de otro modo, empleando UML se puede estudiar el sistema desde diferentes puntos de vista, ya que no todos los usuarios de un sistema precisan tener la misma visión de este. UML incorpora distintos tipos de diagramas y notaciones gráficas y textuales destinadas a mostrar el sistema desde diferentes perspectivas, que pueden usarse en las diferentes fases del ciclo de vida del desarrollo de software. UML establece las normas que se deben seguir cuando se elaboran estos diagramas.

■ ■ ■ 5.2.1. Tipos de elementos en UML

En el lenguaje UML, existen **cuatro tipos de elementos** que se describen a continuación.

■ ■ ■ Elementos estructurales

En su mayoría, los elementos estructurales son la parte estática de un modelo y representan conceptos o cosas materiales. Son los siguientes:

1. **Clase:** es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Gráficamente, una clase se representa como un **rectángulo** que incluye su nombre, sus atributos y sus operaciones o métodos, como se muestra en la Figura 5.1. En los siguientes apartados de esta unidad, se explica con mayor detalle la manera de representar clases y los diagramas de clases.
2. **Interfaz:** es una colección de operaciones que especifican un servicio de una clase o componente. Por lo tanto, una interfaz describe el comportamiento visible desde el exterior de ese elemento. Una interfaz define un conjunto de especificaciones de operaciones, pero no un conjunto de implementaciones de operaciones. Esto quiere decir que de las operaciones solo se conoce su nombre, los datos que hay que proporcionar para llevarlas a cabo y los datos que devuelven, si fuera el caso, pero no se conoce **cómo** se llevan a cabo las operaciones, es decir, **cómo** estas

se implementan. Gráficamente, una interfaz se representa mediante un **rectángulo** que incluye su nombre **con la leyenda <<Interface>>** y sus operaciones o métodos, tal y como se muestra en la Figura 5.2.

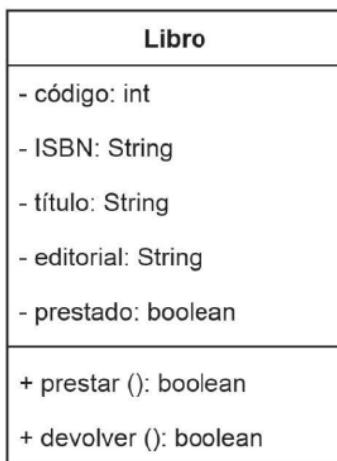


Figura 5.1. Clase llamada Libro con cinco atributos (código, ISBN, título, editorial y prestado) y dos métodos (prestar y devolver).



Figura 5.2. Interfaz IVentana con las operaciones que ofrece (abrir, cerrar, mover y dibujar).

3. **Colaboración:** define una interacción y se trata de una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Se representa gráficamente por medio de una **elipse con borde discontinuo** que normalmente incluye solo su nombre (Figura 5.3).

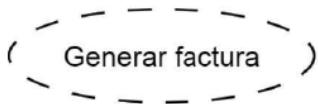


Figura 5.3. Representación de una colaboración, en este caso Generar factura.

4. **Caso de uso:** describe un comportamiento de un sistema, clase o componente desde el punto de vista de la persona usuaria. Describe un conjunto de secuencias de acciones que ejecuta un sistema y que produce un resultado observable de interés para un usuario o una usuaria particular. Gráficamente, se representa mediante una **elipse con borde continuo** que, por lo general, solo incluye su nombre (véase la Figura 5.4).



Figura 5.4. Representación del caso de uso *Registrar pedido*, que describe un conjunto de operaciones que realiza la aplicación y que son de interés para la persona que la utiliza.

5. **Clase activa:** es un tipo especial de clase cuyos objetos tienen uno o más procesos o hilos de ejecución. Una clase activa es lo mismo que una clase excepto en el hecho de que sus objetos pueden ejecutarse concurrentemente con otros objetos de clases activas. Se representa mediante un rectángulo, al igual que una clase, pero con **líneas dobles a la derecha e izquierda** (Figura 5.5).

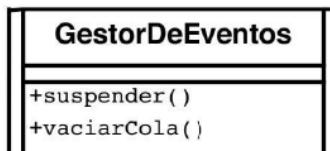


Figura 5.5. Representación de la clase activa *GestorDeEventos* con los métodos *suspender* y *vaciarCola*.

6. **Componente:** es una parte modular del diseño del sistema que oculta su implementación tras un conjunto de interfaces externas. El sistema se define a partir de los componentes conectados entre sí. La implementación de un componente puede expresarse conectando partes y conectores, y las partes pueden incluir componentes más pequeños. Por consiguiente, los componentes pueden ser de **granularidad variable**. Gráficamente, se representa como una clase con un **ícono especial en la esquina superior derecha** (véase la Figura 5.6).

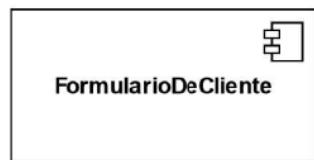


Figura 5.6. Representación del componente *FormularioDeCliente*.

Argot técnico

El vocablo **granularidad** tiene diferentes acepciones, pero la que aquí interesa es la siguiente: «el nivel de detalle considerado en un modelo o proceso». Cuanto mayor sea la granularidad, mayor será el nivel de detalle. Así, cabe afirmar que un componente con procesos genéricos tiene un nivel de granularidad bajo y, por este motivo, se puede dividir en componentes más detallados, es decir, en componentes con una granularidad mayor o más nivel de detalle.



7. **Artefacto:** es una parte física y reemplazable de un sistema que contiene información física (*bits*). En un sistema, existen diferentes tipos de artefactos de despliegue, como archivos de código fuente, ejecutables y *scripts*. Los artefactos se representan por medio de un **rectángulo con la leyenda «artifact»** sobre el nombre (Figura 5.7).

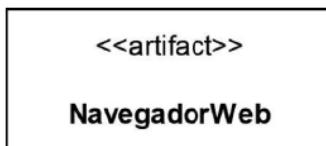


Figura 5.7. Representación de un artefacto llamado NavegadorWeb.

8. **Nodo:** es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional que, por lo general, dispone de algo de memoria y, con frecuencia, capacidad de procesamiento. Un conjunto de artefactos pueden residir en un nodo. Sirven para describir las plataformas en las que se ejecutan las aplicaciones. Gráficamente, se representa mediante un **cubo**, en cuyo interior se puede leer su nombre (Figura 5.8).

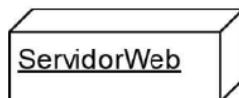


Figura 5.8. Nodo que representa un servidor web.

■■■ Elementos de comportamiento

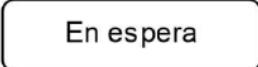
Los elementos de comportamiento son las partes dinámicas de los modelos UML y representan un comportamiento en el tiempo y en el espacio. Suelen estar conectados semánticamente a elementos estructurales. Existen de tres tipos:

1. **Interacción:** comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular y para un propósito específico. Un **mensaje** se representa gráficamente por medio de una **flecha** con el nombre de la operación sobre ella (Figura 5.9).



Figura 5.9. Mensaje que solicita la impresión de una factura.

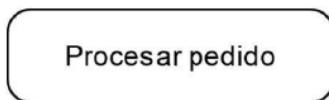
2. **Máquina de estados:** especifica las secuencias de estados por los que pasa un objeto o una interacción durante su vida, en respuesta a eventos, junto con sus reacciones a esos eventos. Sirve para especificar el comportamiento de una clase individual o de una colaboración de clases. Una máquina de estados involucra los siguientes elementos: **estados**, **transiciones** (paso de un estado a otro), **eventos** (disparadores de una transición) y **actividades** (la respuesta a una transición). Gráficamente, un estado se representa como un **rectángulo con esquinas redondeadas** con el nombre en su interior (Figura 5.10).



En espera

Figura 5.10. Representación de un estado en espera.

3. **Actividad:** especifica la secuencia de pasos que ejecuta un proceso computacional. En una máquina de estados, el énfasis se pone en el ciclo de vida de un objeto, mientras que en una actividad, lo importante es la secuencia o el flujo de pasos, sin importar qué objeto ejecuta cada paso. Cada paso de una actividad recibe el nombre de **acción**. Una acción se representa por medio de un rectángulo con esquinas redondeadas, en cuyo interior, se lee su nombre, que **indica su propósito** (véase la Figura 5.11). Los estados y las acciones se distinguen por sus diferentes contextos, pues, aunque los estados y las acciones se representan de igual modo, los diagramas en los que se incluyen (diagramas de estados y diagramas de actividades, respectivamente) no tienen la misma forma, como se verá posteriormente en la Unidad 6. Esto se debe a que los estados incluidos en un diagrama de estados no se conectan de igual manera que las actividades que aparecen en un diagrama de actividades.

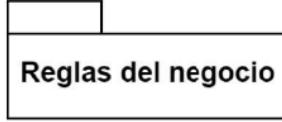


Procesar pedido

Figura 5.11. Acción que hace referencia al procesamiento de un pedido. Las acciones se representan de igual modo que los estados, pero se distinguen de estos por el contexto en el que se usan.

Elementos de agrupación

Son las partes organizativas de los modelos UML. El principal elemento de agrupación es el paquete. Frente a las clases, que organizan construcciones de implementación, un paquete es un mecanismo de propósito general para organizar el diseño. Un paquete puede incluir elementos estructurales, de comportamiento y otros paquetes. Al contrario de los componentes, que existen en tiempo de ejecución, los paquetes son puramente conceptuales, de manera que solo existen en tiempo de desarrollo. Los paquetes se representan gráficamente como una **carpeta** que lleva en su interior el nombre del paquete (Figura 5.12).



Reglas del negocio

Figura 5.12. Representación de un paquete llamado Reglas del negocio.

Un paquete puede contener otros paquetes sin límite de *anidamiento*, pero cada elemento pertenece a solo un paquete. Los paquetes se pueden utilizar en cualquier diagrama

UML y la visibilidad de los elementos incluidos en un paquete puede controlarse para que algunos elementos sean visibles desde el exterior del paquete, mientras que otros permanecen ocultos.

Argot técnico



El vocablo **anidamiento** en el diccionario de la RAE viene definido como «acción y efecto de anidar» y el verbo anidar posee varias acepciones. En este ámbito, la que se aplica es la de «abrigar, acoger». Así, si se dijera que se admiten paquetes de hasta dos niveles de anidamiento, se querría decir que un paquete puede acoger en su interior a varios paquetes, y cada uno de estos puede acoger a varios paquetes más pequeños. Si solo se permite dos niveles de anidamiento, cada uno de los paquetes de tercer nivel ya no se puede descomponer en paquetes más pequeños. El hecho de que no haya límite de anidamiento quiere decir que no existe ninguna limitación en cuanto a los niveles de descomposición de un paquete en paquetes más pequeños.

Los paquetes son los elementos básicos de agrupación, pero también hay otros, como *frameworks*, modelos y subsistemas.

Argot técnico



El término inglés **framework** se puede traducir por ‘marco de trabajo’ y en Wikipedia se define como «un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia para enfrentar y resolver nuevos problemas de índole similar» (<https://es.wikipedia.org/wiki/Framework>). Este término se usa ampliamente en el mundo del software, en el que se podría describir como un conjunto de herramientas que hacen posible que los desarrolladores sean más eficientes en su trabajo. Un *framework* ofrece un conjunto de componentes de software listos para su uso y reutilización. Ejemplos de ello son los *frameworks* .NET de Microsoft, Angular.js y CakePHP.

■ ■ ■ Elementos de anotación

Son las partes explicativas de los modelos UML. Son comentarios que se añaden para describir, clarificar y hacer observaciones. El elemento de anotación más importante se llama *nota*. Una *nota* es simplemente un símbolo para mostrar restricciones y comentarios asociados a un elemento o a una colección de elementos. Gráficamente, una nota se representa por medio de un **rectángulo con una esquina doblada** junto con un comentario textual o gráfico (véase la Figura 5.13).

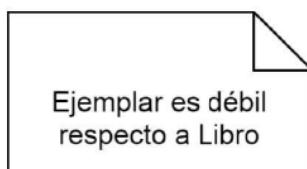


Figura 5.13. Nota que indica que la clase Ejemplar es débil respecto a la clase Libro.

5.2.2. Tipos de diagramas en UML

En UML se pueden construir hasta 14 tipos de diagramas distintos que se pueden agrupar en **dos tipos de diagramas genéricos**: estructurales y de comportamiento. En los siguientes subapartados, se describen todos los tipos de diagramas y se clasifican según su tipo.

Diagramas estructurales

Estos diagramas sirven para visualizar, construir, especificar y documentar los aspectos estáticos de un sistema. Se pueden emplear los **siete tipos** de diagramas estructurales que se indican a continuación:

1. **Diagramas de clases**: muestran un conjunto de clases y las relaciones entre ellas. Son los diagramas más comunes en el análisis y diseño de un sistema. Un diagrama de clases incluye básicamente clases (con atributos y métodos) y relaciones (asociación, agregación, composición, generalización y dependencia). Se explicarán en detalle los diagramas de clases en los Apartados 5.3 y 5.4.
2. **Diagramas de objetos**: muestran un conjunto de objetos y sus relaciones. Se pueden considerar como un caso especial de diagramas de clases en los que se muestran objetos, es decir, instancias de clases en un momento del tiempo. Por ello, cabe decir que representan instantáneas estáticas de los elementos existentes en los diagramas de clases.
3. **Diagramas de componentes**: describen la estructura del software mostrando la organización y las dependencias entre un conjunto de componentes. Un diagrama de componentes representa la encapsulación de una clase, junto con sus interfaces, puertos y estructura interna, la cual está formada por otros componentes anidados y conectores. Se usan para construir sistemas grandes a partir de partes pequeñas.
4. **Diagramas de despliegue**: muestran un conjunto de nodos de procesamiento y los artefactos que residen en ellos. Normalmente, un nodo (hardware) suele albergar uno o más componentes. Muestran el hardware, el software y el *middleware* usados para conectar las máquinas.

Argot técnico



El vocablo **middleware**, como se puede deducir por analogía con los términos hardware y software, hace referencia a aquello que no está muy claro si se debe situar en el ámbito del hardware o en el del software. Se trata de un tipo de software que se sitúa entre el sistema operativo y las aplicaciones que se ejecutan en el ordenador. Gracias al *middleware* es posible la comunicación y la administración de datos en aplicaciones distribuidas.

5. **Diagramas de paquetes**: muestran la descomposición del modelo en unidades organizativas (paquetes) y sus dependencias. Sirven para simplificar diagramas de clases que sean complejos y permiten la agrupación de elementos estructurales en paquetes.

6. **Diagramas de perfiles:** permiten extender el lenguaje UML para su uso con una plataforma de programación en particular (como el *framework* .NET de Microsoft o la plataforma Java Enterprise Edition) o modelar sistemas para su uso en un dominio particular, como la medicina o los servicios financieros.
7. **Diagramas de estructura compuesta:** muestran la estructura interna, incluyendo partes y conectores, de un elemento estructural o de una colaboración. Son pues muy similares a los diagramas de componentes.

■ ■ ■ Diagramas de comportamiento

Estos diagramas sirven para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema. Se pueden distinguir siete tipos de diagramas de comportamiento, si bien los cuatro últimos que se muestran están agrupados bajo el subtipo de diagramas de interacción. Se explicarán en detalle estos diagramas en la Unidad 6.

1. **Diagramas de casos de uso:** ayudan a determinar la funcionalidad y las características del software desde el punto de vista del usuario. En estos diagramas se muestran casos de uso, actores y las relaciones entre ellos. Un **caso de uso** es un fragmento de funcionalidad del sistema que proporciona a la persona usuaria un resultado de interés, como por ejemplo dar de alta a un empleado. Por su parte, un **actor** es un conjunto coherente de roles que desempeñan los usuarios y usuarias de los casos de uso cuando interactúan con estos. Estos diagramas se estudiarán a fondo en el Apartado 6.2.
2. **Diagramas de actividades:** muestran el flujo paso a paso de una computación. Una actividad muestra un conjunto de acciones, el flujo entre ellas y los valores producidos o consumidos. Se emplean para especificar una operación compleja, un proceso de negocio o un flujo de trabajo. En el Apartado 6.5 se ofrece una descripción de estos diagramas.
3. **Diagramas de estados:** estos diagramas muestran una máquina de estados, que consta de estados, transiciones (pasos de un estado a otro), eventos (que provocan transiciones) y actividades (tareas que se llevan a cabo al realizar una transición). Son especialmente importantes en el modelado de una clase o de una colaboración con comportamiento significativo. Así pues, podría afirmarse que muestran el comportamiento dirigido por los eventos de un objeto. Se estudiarán estos diagramas en el Apartado 6.4.
4. **Diagramas de interacción:** son un grupo especial de diagramas de comportamiento que muestran una interacción, es decir, muestran un conjunto de objetos o roles y los mensajes que se envían entre ellos. Un mensaje es una solicitud de un objeto a otro (o a él mismo) para que ejecute el método indicado en el mensaje. Los objetos interactúan unos con otros para realizar colectivamente las tareas encomendadas a la aplicación. Se pueden distinguir **cuatro tipos** de diagramas de interacción:
 - **Diagramas de secuencia:** muestran la secuencia cronológica de los mensajes entre objetos durante un escenario de un caso de uso. Sirven para mostrar cómo interactúan unos objetos con otros para ejecutar lo especificado en un escenario de un caso de uso (para una descripción más detallada véase el Apartado 6.3.1).

- **Diagramas de colaboración:** muestran un conjunto de objetos, enlaces entre ellos y los mensajes enviados y recibidos entre ellos. Los diagramas de colaboración muestran la misma información que los diagramas de secuencia, pero mientras que estos resaltan la ordenación temporal, los de colaboración resaltan la estructura de datos a través de la cual fluyen los mensajes (véase el Apartado 6.3.2 para una explicación más amplia).
- **Diagramas de tiempos:** muestran los tiempos reales en la interacción entre diferentes objetos y roles. O dicho de otra forma, en ellos se representa el comportamiento de los objetos en un determinado periodo de tiempo.
- **Diagrama global de interacciones:** aportan una visión general del flujo de control de las interacciones. Es un tipo de diagrama híbrido entre el diagrama de secuencia y el diagrama de actividad. El empleo de este diagrama es útil cuando el modelo de interacción tiene un gran tamaño, ya que este ofrece una visión global de la interacción.

El lenguaje UML es independiente de la metodología de análisis y diseño que se siga en el desarrollo de software, si bien utiliza una perspectiva orientada a objetos.

5.3. Clases, atributos, métodos y visibilidad

Un objeto es una instancia de una clase, de modo que una clase está formada por un conjunto de objetos que poseen la misma estructura y el mismo comportamiento. Por ejemplo, la clase *Libro* representa cualquier libro de una biblioteca, siendo un objeto cada uno de los ejemplares de libros disponibles en la biblioteca. Por otro lado, la clase *Cuenta* representa cualquier cuenta bancaria, siendo un objeto cada una de las cuentas de los clientes del banco.

Una clase tiene tanto una serie de atributos como una serie de métodos, comunes a todos los objetos de la clase:

- **Los atributos** son las propiedades que posee una clase. Así, la clase *Libro* podría tener los atributos: código, ISBN, título, editorial y prestado. Cada atributo tendrá un nombre, un tipo de dato (numérico entero, numérico real, cadena de caracteres, etc.) y, posiblemente, un valor por defecto. Cada objeto de la clase *Libro* tendrá estos mismos atributos y cada uno de estos atributos tendrá un valor asignado. La clase *Cuenta* podría tener los atributos número de cuenta y saldo.
- **Los métodos** son las operaciones que se pueden llevar a cabo con los objetos de la clase y definen el **comportamiento** de la clase. Así, por ejemplo, la clase *Libro* podría disponer de los métodos *prestar* y *devolver*, que deben ser invocados cuando un usuario o una usuaria pide prestado un libro de la biblioteca y devuelve un libro que le fue prestado, respectivamente. Por otro lado, la clase *Cuenta* podría disponer de los métodos *ingresarDinero* y *extraerDinero*, que deben ser invocados cuando un cliente ingrese dinero en la cuenta o extraiga dinero de ella, respectivamente. Estos dos métodos recibirán como argumento o parámetro el importe que se desea ingresar y extraer, respectivamente.

La información acerca de las clases se suele representar mediante los llamados diagramas de clases siguiendo la notación UML. Cada clase se representa mediante un rectángulo con tres secciones:

- En la superior se indica el nombre de la clase.
- En la central se coloca información sobre los atributos de la clase.
- En la inferior se coloca información sobre los métodos de la clase.

Sin embargo, la información que se incluye en las secciones correspondientes a los atributos y a los métodos puede diferir dependiendo de lo avanzado que se encuentre el proceso de análisis y diseño del software. Así, inicialmente, puede ser suficiente con indicar únicamente los nombres de los atributos y los nombres de los métodos, como se muestra en la Figura 5.14 para las clases *Libro* y *Cuenta*.

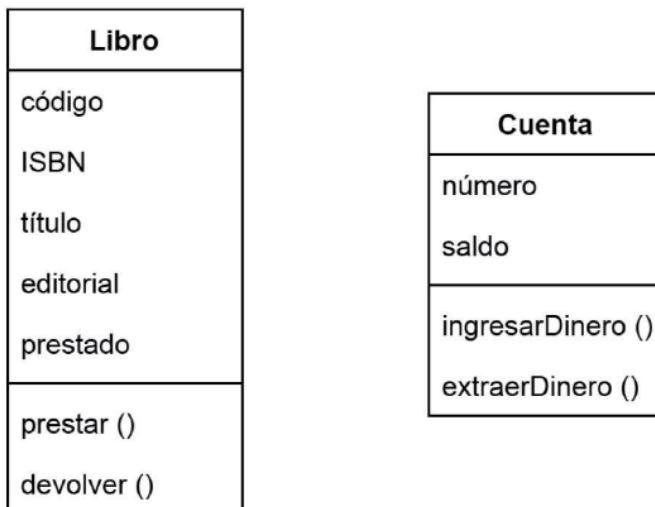


Figura 5.14. Representación de las clases *Libro* (con los atributos código, ISBN, título, editorial y prestado y los métodos prestar y devolver) y *Cuenta* (con los atributos número y saldo y los métodos ingresarDinero y extraerDinero).

No obstante, a medida que se avanza en el proceso de desarrollo de software, se puede añadir más información a cada clase. Así, se pueden detallar más los diagramas de clases UML, indicando para cada atributo de una clase un **modificador de acceso** y el **tipo de dato** asociado a este, escribiendo:

modificador atributo: tipoDatos

El modificador de acceso hace referencia a la visibilidad del atributo, es decir, al ámbito desde el cual es visible el atributo en cuestión. Los modificadores de acceso más importantes son:

- **Private:** si un atributo o un método de una clase se define con el modificador de acceso *private*, quiere decir que ese atributo o método solo es accesible desde métodos de la clase en la que está declarado el atributo o método. Este modificador se simboliza con un guion (-).

- **Public:** si un atributo o un método de una clase se define con el modificador de acceso *public*, quiere decir que ese atributo o método es accesible desde métodos de cualquier clase. Este modificador se simboliza mediante el signo más (+).
- **Protected:** este modificador de acceso indica que el atributo o el método que lo tenga asignado es accesible desde métodos de la propia clase y desde métodos de su subclase o de sus subclases. Este modificador se simboliza por medio del símbolo almohadilla (#).
- **Package:** indica que el atributo o el método es visible en las clases incluidas en el mismo paquete. Este modificador se simboliza con el carácter tilde (~).

Lo habitual es, al menos en las fases iniciales del proceso de desarrollo, asignar a los atributos el modificador de acceso *private*, y a los métodos, el modificador *public*, si bien este último se puede cambiar posteriormente a *private* si se detecta que no es necesario el uso del método por parte de otras clases.

En cuanto a los tipos de datos, se pueden utilizar los **tipos básicos de datos de UML**, que son los siguientes:

- *int* o *integer*: hace referencia a un número entero, es decir, sin decimales.
- *String*: hace referencia a cadenas de caracteres.
- *boolean*: permite almacenar solo los valores *true* (verdadero) o *false* (falso).

También, es posible usar los **tipos de datos de lenguajes orientados a objetos**, como Java, a saber:

- *float* o *double*: hace referencia a un número real, es decir, con decimales o bien un número que permite un rango de valores mayor que el tipo *int*.
- *char*: permite almacenar un solo carácter, que puede ser una letra, un dígito, un carácter especial (*, @, \$, :, etc.) o una secuencia de escape (como, por ejemplo, el carácter de nueva línea, \n).
- *Date*: permite almacenar una fecha.
- *Time*: permite almacenar una hora.
- *DateTime*: permite almacenar una fecha y una hora conjuntamente.

Además, en el caso de los métodos, se puede indicar el tipo del valor devuelto, si es que el método devuelve algún valor, y para cada uno de los datos que recibe el método (parámetro), el nombre del parámetro y su tipo de dato, siguiendo la siguiente sintaxis:

```
modificador método(parám1: tipoDato1, parám2: tipoDato2, ...): tipoDatoDevuelto
```

Así, se podría indicar que el atributo *código* de *Libro* es de tipo entero (*int*), que los atributos *ISBN*, *título* y *editorial* son de tipo cadena de caracteres (*String*) y que el atributo *prestado* es de tipo booleano (*boolean*), dado que un libro solo puede estar prestado (que se representa con valor *true* para el atributo *prestado*) o no prestado (que se representa con

valor *false* para este atributo). Por otro lado, para los métodos *prestar* y *devolver*, podría indicarse que devuelven un valor de tipo *boolean*, que informaría si el préstamo o la devolución ha tenido lugar o no, respectivamente.

En el caso de la clase *Cuenta*, podría indicarse que su atributo *número* es de tipo cadena de caracteres (*String*) y que su atributo *saldo* es de tipo número real (*float*). En el caso de los métodos *ingresarDinero* y *extraerDinero*, estos no devuelven nada, pero ambos recibirán un parámetro, que se puede llamar *importe*, que debe ser de tipo número real (*float*).

Se podría entonces representar las clases *Libro* y *Cuenta* en un diagrama UML con la información antes expuesta tal y como se muestra en la Figura 5.15.

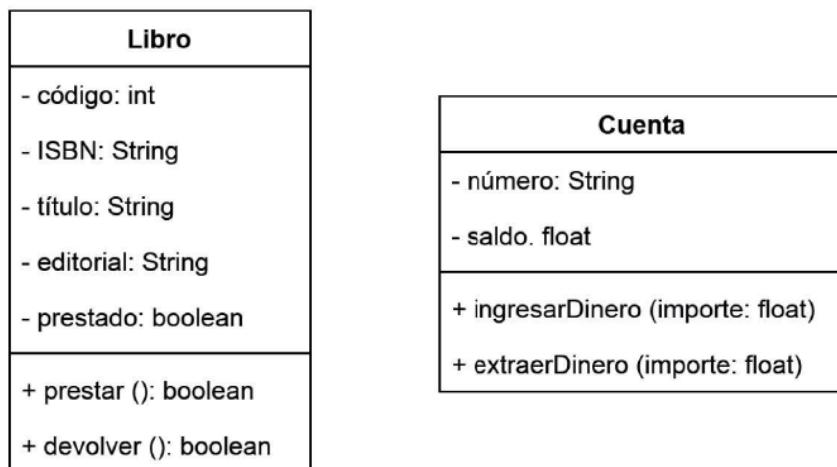


Figura 5.15. Representación detallada de las clases *Libro* y *Cuenta*, indicando la visibilidad de cada atributo y método. En el caso de los atributos, se indica, además, su tipo de dato, y en el caso de los métodos, el tipo de dato devuelto, si lo hay, y por cada parámetro, si es el caso, su nombre y tipo de dato.

■ 5.4. Relaciones entre clases

Para todos los diagramas de clases, se han de establecer relaciones entre las clases, de manera similar a como se establecen relaciones entre las entidades de un diagrama entidad-relación. Existen diferentes tipos de relaciones entre clases, que se van a exponer a continuación.

■ ■ 5.4.1. Agregación

Una agregación es un tipo de relación entre clases que representa un objeto compuesto por otros objetos. En esta relación, se distingue la parte que representa el *todo*, que recibe el nombre de **compuesto**, y las diferentes *partes* que lo componen, que reciben el nombre de **componente**.

La agregación implica que solo puede existir una instancia del objeto agregado si existe al menos una instancia relacionada de alguno de los componentes. Por ejemplo, no tiene sentido hablar de un plan de estudios si no tiene al menos una asignatura.

Existen dos tipos de agregaciones: la **agregación débil**, que se conoce generalmente como **agregación «a secas»** y la **agregación fuerte**, conocida como **composición**.

En el caso de la agregación débil, o simplemente agregación, como indican Alonso, Martínez y Segovia (2005):

el tiempo de vida de los componentes es diferente que el del agregado. En otras palabras, los objetos de las clases componentes pueden existir antes de crear el agregado relacionado e incluso después de desaparecer el agregado. Por ejemplo, si se especifica que un cuadro puede estar compuesto de un marco, un cristal y una lámina, es perfectamente válido suponer que el marco puede existir antes de hacer el cuadro y no tiene que dejar de existir al destruir el cuadro.

Las relaciones de agregación se representan uniendo el compuesto con sus partes componentes mediante una línea colocando un rombo con fondo blanco al lado del compuesto, como se muestra en la Figura 5.16.

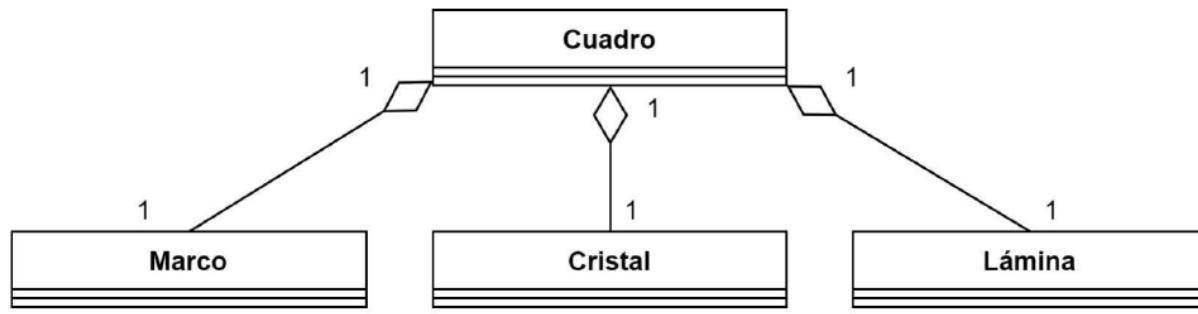


Figura 5.16. Diagrama UML que muestra una relación de agregación entre la clase de tipo compuesto Cuadro y sus partes o componentes Marco, Cristal y Lámina.

Los números 1 al lado de las clases reciben el nombre de **multiplicidades o cardinalidades** e indican el número de objetos de una clase que pueden estar vinculados con cada objeto de la otra clase. Las cardinalidades en un diagrama UML pueden ser:

- **1:** uno.
- **0..1:** cero o uno.
- **0..*:** cero o varios. También se puede representar solo con el símbolo del asterisco (*).
- **1..*:** uno o varios.
- **N:** el número indicado.
- **N..M:** entre los números N y M.

Así, el 1 al lado de la clase *Marco* se debe interpretar como que un cuadro posee un solo marco, el 1 al lado *Cristal* se debe interpretar como que un cuadro posee un único cristal, y el 1 al lado de *Lámina*, como que un cuadro posee una única lámina. El 1 situado al lado de *Cuadro* y del rombo de la línea que representa su relación con *Marco* debe interpretarse como que un marco pertenece a un solo cuadro y lo mismo ocurre con *Cristal* y *Lámina*.

En la Figura 5.17, se muestra otro ejemplo de agregación, en el que se indica que un plato se elabora con un conjunto de varios ingredientes. El $1..*$ al lado de *Ingrediente* indica que un plato consta de uno o varios ingredientes. El $*$ al lado de *Plato* indica que un ingrediente puede estar presente en ninguno o en varios platos. Se puede asignar o no un nombre a cualquier agregación, en cuyo caso, este nombre se debe colocar al lado de la línea que une las dos clases. Así, en el diagrama UML que se observa en la Figura 5.17, se ha asignado el nombre *se elabora con* a la agregación, aunque no sea obligatorio.

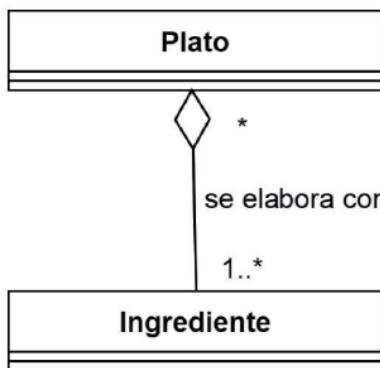


Figura 5.17. Diagrama UML que muestra una relación de agregación entre la clase de tipo compuesto Plato y sus ingredientes.

■ ■ ■ 5.4.2. Composición

La agregación fuerte o composición es un tipo especial de agregación en la que la multiplicidad al lado de la clase que representa el compuesto es siempre 1 y en la que la existencia de los componentes depende del compuesto. De esto se puede deducir que, a diferencia de la agregación, la composición impone dos restricciones:

1. Cada componente solo puede estar presente en un compuesto.
2. Si se elimina el compuesto, hay que eliminar todos los componentes vinculados a él.

Las relaciones de composición se representan como las de agregación, pero el rombo que se coloca al lado del compuesto tiene el fondo de color negro en lugar de blanco. Se puede asignar o no un nombre a la composición, en cuyo caso este nombre se debe colocar al lado de la línea que une las dos clases.

Un ejemplo de composición es el que se da entre un plan de estudios y las asignaturas de que consta dicho plan. Una asignatura solo puede pertenecer a un plan de estudios, y cuando desaparece este plan, desaparecen con él todas sus asignaturas. En la Figura 5.18, se representa la relación de composición de dicho ejemplo.

Otro ejemplo de composición (Figura 5.19) sería aquel en el que se relaciona una ventana de una interfaz gráfica de usuario con sus partes, de tal manera que toda ventana consta de un único panel, puede tener o no una cabecera y dispone de dos barras de desplazamiento (una por la izquierda y otra por la derecha).

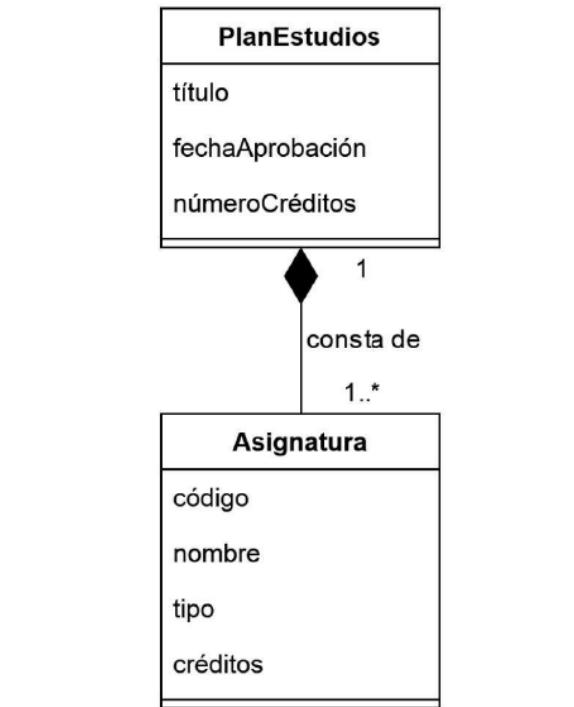


Figura 5.18. Diagrama UML que muestra una relación de composición entre la clase 'compuesto' PlanEstudios y sus asignaturas.

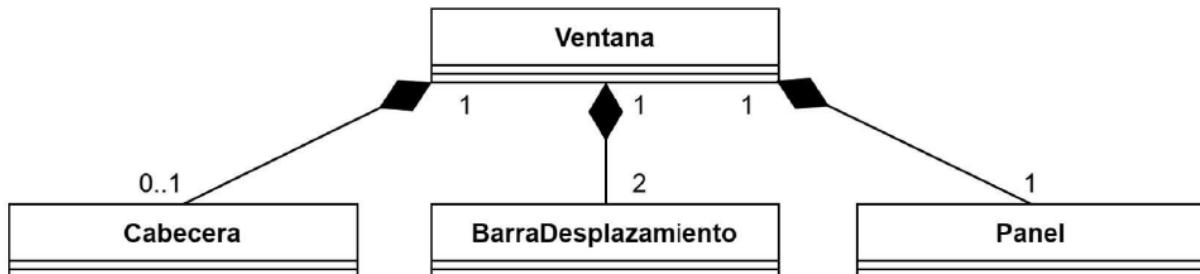


Figura 5.19. Diagrama UML que muestra una relación de composición entre la clase 'compuesto' Ventana y sus componentes Cabecera, BarraDesplazamiento y Panel.

Actividad resuelta 5.1

Agregación o composición

En un programa, se quiere manejar información sobre países y las regiones en que se dividen esos países. Como sabes, un país consta de varias regiones y una región pertenece a un único país. Si se dispone de las clases *País* y *Región*, ¿Qué tipo de relación (agregación o composición) se debe establecer entre estas dos clases?, ¿por qué?

Solución

Entre las clases *País* y *Región* se debe establecer una relación de tipo composición porque un país es un compuesto que consta de varios componentes (regiones) y no pueden existir regiones sin un país asociado. Además, se cumplen las dos restricciones que impone una composición respecto a una agregación:

1. Una región solo puede pertenecer a un país.
2. Si se elimina un país porque, por ejemplo, ya no se va a trabajar con información sobre ese país en el programa, se deben eliminar también todas las regiones pertenecientes a dicho país.

5.4.3. Generalización y especialización

Las relaciones de generalización-especialización ocurren cuando se establece una jerarquía de clases y subclases motivada por el hecho de que hay varias clases que poseen atributos y/o métodos comunes. En este caso, se deben asignar los atributos y métodos comunes a la superclase, dejando los atributos y métodos específicos en las subclases. En este contexto, se habla de **generalización** porque se puede decir que las subclases se generalizan en una superclase, dado que los atributos y métodos comunes a las subclases se asignan a la superclase. Por otro lado, se habla de **especialización** porque la superclase se especializa en una o varias subclases, las cuales poseen atributos y/o métodos específicos además de los de la superclase.

Así, si en una aplicación se quisieran contemplar cuentas corrientes y cuentas de ahorro, se debería tener en cuenta que toda cuenta corriente tiene un número de cuenta, un saldo y un saldo medio; y, por otro lado, toda cuenta de ahorro tiene un número de cuenta, un saldo y un tipo de interés. En este caso, se creará una superclase llamada *Cuenta* que englobe los atributos comunes a los dos tipos de cuentas (número de cuenta y saldo), de manera que *CuentaCorriente* y *CuentaDeAhorro* serán las subclases con los atributos específicos (saldo medio y tipo de interés, respectivamente).

Cabe afirmar, por tanto, que la clase *Cuenta* es una generalización de las subclases *CuentaCorriente* y *CuentaDeAhorro*, y que *CuentaCorriente* es una especialización de la superclase *Cuenta* porque una cuenta corriente es algo más especializado que una cuenta, pues posee todas las características de una cuenta, pero también una característica adicional, que es el saldo medio.

Este tipo de relaciones se representan uniendo mediante una línea cada subclase con su superclase y esta línea lleva en el extremo un triángulo apuntando a la superclase, como se puede observar en la Figura 5.20.

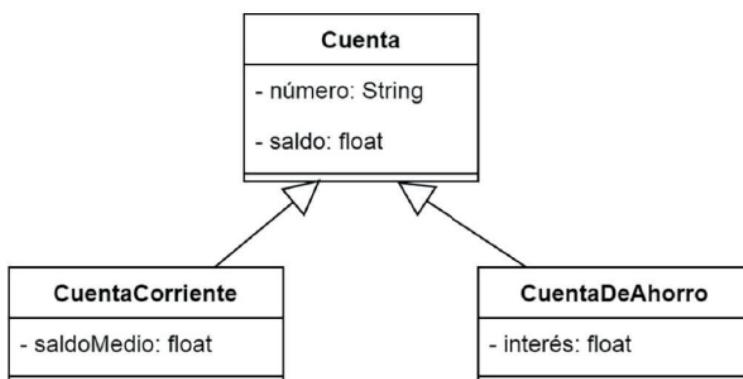


Figura 5.20. Diagrama UML que muestra una relación de generalización-especialización entre la superclase Clase y sus subclases CuentaCorriente y CuentaDeAhorro.

La herencia hace referencia al hecho de que las superclases poseen no solo sus atributos o métodos propios, sino que también heredan los de la superclase correspondiente. Así, la subclase *CuentaCorriente* tiene un saldo medio (atributo propio), pero también tiene un número de cuenta y un saldo, atributos que hereda de la superclase *Cuenta*.

Argot técnico



El hecho de emplear el término **herencia** para referirse al hecho de que las subclases poseen, además de sus atributos y métodos propios, los de la superclase con la que se vinculan, está directamente relacionado con el hecho de que a la superclase se le llame también **clase padre**, y a las subclases, **clases hijas**.

5.4.4. Asociación

Entre las clases pueden existir relaciones genéricas, que reciben el nombre de **asociaciones**. Cabe definir las **asociaciones** como relaciones entre clases del dominio del problema que no tienen las características de la agregación ni de la relación de generalización-especialización. También es común referirse a las asociaciones con el nombre genérico de **relación**, por lo que se emplearán ambos términos indistintamente.

Las asociaciones pueden ser de diferentes tipos en función del número de clases que vinculan. El número de clases que se vinculan por medio de una asociación recibe el nombre de **grado de la relación**. En función de su grado, existen los siguientes tipos de relaciones:

- **Reflexivas:** son las relaciones de grado 1, es decir, aquellas que vinculan a una clase consigo misma.
- **Binarias:** son las relaciones de grado 2, esto es, aquellas que vinculan dos clases.
- **Ternarias, cuaternarias...:** son las relaciones de grado 3, 4..., es decir, aquellas que vinculan tres, cuatro... clases, respectivamente.

Toda asociación se representa mediante una línea que une las clases vinculadas y lo más habitual es asociarle un nombre, que suele ser un verbo y se debe escribir sobre dicha línea.

Una asociación reflexiva se representa por medio de una línea que une una clase consigo misma. Por ejemplo, en una aplicación puede haber una clase *Empleado* y se podría querer reflejar la jerarquía de la plantilla de la empresa, esto es, se podría querer saber, por cada persona empleada, qué empleado o empleada es su jefe directo, que será solamente uno, en caso de tenerlo. Para determinar las cardinalidades o multiplicidades, habría que plantearse las siguientes cuestiones:

- Un empleado es jefe de ninguno o varios empleados, por lo que una de las cardinalidades será 0..* o *. Esto quiere decir que un empleado tiene cero o muchos subordinados.

- Un empleado tiene como jefe a ningún empleado (si es el máximo responsable de la empresa) o a solo uno (su jefe directo), por lo que la otra cardinalidad será 0..1.

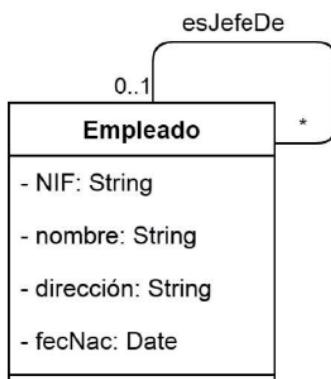


Figura 5.21. Diagrama UML que muestra una relación reflexiva de Empleado, la cual refleja el jefe directo de cada persona empleada, en caso de que lo tenga.

Una relación binaria se representa por medio de una línea que une las dos clases. Por ejemplo, puede haber una relación entre *Cliente* y *Pedido*, que indique por cada cliente los pedidos que este ha realizado. En este caso, para la determinación de las cardinalidades, habría que plantearse lo siguiente:

- Un cliente puede realizar uno o varios pedidos, por lo que se pondrá 1..* al lado de la clase *Pedido*.
- Un pedido es realizado por un único cliente, por lo que se escribirá 1 al lado de la clase *Cliente*.

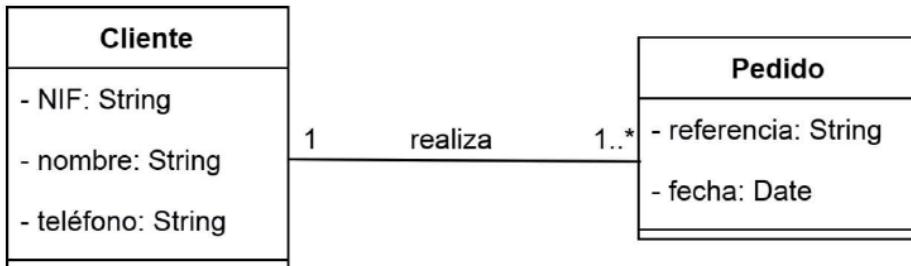


Figura 5.22. Diagrama UML que muestra una asociación binaria entre Cliente y Pedido, que refleja los pedidos que realiza cada cliente.

Para las asociaciones, agregaciones y composiciones, también se puede indicar su **navegabilidad**, esto es, el sentido en el que se puede acceder a información desde una de las clases intervenientes en la asociación. La navegabilidad se representa mediante una punta de flecha que indica en qué sentido es navegable la relación. Así, en el diagrama que se muestra en la Figura 5.23, en el que se ha dibujado una punta de flecha al lado de la clase *Pedido*, se indica que la relación es navegable de *Cliente* a *Pedido*, lo que quiere decir que a partir de un cliente se puede acceder a todos sus pedidos, pero no es posible saber por cada pedido qué cliente lo ha realizado.



Figura 5.23. Diagrama UML que muestra una asociación binaria entre Cliente y Pedido navegable solo de Cliente a Pedido, lo que quiere decir que a partir de un cliente se puede acceder a todos los pedidos que este ha realizado.

El código que se genera a partir de este diagrama de clases debe ser similar al que se muestra a continuación, en el que, como se puede observar, en la clase *Cliente* se ha creado un atributo de tipo *ArrayList* para registrar sus pedidos.

```

1 public class Cliente {
2     private String NIF;
3     private String nombre;
4     private String telefono;
5     private List<Pedido> pedidos = new ArrayList<Pedido> ();
6 ...
7 }
8 public class Pedido {
9     private String referencia;
10    private Date fecha;
11 ...
12 }
  
```

Si la punta de flecha se hubiese colocado solo al lado de *Cliente*, querría decir que, a partir de un pedido, se puede saber el cliente que lo ha realizado, pero no es posible conocer para un cliente todos los pedidos que ha solicitado. En este caso, se incluiría en la clase *Pedido* un atributo de tipo objeto de la clase *Cliente*, como se puede observar en el siguiente código.

```

1 public class Cliente {
2     private String NIF;
3     private String nombre;
4     private String telefono;
5 ...
6 }
7 public class Pedido {
8     private String referencia;
9     private Date fecha;
10    private Cliente cliente;
11 ...
12 }
  
```

Si se dibujaran puntas de flecha tanto al lado de *Cliente* como al lado de *Pedido*, entonces la relación sería navegable en los dos sentidos, y en el código generado, aparecería en *Cliente* el atributo que indica el conjunto de pedidos que ha realizado, y en *Pedido*, el atributo que indica el cliente que ha realizado el pedido.

No todas las herramientas emplean la misma notación para representar la navegabilidad. Así, en algunas de ellas, la ausencia de punta de flecha en las dos partes indica que la relación es navegable en los dos sentidos. En la mayoría de los diagramas que se muestran a partir de aquí no hay dibujadas puntas de flecha, salvo que la navegabilidad de la relación correspondiente sea relevante.

Las relaciones de grado mayor que 2 (ternarias, cuaternarias, etc.) se representan mediante un rombo que une a las clases relacionadas. Sería posible tener, por ejemplo, una relación ternaria entre las clases *Atleta*, *Prueba* y *Fecha*, que indica, para los participantes en una competición de atletismo, qué día o días han participado en cada prueba. Para la determinación de las cardinalidades en el caso de las relaciones ternarias, se debe tomar un objeto de cada dos clases y sería necesario plantearse con cuántos objetos de la otra clase como mínimo y como máximo puede estar relacionado ese par de objetos. En este caso, habría que plantearse lo siguiente:

- Dados un atleta y una prueba, ese atleta en esa prueba ha podido participar ningún día o varios días, por lo que se indicará la cardinalidad $0..*$ o $*$ al lado de *Fecha*.
- Dados un atleta y una fecha, ese atleta ese día puede haber competido en ninguna o varias pruebas, por lo que se indicará la cardinalidad $0..*$ al lado de *Prueba*.
- Dadas una prueba y una fecha, ese día en esa prueba han podido participar ninguno o varios atletas, por lo que se indicará la cardinalidad $0..*$ al lado de *Atleta*.

En la Figura 5.24 se muestra el diagrama UML que representa la relación ternaria arriba descrita.

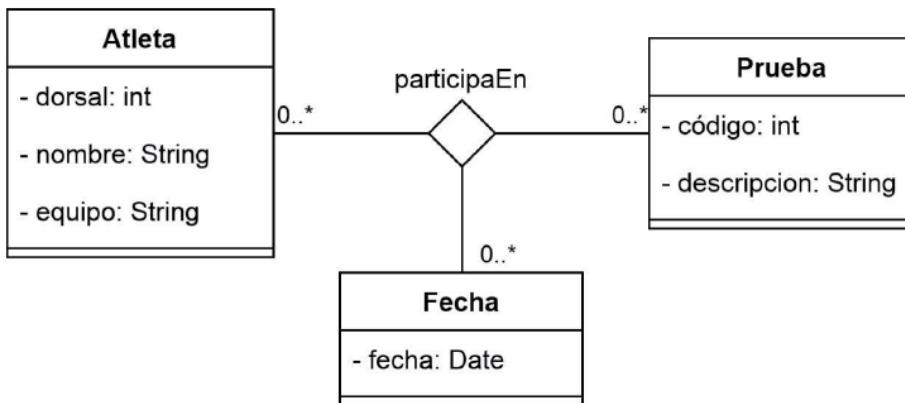


Figura 5.24. Diagrama UML que muestra una asociación ternaria entre Prueba, Atleta y Fecha, que refleja el día o los días en los que participa cada atleta en cada prueba.

Hay algunas ocasiones en las que es necesario almacenar información acerca de las asociaciones. La manera de reflejar esto es por medio de la creación de lo que se conoce como una **clase asociativa**. A modo de ilustración, aquí se describe la relación binaria

entre las clases *Pedido* y *Artículo*. Un pedido tiene como atributos su referencia y fecha, y un artículo, su código, descripción y precio (PVP). En un pedido, se pueden solicitar uno o varios artículos y un artículo puede ser solicitado en ninguno o en varios pedidos. Esto se representa mediante el diagrama UML que se muestra en la Figura 5.25.



Figura 5.25. Diagrama UML que muestra una asociación binaria entre Pedido y Artículo, que refleja el artículo o los artículos solicitados en cada pedido.

Actividad propuesta 5.1

Diagrama de clases para una cadena de supermercados I

Crea un diagrama de clases que refleje la información necesaria para gestionar una cadena de supermercados, de acuerdo con los siguientes requisitos:

- La cadena dispone de varios supermercados y, para cada uno de estos, se desea registrar su código, nombre, dirección, número de teléfono y la ciudad donde está ubicado.
- Cada ciudad tiene asignado un código, su nombre y el nombre de la provincia a la que pertenece.
- Se sabe que en una ciudad puede haber varios supermercados de la cadena.

Cuando un cliente realiza un pedido, este puede solicitar un número diferente de unidades para cada uno de los artículos incluidos en dicho pedido, por ejemplo, en un pedido podría solicitar 5 bolígrafos azules de la marca X y 20 gomas de borrar de la marca Y. Este atributo, que hace referencia al número de unidades de cada artículo que se solicita en cada pedido, y que se llamará *cantidad*, ¿dónde se sitúa: en la clase *Pedido* o en la clase *Artículo*? No tiene sentido colocarlo en la clase *Pedido* porque en un pedido se pueden solicitar cantidades distintas por cada uno de los artículos incluidos en él. Así, en este ejemplo, se solicitan dos cantidades: 5 (bolígrafos) y 20 (gomas de borrar). Asimismo, no es posible en un objeto de una clase asignar varios valores al mismo atributo. Tampoco tiene sentido colocar este atributo en la clase *Artículo* porque un mismo artículo se puede solicitar en varios pedidos y la cantidad que se solicita de dicho artículo en cada pedido puede ser diferente. Por ejemplo, la goma de borrar de la marca Y puede estar incluida en tres pedidos: en uno de ellos, se han podido solicitar 20 unidades; en otro, 12; y en otro, 25.

Así pues, realmente, el atributo *cantidad* no proporciona información acerca de un pedido ni acerca de un artículo, sino acerca de la relación entre pedido y artículo. O dicho de otra forma, el atributo *cantidad* nos indica por cada pareja de pedido-artículo, el número

de unidades de ese artículo que se solicitan en ese pedido. La manera de reflejar esta información es creando una clase asociativa vinculada a la relación entre *Pedido* y *Artículo*. Se podría llamar a esta clase *LíneaPedido*, que en la vida real hace referencia a la parte de cada pedido correspondiente a cada artículo solicitado en dicho pedido. Teniendo esto en cuenta, en esta clase asociativa se colocará el atributo *cantidad*, que indica el número de unidades que se solicitan de cada artículo en cada pedido. Una clase asociativa se representa como una clase normal, pero se une mediante una línea discontinua a la asociación a la que se vincula, como se muestra en la Figura 5.26.

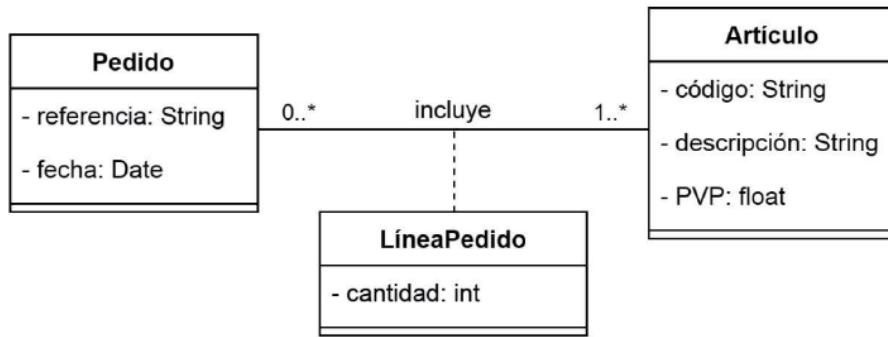


Figura 5.26. Diagrama UML que muestra, además de una asociación binaria entre Pedido y Artículo, una clase asociativa, a la que se ha llamado LíneaPedido. Esta indica, por artículo solicitado en cada pedido, la cantidad o número de unidades que de ese artículo se han solicitado en ese pedido.

Recuerda



Si se establece una analogía entre los diagramas de clases y los **diagramas entidad-relación**, que se emplean para representar conceptualmente la información que se debe almacenar en una base de datos, se podría decir que una clase asociativa de un diagrama de clases es un constructo que se crea para albergar lo que en un diagrama entidad-relación serían los atributos de una relación. De esta forma, una clase asociativa contiene los atributos que corresponderían a una relación en un diagrama entidad-relación.

Actividad propuesta 5.2

Diagrama de clases para una cadena de supermercados II

Amplía el diagrama de clases de la Actividad propuesta 5.1 teniendo en cuenta que ahora, por cada supermercado, además se desean conocer los artículos que tiene a la venta. Para ello, se deben considerar los siguientes requisitos:

- Por cada artículo, se desea registrar su código, descripción, precio de venta al público (PVP) y stock o existencias que hay de ese artículo en cada supermercado.
 - No es necesario que en todos los supermercados se vendan todos los artículos.
 - El PVP de un artículo determinado puede diferir de un supermercado a otro, esto es, un mismo artículo no tiene por qué venderse al mismo precio en todos los supermercados.

También se pueden vincular clases asociativas a relaciones de grado mayor que 2 (ternarias, cuaternarias, etc.). Si, por ejemplo, para el diagrama UML de la Figura 5.24, se

quisiera almacenar por cada día en el que un atleta participa en una prueba, el resultado obtenido por ese atleta ese día en esa prueba, habría que vincular una clase asociativa a la relación ternaria con el atributo *resultado* de tipo *String*. En este atributo, se almacenaría el tiempo que ha empleado en la carrera, la longitud saltada, la distancia a la que ha llegado el elemento lanzado, etc. La Figura 5.27 muestra el diagrama UML con la clase asociativa.

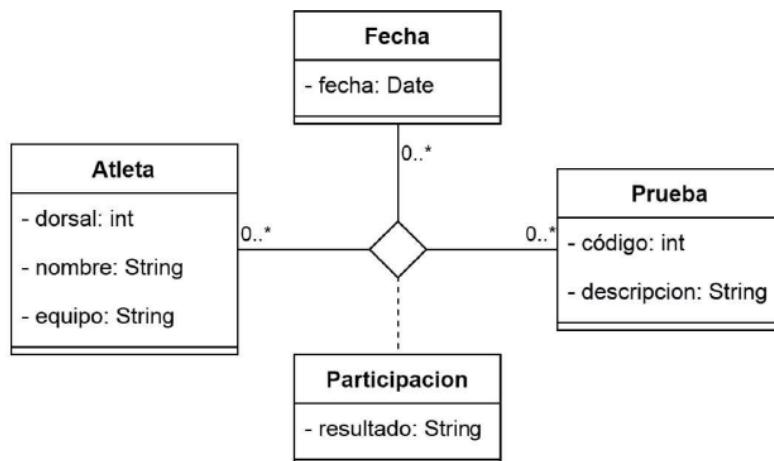


Figura 5.27. Diagrama UML que muestra una asociación ternaria entre Prueba, Atleta y Fecha, que refleja el día o los días que participa cada atleta en cada prueba. Además, se ha vinculado una clase asociativa a dicha relación ternaria con el objetivo de almacenar el resultado obtenido por cada atleta en cada prueba en la que ha participado.

Actividad propuesta 5.3

Diagrama de clases para una cadena de supermercados III

Amplía el diagrama de clases de la Actividad propuesta 5.2, teniendo en cuenta que se desea manejar información sobre los proveedores que suministran artículos a los supermercados. Para ello, se debe tener presente que:

- Por cada proveedor, se desea registrar su número de identificación, nombre, dirección y qué artículos suministra a cada supermercado, dado que un proveedor no tiene por qué suministrar los mismos artículos a todos los supermercados de la cadena.
 - Además, un mismo artículo lo puede vender un proveedor a diferentes precios en función del supermercado al que se lo suministra.
 - Se quiere tener constancia de qué artículos suministra cada proveedor a cada supermercado y a qué precio. Este se conoce habitualmente como *precio de venta de distribución* (PVD).

Actividad resuelta 5.2

Relación reflexiva con clase asociativa vinculada

En una empresa fabricante de vehículos, se precisa que esta disponga de información sobre las piezas necesarias para fabricar cada tipo de vehículo. Para ello, al desarrollar la aplicación se necesita crear un diagrama de clases. Se sabe que:

- Cada tipo de vehículo consta de varias piezas (carrocería, motor, ruedas, etc.) y cada una de estas piezas se puede descomponer en piezas más pequeñas, y así sucesivamente, hasta llegar a piezas indivisibles, como por ejemplo, un espejo o un tornillo.
- Se consideran piezas todos los componentes de un tipo de vehículo independientemente de su tamaño, por lo que se considera que un vehículo es una pieza en sí misma, pero también lo es un simple tornillo.
- Dada una pieza, puede que esta no forme parte de ninguna pieza superior (tal es el caso de un vehículo) o que forme parte de varias piezas superiores.

Se desea conocer:

- Por cada pieza, su código, nombre, existencias, existencias mínimas deseables, en qué pieza o piezas está incluida (si es el caso) y de qué pieza o piezas consta (si es el caso).
- Por cada pieza que consta de piezas inferiores, el número de unidades de cada pieza inferior que incluye cada pieza superior; así, no solo interesa saber que un determinado tipo de vehículo consta de motor y de ruedas, sino también cuántos motores tiene ese tipo de vehículo (uno) y cuántas ruedas (4, por ejemplo).

Solución

Para poder manejar toda esa información, se debe crear una clase *Pieza*, y para reflejar la jerarquía de piezas, se debe establecer una relación reflexiva. Así pues, con el objeto de determinar las cardinalidades o multiplicidades de esta relación, hay que plantearse las siguientes cuestiones:

- Una pieza se descompone o divide en ninguna o varias piezas, por lo que una de las cardinalidades es 0..* o *. Las piezas indivisibles no se dividen en piezas más pequeñas.
- Una pieza está incluida en ninguna pieza (si la pieza es un tipo de vehículo) o en varias, por lo que la otra cardinalidad también es 0..* o *.

Por otro lado, para saber el número de unidades de cada subpieza en que se divide cada pieza, se necesita vincular, a la relación reflexiva, una clase asociativa albergando ese número, que indicará, por cada pareja de pieza-subpieza, el número de subpiezas que contiene cada pieza (véase la Figura 5.28).

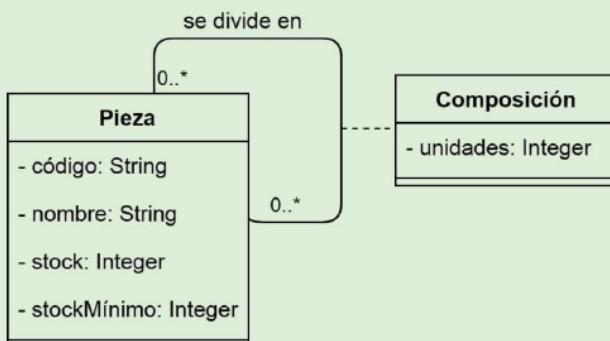


Figura 5.28. Diagrama UML que muestra una relación reflexiva de la clase *Pieza*, necesaria para conocer la relación jerárquica existente entre las piezas. Se precisa de una clase asociativa para conocer el número de subpiezas que contiene cada pieza.

5.4.5. Realización

Una relación de realización es la que se establece entre una clase *Interface* y las clases que implementan esa interfaz. Una clase *Interface* es un mecanismo que se puede emplear en Java para permitir la herencia múltiple, es decir, que una clase herede de varias superclases. El motivo es que, en Java, una clase solo puede heredar de una superclase, pero puede implementar una o varias interfaces.

Las interfaces suelen contener métodos comunes a varias clases, de forma que todas las clases que implementan una interfaz o que establecen una relación de realización con una interfaz, *heredan* de dicha interfaz todos sus métodos.

Por ejemplo, se podría tener las clases *Figura* y *Fracción* que implementan la interfaz *Comparar* que incluye un método *esMayorQue()*, el cual permite determinar si una figura es mayor que otra (tiene un área mayor que otra) o si una fracción es mayor que otra. A su vez, la clase *Figura* tiene tres subclases: *Rectángulo*, *Triángulo* y *Círculo*. Una relación de realización se representa gráficamente uniendo mediante una línea con trazo discontinuo cada clase con la interfaz que implementa y esta línea tiene en el extremo un triángulo apuntando a la interfaz, como se puede observar en la Figura 5.29.

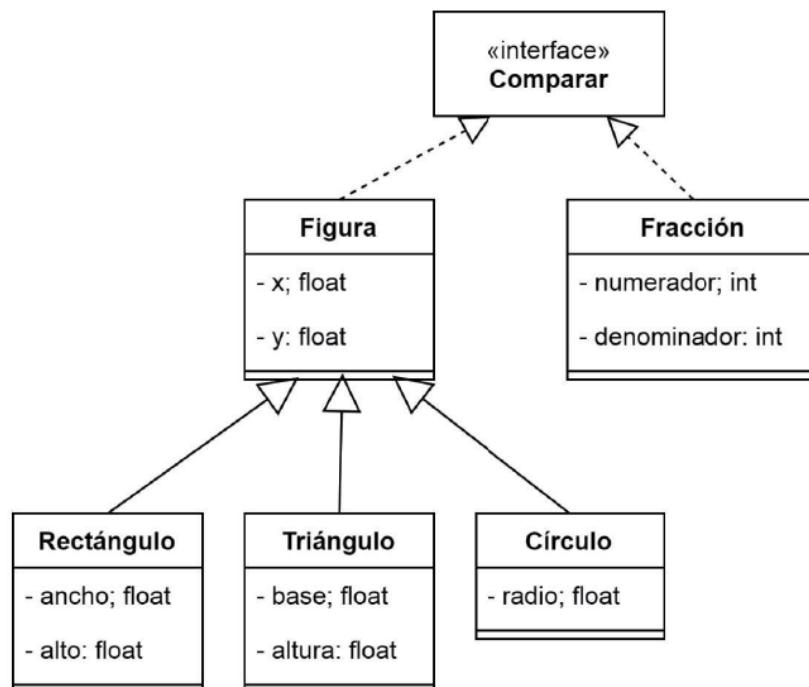


Figura 5.29. Diagrama UML que muestra una relación de realización entre Figura y la interfaz Comparar y otra relación de realización entre Fracción y la interfaz Comparar. Además, la clase Figura es la superclase de las subclases Rectángulo, Triángulo y Círculo.

5.5. Tipos de clases de análisis

Durante la fase de análisis del desarrollo de una aplicación, siguiendo el paradigma orientado a objetos, se deben identificar tres tipos de clases:

1. Las **clases de interfaz** se usan para modelar la interacción entre el sistema y sus actores. Esta interacción consiste normalmente en recibir y mostrar información o pedirla a las personas usuarias o sistemas externos. Estas clases se deben limitar a describir la información y peticiones que se intercambian los actores y el sistema a través de ellas. Cada clase de interfaz se debería asociar a, al menos, un actor, y viceversa. Se usan estas clases para modelar ventanas, formularios, impresos, etc.
2. Las **clases de entidad** se emplean para modelar información que persiste en el tiempo. Suelen mostrar una estructura de datos lógica.
3. Las **clases de control** realizan la coordinación y el control sobre otros objetos del sistema. Toda la dinámica del sistema es modelada por clases de control. Se asocia una clase de control a cada aplicación y esta clase modela la dinámica del sistema delegando trabajo a otras clases.

Cada tipo de clase se representa como se muestra en la Figura 5.30.

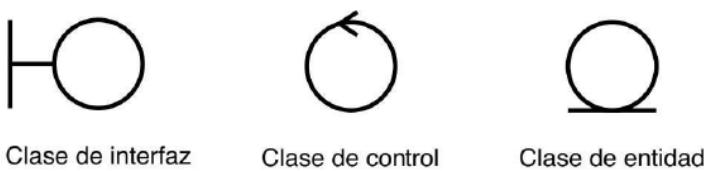


Figura 5.30. Representación de los tipos de clases de interfaz, de control y de entidad.

■ 5.6. Herramientas para la creación de diagramas de clases

En la actualidad, existen muchas herramientas para la creación de diagramas de clases. En esta unidad, se aprenderá a emplear tres de ellas:

- **diagrams.net:** es una herramienta muy sencilla de aprender y utilizar, pero que no realiza comprobaciones sobre los diagramas creados, por lo que es posible crear diagramas que no respeten las normas de UML.
- **Papyrus:** es una herramienta mucho más avanzada que se puede integrar en Eclipse como un módulo y que permite realizar todo tipo de diagramas UML.
- **Modelio:** es una aplicación de modelado UML de código abierto que permite crear todo tipo de diagramas y, además, generar el código correspondiente.

■ 5.6.1. Creación de diagramas de clases con diagrams.net

La herramienta *diagrams.net* permite elaborar multitud de diagramas distintos y se puede utilizar accediendo a la página web <https://app.diagrams.net/>, por lo que no es necesario instalarla en el ordenador, aunque también se ofrece la posibilidad de descargarse una aplicación de escritorio e instalarla.

Nada más acceder a la página web indicada, se ofrecen dos opciones: crear un nuevo diagrama o abrir un diagrama (véase la Figura 5.31).



Figura 5.31. Pantalla de inicio de diagrams.net, en la que se muestran dos opciones: crear un nuevo diagrama o abrir un diagrama existente.

Tras seleccionar la opción *Create New Diagram*, en la ventana que aparece a continuación, se asigna un nombre al diagrama y se deja el tipo de archivo por defecto: de tipo XML con la extensión .drawio. Se puede elegir, además, el tipo de diagrama que se desea crear; en este caso, UML. Por último, se hace clic en el botón *Create* (Figura 5.32).

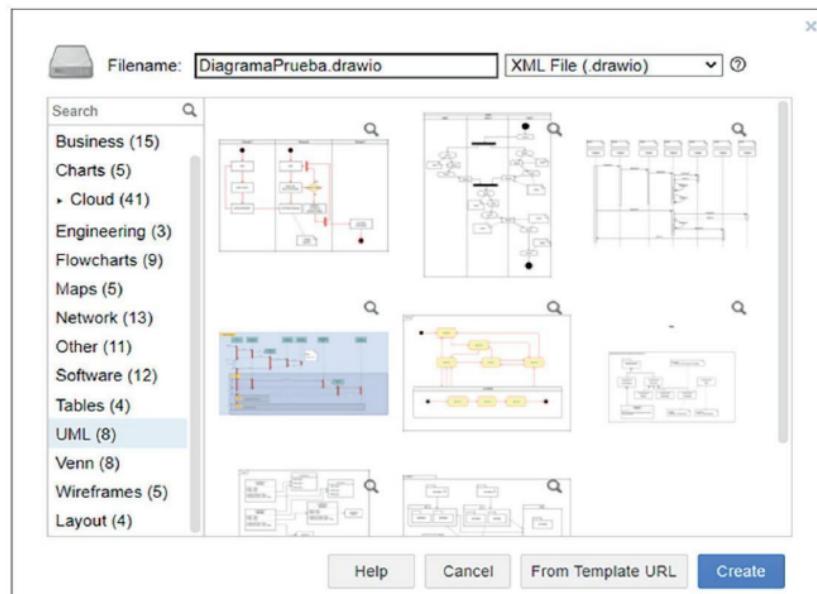


Figura 5.32. Ventana en la que se da un nombre al diagrama y se deja el tipo y la extensión por defecto.

A continuación, la herramienta pide indicar la ubicación en nuestro equipo donde se desea guardar el diagrama y se debe volver a escribir su nombre (Figura 5.33).

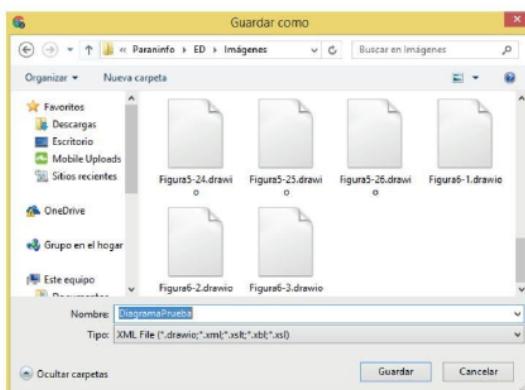


Figura 5.33. Ventana en la que se indica la ubicación donde se desea guardar el diagrama en el equipo y se asigna su nombre.

Aparece entonces la pantalla principal de diagrams.net (Figura 5.34), en la que se distinguen las siguientes áreas:

- **Área de paletas:** en esta área situada en la parte izquierda de la pantalla, hay distintas paletas disponibles para seleccionar los elementos que se desean dibujar en nuestro diagrama. Hay varias paletas con distintos nombres para crear distintos tipos de diagramas: *General*, *Flowchart*, *Entity Relation*, *UML*, *UML 2.5*, etc. En el ejemplo que aquí se expone, se hará uso de las paletas *UML* y *UML 2.5*. Al hacer clic en cualquiera de ellas, se pueden visualizar todas las formas de que dispone cada paleta. Si no está disponible la paleta *UML 2.5*, se debe pulsar en el enlace *+ More Shapes* de la esquina inferior izquierda de la pantalla y, en la sección *Software* de la ventana que aparecerá a continuación, se marca la casilla de verificación correspondiente a *UML 2.5*, tras lo cual aparecerá esta paleta en el área de paletas.
- **Área de edición:** es el área que aparece en la parte central de la pantalla y en ella se dibujará el diagrama correspondiente.
- **Área de propiedades:** en esta área de la parte derecha, se muestran y se pueden modificar las propiedades del elemento seleccionado en el área de edición o del diagrama, si no está seleccionado ningún elemento.

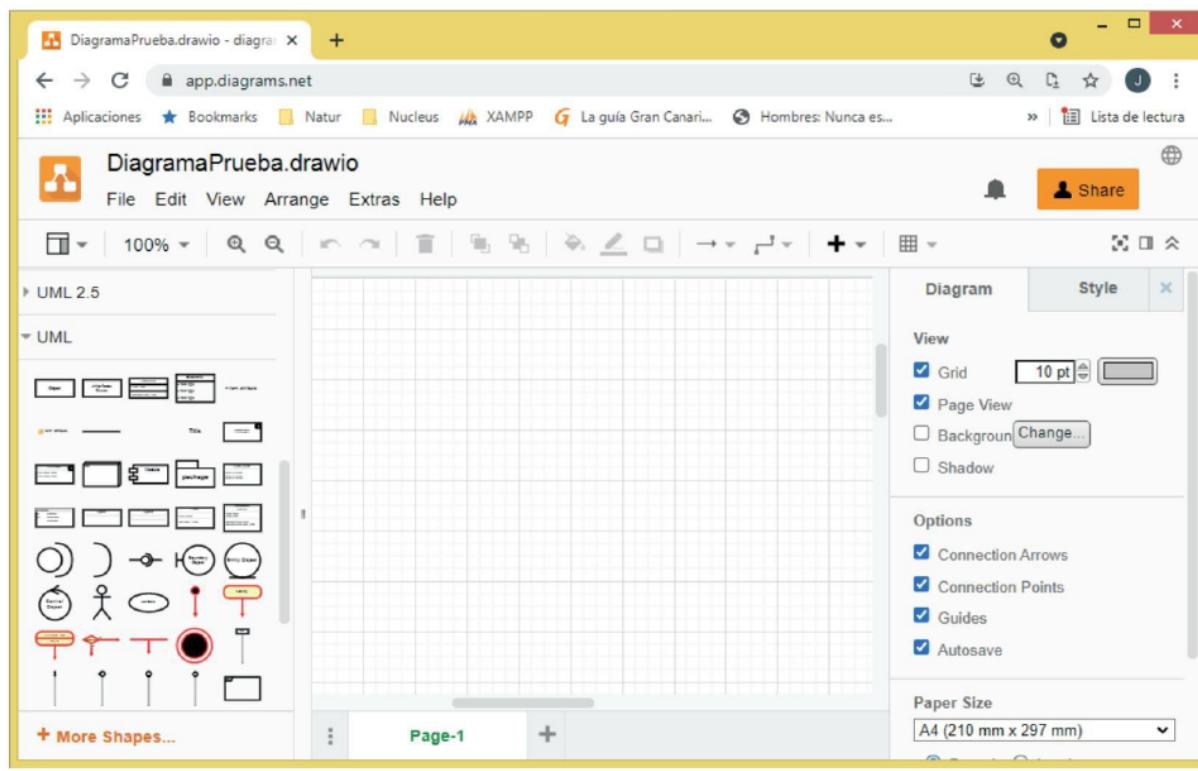


Figura 5.34. Pantalla principal de diagrams.net, donde se distinguen tres áreas de izquierda a derecha: área de paletas, área de edición y área de propiedades.

Para practicar el uso de esta herramienta, aquí se va a crear un diagrama de clases que refleja información sobre el personal y los clientes de una empresa, así como los artículos que esta vende y los pedidos que realizan los clientes de estos artículos (véase su representación en la Figura 5.35).

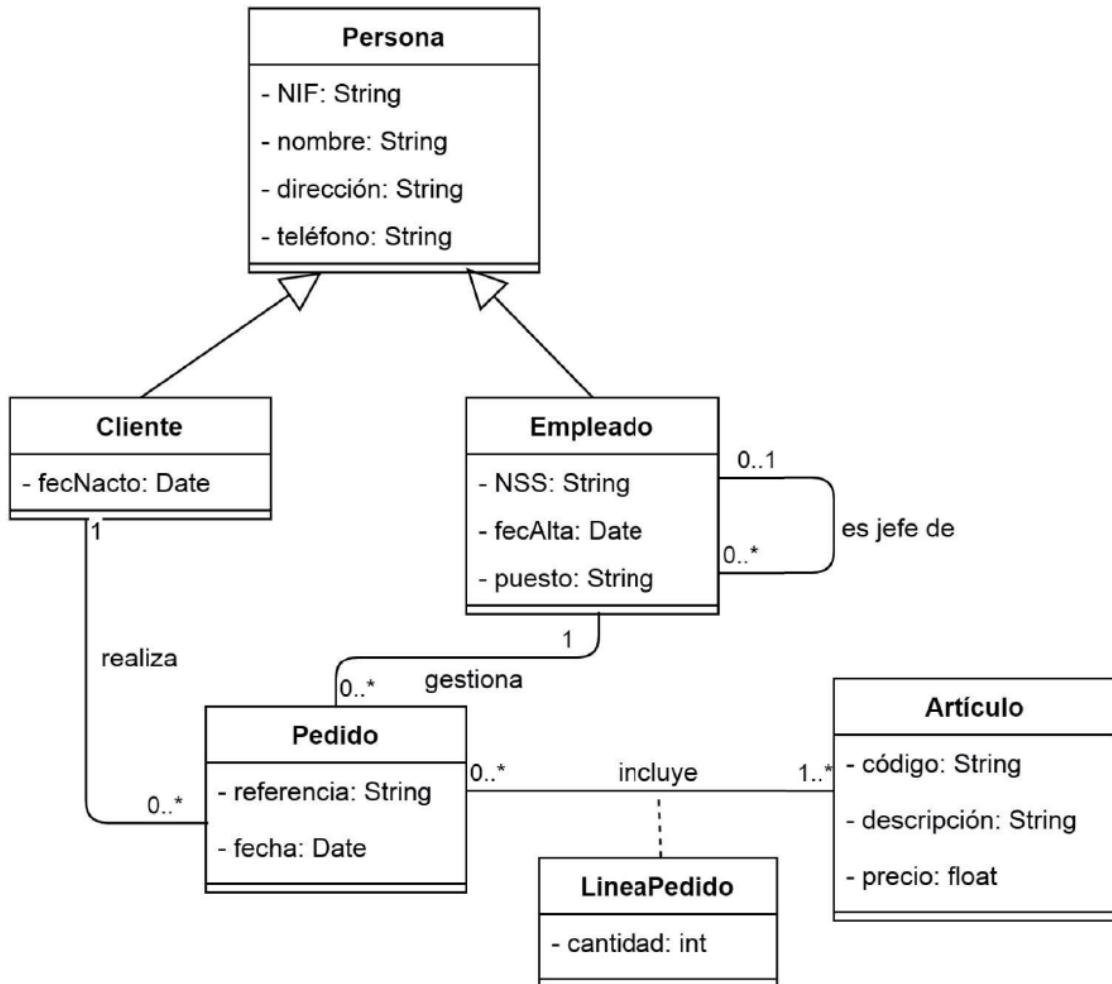


Figura 5.35. Diagrama de clases creado con diagrams.net que refleja el personal y los clientes de una empresa y los pedidos que estos hacen de los artículos que vende la empresa.

Para crear una clase, tan solo hay que hacer clic sobre uno de los dos siguientes íconos de la paleta UML: , en función de si se quiere reflejar atributos y métodos de cada clase o solamente sus atributos, respectivamente. Aquí, se hará clic con el botón izquierdo del ratón en el primero de estos iconos para crear las clases. Aparecerá entonces una clase en el área de edición. Después, hay que colocarse sobre la palabra *Classname* y escribir el nombre que se desea dar a la clase, por ejemplo, *Pedido*. Una vez hecho esto, hay que colocarse sobre cada campo e indicar su visibilidad, nombre y tipo, por ejemplo: *- referencia: String* (Figura 5.36). Para añadir otro atributo, se selecciona este con el botón izquierdo del ratón, se activa en el menú contextual la opción *Duplicate* y aparecerá duplicado debajo del último método, pero se puede subir, arrastrándolo con el botón izquierdo del ratón pulsado hasta el área de atributos que se quiera. Es posible asimismo cambiar su visibilidad, nombre y tipo. Si no se desea indicar métodos para alguna clase, es posible colocarse sobre todos los métodos que haya y eliminarlos, eligiendo la opción *Delete* del menú contextual o pulsando la tecla *Supr*. Se puede modificar el tamaño de una clase, seleccionándola y pulsando sobre la sección superior de la clase (donde se muestra el nombre) y arrastrando cualquiera de las flechas que se muestran.

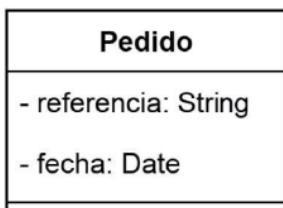


Figura 5.36. Clase *Pedido* creada con *diagrams.net* con dos atributos (*referencia* y *fecha*) y sin métodos.

La clase *Artículo* se podría crear de manera similar.

Para dibujar una asociación sin indicación de navegabilidad entre dos clases, se puede clicar en el elemento de la sección *UML* de la paleta que se muestra en la Figura 5.37.



Figura 5.37. Elemento de la paleta UML para crear una asociación entre dos clases.

Al hacer clic sobre este elemento, este pasará al área de edición y se debe arrastrar pulsando cada uno de sus extremos hasta el borde de la clase donde se quiere unir la línea. Una vez hecho esto, se sobrescriben las palabras *parent* y *child* con las cardinalidades o multiplicidades que corresponda. En este caso, para la relación entre *Pedido* y *Artículo*, una de las cardinalidades debe ser $0..*$ y la otra, $1..*$. Aunque en un diagrama de clases UML, no es obligatorio asignar nombre a las relaciones, es muy conveniente. Para ello, se puede hacer uso del elemento *Text* de la sección *General* de la paleta: se hace clic en este elemento y aparecerá en el área de edición. Se sobrescribe la palabra *Text* con el texto *incluye* y se arrastra este hasta colocarlo sobre la línea de la relación. De este modo, quedarán unidas las clases *Pedido* y *Artículo* (Figura 5.38).

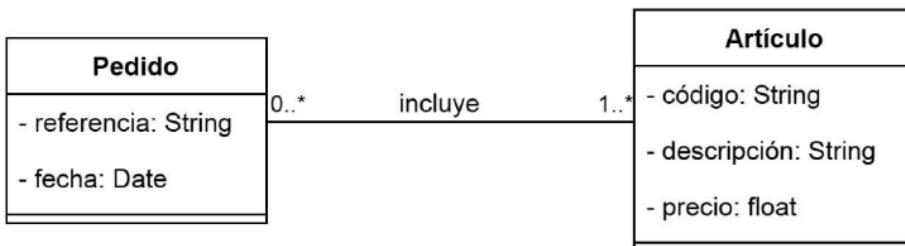


Figura 5.38. Parte del diagrama de clases en el que se muestran las clases *Pedido* y *Artículo* unidas por medio de una asociación llamada *incluye*.

Como se puede observar en la Figura 5.35, hay una clase asociativa vinculada a la relación *incluye* llamada *LíneaPedido*. En este ejemplo, se creará esta clase como cualquier otra, y para unirla a la relación *incluye*, se usará una línea con trazado discontinuo mediante el elemento *Dashed Line* de la sección *General* de la paleta (Figura 5.39).

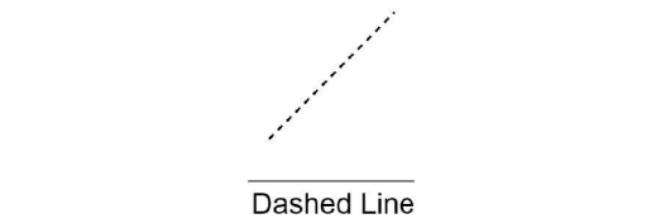


Figura 5.39. Elemento de la paleta General para crear una línea discontinua.

Después, se pueden crear las clases *Persona*, *Cliente* y *Empleado* y para reflejar la relación de generalización-especialización entre la superclase *Persona* y las subclases *Cliente* y *Empleado*, se utilizará el elemento llamado *Generalization* de la sección *UML* de la paleta (Figura 5.40).

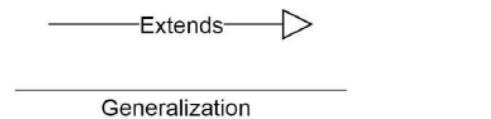


Figura 5.40. Elemento de la paleta UML para crear una relación de generalización-especialización.

Tras hacer clic en este elemento, se arrastra el extremo con el triángulo hasta que toque la superclase *Persona* y el otro extremo hasta tocar una subclase. Conviene eliminar la leyenda *Extends* de la flecha, para lo que hay que colocarse sobre esta leyenda y pulsar la tecla *Supr*.

Por último, se pueden establecer las relaciones binarias *realiza* y *gestiona* entre *Cliente* y *Pedido* y entre *Empleado* y *Pedido*, respectivamente, y la relación reflexiva es *jefe* de la clase *Empleado*.

Una vez terminado el diagrama, se debe guardar activando la opción de menú *File → Save*, o bien *File → Save as*, esta última opción si se quiere dar un nombre distinto del indicado con anterioridad. Se creará un archivo editable desde *diagrams.net* en la ubicación indicada de tipo XML y con extensión *.drawio*.

También es posible exportar el archivo a diferentes formatos: de tipo imagen (PNG, JPEG y SVG), PDF, HTML, etc. Para ello, hay que seleccionar la opción de menú *File → Export as*.

■■■ 5.6.2. Creación de diagramas de clases con Papyrus

Papyrus SysML fue instalado como un módulo del IDE Eclipse en el Apartado 2.5.1. Este módulo permite crear todos los tipos de diagramas UML.

Una vez instalado, para su uso, es necesario abrir la vista *Papyrus* para que aparezcan en pantalla todos los elementos necesarios para la creación de diagramas UML. Para que se abra dicha vista, se selecciona la opción de menú *Window → Perspective → Open Perspective → Other → <Papyrus>*.

Con objeto de generar un diagrama de clases, lo primero que se debe hacer es crear un proyecto, para lo que hay que seleccionar la opción de menú *File → New → Papyrus Project*. En la ventana que se muestra en la Figura 5.41, se marca la casilla de verificación correspondiente a UML:

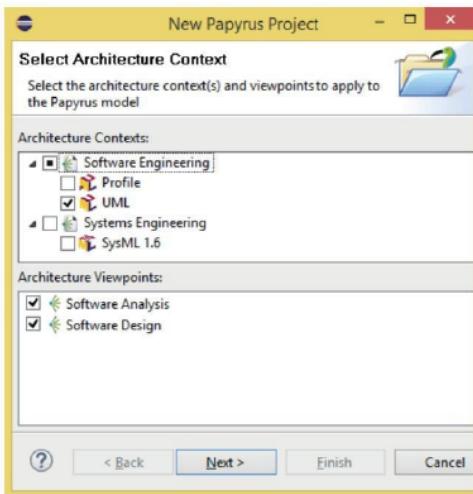


Figura 5.41. Al crear un proyecto en Papyrus para elaborar un diagrama de clases, se debe seleccionar el contexto UML.

Después de hacer clic en el botón *Next*, en la siguiente ventana, se debe asignar un nombre al proyecto, por ejemplo, *DClasesEmpresa*. En la ventana siguiente (Figura 5.42), es necesario indicar el tipo o los tipos de diagramas que se desea incluir en el proyecto. En este caso, solo se indica que se desea hacer diagramas de clases marcando la casilla de verificación *Class Diagram*.

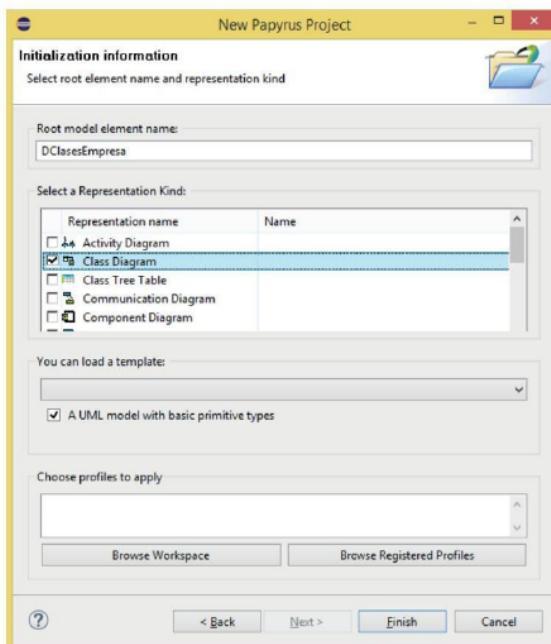


Figura 5.42. Como último paso en el proceso de creación de un proyecto Papyrus, se han de seleccionar el tipo o los tipos de diagramas UML que se desean crear en el proyecto.

Tras pulsar en el botón *Finish*, aparece una ventana (Figura 5.43), con varias áreas:

- **Explorador de proyectos (Project Explorer)**, en la parte superior izquierda.
- **Explorador del modelo (Model Explorer)**, en la parte izquierda debajo del explorador de proyectos; en esta área, se permite navegar por los diferentes elementos del modelo. Se creó un modelo asociado al proyecto en el proceso de creación del proyecto Papyrus.
- **Outline**, en la parte inferior izquierda; en esta área, se muestra una miniatura del diagrama que se está creando.
- **Editor de diagramas**, en la parte central; esta área se emplea para el dibujo del diagrama correspondiente.
- **Paleta**, en la parte derecha; se debe seleccionar de la paleta los elementos que se desean añadir al diagrama haciendo clic, en primer lugar, sobre el elemento correspondiente y luego sobre el editor de diagramas. En la sección *Nodes* de la paleta, se pueden seleccionar clases, atributos, interfaces, métodos, etc.; y en la sección *Edges*, las asociaciones entre clases.
- **Propiedades (Properties)**, en la parte inferior; en esta área se pueden ver y modificar las propiedades o características del elemento del modelo seleccionado, el cual se puede seleccionar en el diagrama (editor de diagramas) o en el explorador del modelo.

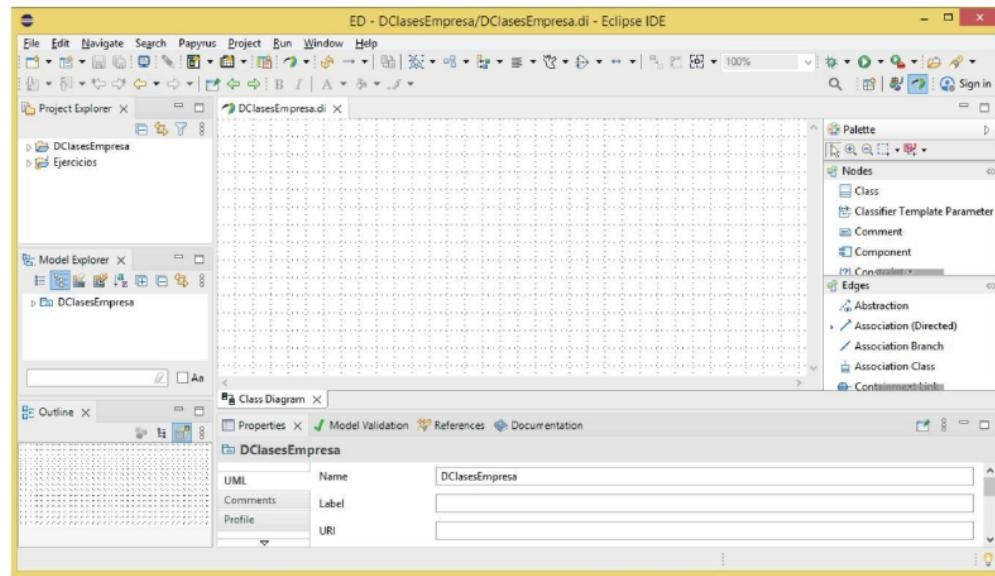


Figura 5.43. Vista Papyrus de Eclipse con los elementos necesarios para crear diagramas de clases repartidos en diferentes áreas.

Con el propósito de practicar el uso de esta herramienta, aquí se va a crear el mismo diagrama de clases que se creó con diagrams.net en el Apartado 5.6.1 (Figura 5.35).

Para crear una clase hay que hacer clic sobre el elemento con el nombre *Class* de la paleta y luego clicar en el editor de diagramas. Seleccionada la clase, en el área de propiedades, se escribe el nombre que se desea dar a la clase en el campo *Name*, por ejemplo, *Pedido*.

Si se escribe algo en el campo *Label*, lo que se escriba en él es lo que se mostrará en el diagrama como nombre de la clase. Se puede indicar si la clase es abstracta o no en *Is abstract*, y si es o no activa en *Is active*. Es posible establecer su visibilidad seleccionando entre las opciones *public*, *private*, *protected* o *package*. En este caso, se van a dejar todas las opciones por defecto (Figura 5.44).

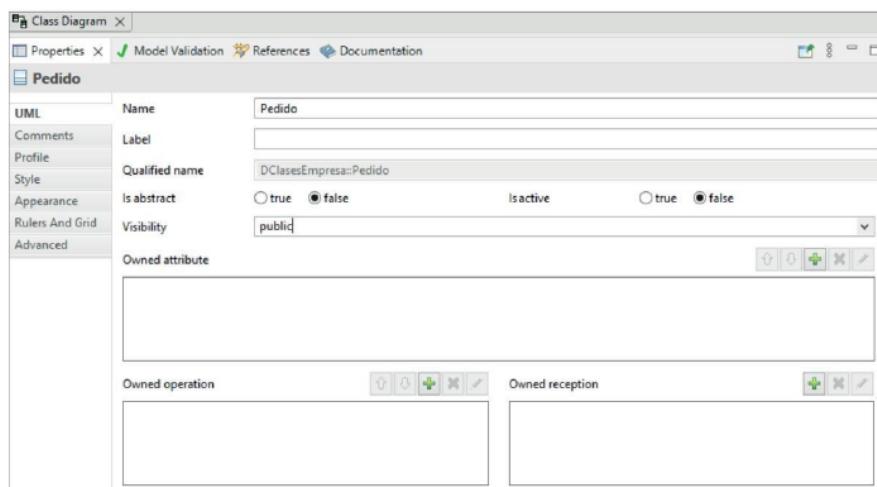


Figura 5.44. En la sección Properties, se pueden ver y modificar las propiedades de una clase (su nombre, visibilidad, etc.).

Para indicar los atributos de la clase, se debe hacer clic en el botón de la sección *Owned attribute* del área de propiedades. En la ventana que aparece entonces (Figura 5.45), se debe indicar, por cada atributo, su nombre en *Name*, su visibilidad, su tipo de dato en *Type* y su valor por defecto, si lo tiene, en *Default value*. Además, si el atributo tiene un valor único, se elige la opción *true* en *Is unique*. En cuanto al tipo de dato, la mayoría de los tipos de datos primitivos de Java están disponibles en el paquete «EPackage, ModelLibrary» Primitive Types. Se dispone de más tipos de datos, como el de tipo fecha (*EDate*) en el paquete «ModelLibrary» EcorePrimitive Types, y se accede a estos al hacer clic en el botón que aparece al lado de *Type*.

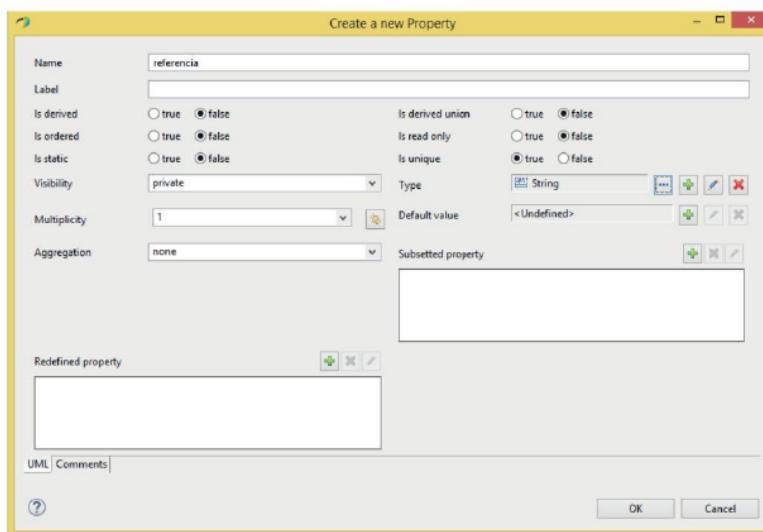


Figura 5.45. Ventana que muestra las propiedades del atributo referencia de la clase Pedido: su nombre, tipo de dato y visibilidad son las más importantes.

Se debe proporcionar la misma información para el otro atributo de la clase *Pedido*, es decir, para *fecha*.

Una vez proporcionada la información de los atributos, para que esta se pueda visualizar en el diagrama, es necesario arrastrar cada atributo del explorador del modelo hasta la clase a la que pertenece. De este modo, la clase *Pedido* se verá en el diagrama como se muestra en la Figura 5.46.

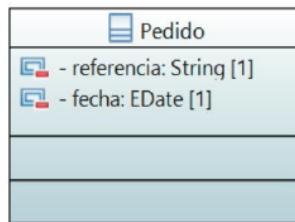


Figura 5.46. Representación de la clase Pedido en un diagrama de clases UML creado con Papyrus SysML.

Se puede crear de manera similar la clase *Artículo*.

Para dibujar una asociación con clase asociativa vinculada, se debe hacer clic en el ícono de la sección *Edges* de la paleta con el nombre *Association Class* y luego se debe pinchar sobre una de las clases y posteriormente sobre la otra. Aparecerá entonces una asociación dirigida entre las dos clases y la clase asociativa, a la que se debe dar el nombre *LíneaPedido* y asignarle el atributo *cantidad* de tipo *integer*.

Al hacer clic sobre la línea que representa la asociación, se mostrarán las propiedades de la asociación, las cuales se pueden modificar. En *Name* aparece el nombre de la clase asociativa, *LíneaPedido*, que se va a mantener. Se deben cambiar las multiplicidades o cardinalidades a cada lado de la asociación, de modo que sean *1..** al lado de *Artículo* y *0..** al lado de *Pedido*. Para que estas se muestren en el diagrama, además de ponerlas en *Multiplicity*, se deben escribir en el campo *Name*, como se muestra en la Figura 5.47.

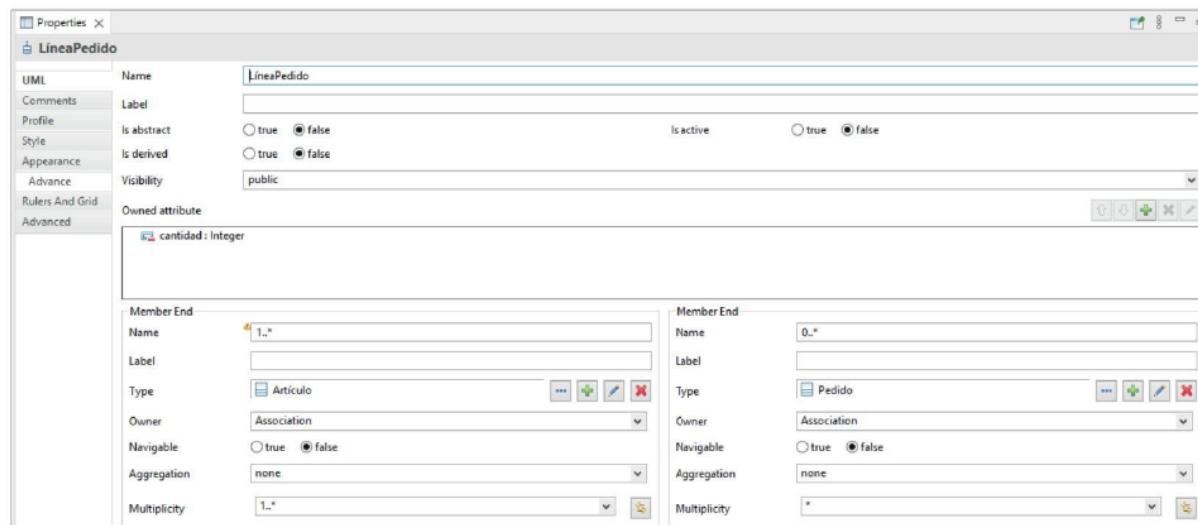


Figura 5.47. Propiedades de la asociación con clase asociativa LíneaPedido entre Pedido y Artículo.

Tras estos cambios, la asociación entre *Pedido* y *Artículo* con la clase asociativa *LíneaPedido* quedará como se refleja en la Figura 5.48.

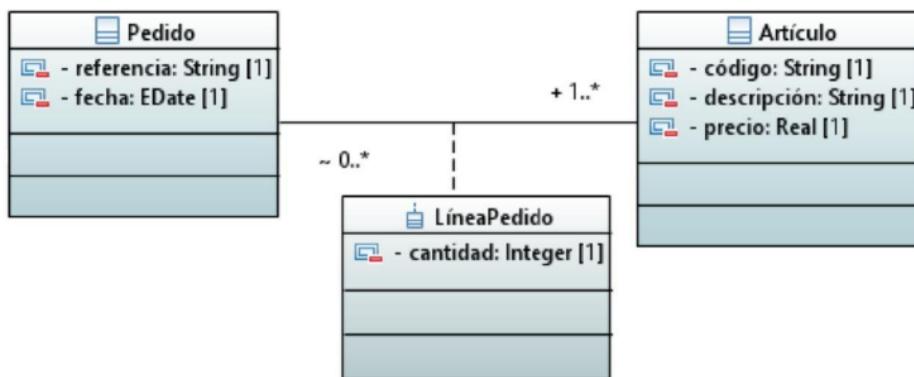


Figura 5.48. Representación de las clases *Pedido* y *Artículo* unidas por medio de una asociación con una clase asociativa vinculada llamada *LíneaPedido*.

A continuación, se crearán las clases *Persona*, *Cliente* y *Empleado*. Una vez creadas estas tres, se debe establecer la relación de generalización existente entre ellas, ya que *Persona* es superclase de *Cliente* y *Empleado*. Para ello, se hace clic sobre *Generalization* en la sección *Edges* de la paleta y luego se clica en la subclase y la superclase.

Se puede ahora establecer la relación binaria *realiza* entre *Cliente* y *Pedido*. A tal fin, se debe hacer clic sobre *Association (Directed)* en la sección *Edges* de la paleta y luego sobre cada una de las clases vinculadas (*Cliente* y *Pedido*). Aparecerá entonces una asociación con flecha (dirigida) entre las dos clases. Al hacer clic sobre la línea que representa la asociación, se puede modificar sus propiedades. En *Name* se escribe el nombre *realiza*. Si no se quiere que la asociación sea navegable o tenga punta de flecha, se seleccionará la opción *false* en *Navigable*. Finalmente, se escribe en *Multiplicity* las cardinalidades correspondientes: *0..** al lado de *Pedido* y *1* al lado de *Cliente*. Las propiedades de esta asociación deben establecerse como se indica en la Figura 5.49.

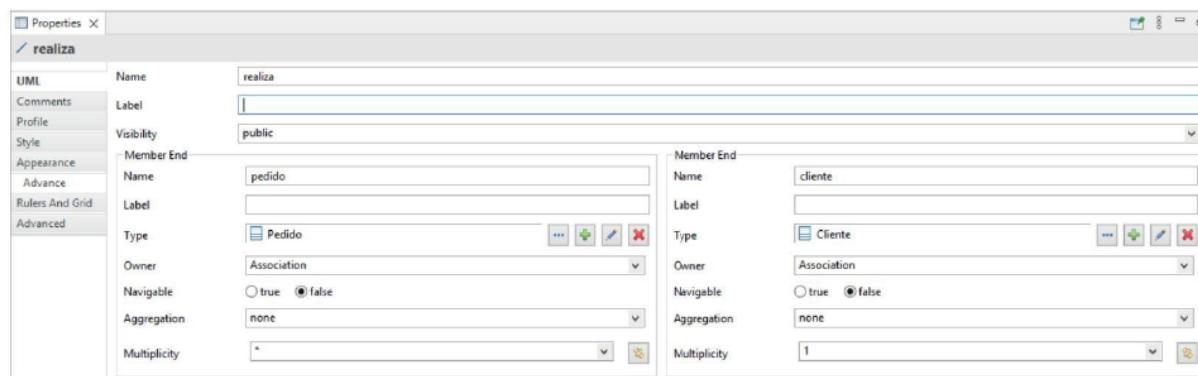


Figura 5.49. Propiedades de la asociación *realiza* entre *Cliente* y *Pedido*.

De manera similar, se crean la asociación binaria *gestiona* entre *Empleado* y *Pedido* y la relación reflexiva *es jefe de* de la clase *Empleado*. El diagrama de clases quedará como se muestra en la Figura 5.50.

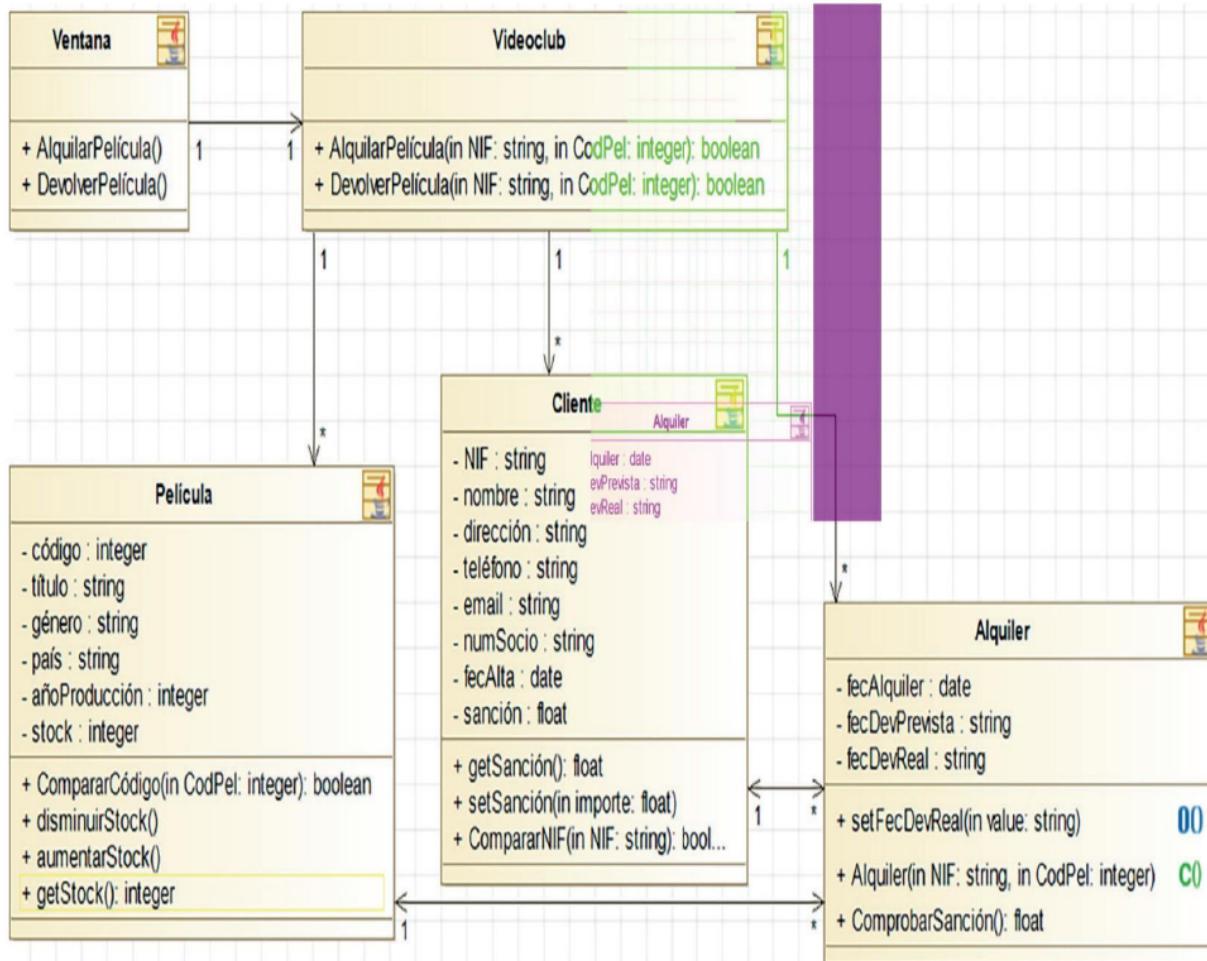


Figura 5.50. Diagrama de clases creado con Papyrus en Eclipse que refleja los empleados y clientes de una empresa y los pedidos que estos hacen de artículos que vende la empresa.

Una vez finalizado el diagrama, se debe guardar seleccionando la opción de menú *File → Save*. También es posible exportar el diagrama a diferentes formatos, seleccionando el modelo en el explorador de proyectos y la opción del menú contextual *Export...* y luego la opción *Papyrus → Export All Diagrams*. Se puede exportar a formato GIF, BMP, JPEG, JPG, SVG o PDF.

5.6.3. Creación de diagramas de clases con Modelio

La herramienta Modelio permite crear todo tipo de diagramas UML y se puede descargar, desde la página web <https://www.modelio.org/downloads/download-modelio.html>, la versión correspondiente al del sistema operativo sobre el que se desee realizar la instalación (*Windows, RedHat, Debian, Ubuntu o macOS*).

Al ejecutar por primera vez la herramienta, aparece una ventana de bienvenida desde la que es posible acceder a diferentes guías. Aquí se va a realizar el diagrama de clases de la Figura 6.18 de la Unidad 6.

Una vez se cierra la ventana de bienvenida, para comenzar a trabajar con Modelio, se debe crear un proyecto seleccionando la opción de menú *File → Create a project*. En el cuadro de diálogo que aparece entonces (Figura 5.51), se debe dar un nombre al proyecto y se puede proporcionar una descripción de este. Después, se activa la casilla de verificación *Java project* para posteriormente generar el código Java correspondiente al diagrama de clases. Para crear el proyecto, se pulsa sobre el botón *Create the project*.

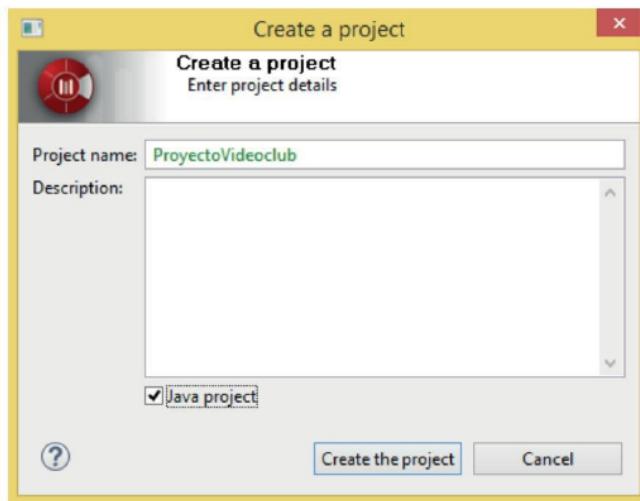


Figura 5.51. Ventana de creación de un proyecto en Modelio, en la que se debe dar un nombre al proyecto que se está creando.

En la parte izquierda de la ventana, aparece entonces una carpeta con el nombre del proyecto. En el proyecto, se creará un paquete con el nombre *Videoclub*. Para ello, se despliegan las dos carpetas con el nombre del proyecto y en el menú contextual de la carpeta inferior, se elige la opción de menú *Create element → Package*, como se puede observar en la Figura 5.52.

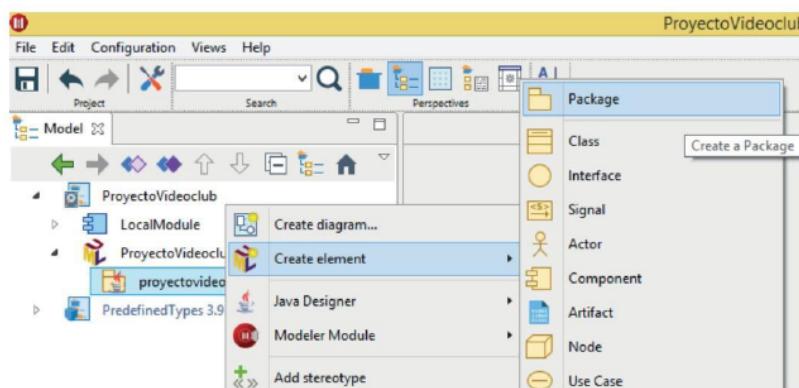


Figura 5.52. Para crear un diagrama en Modelio, una vez creado un proyecto, hay que seleccionar la carpeta correspondiente al proyecto y la opción del menú contextual *Create element → Package*.

Seguidamente aparece en el explorador del modelo un paquete al que se debe asignar un nombre escribiendo directamente el nombre elegido sobre el nombre que aparece por defecto (*Package*); en este caso, se va a llamar *Videoclub*.

A continuación, dentro de este paquete, se indica que se desea crear un diagrama, activando la opción del menú contextual *Create diagram* (véase la Figura 5.53).



Figura 5.53. Para crear un diagrama en Modelio, tras crear un paquete, hay que seleccionar el paquete y elegir la opción del menú contextual Create diagram.

Es necesario indicar el tipo de diagrama que se quiere crear, en este caso, un diagrama de clases, y asignarle un nombre (Figura 5.54).

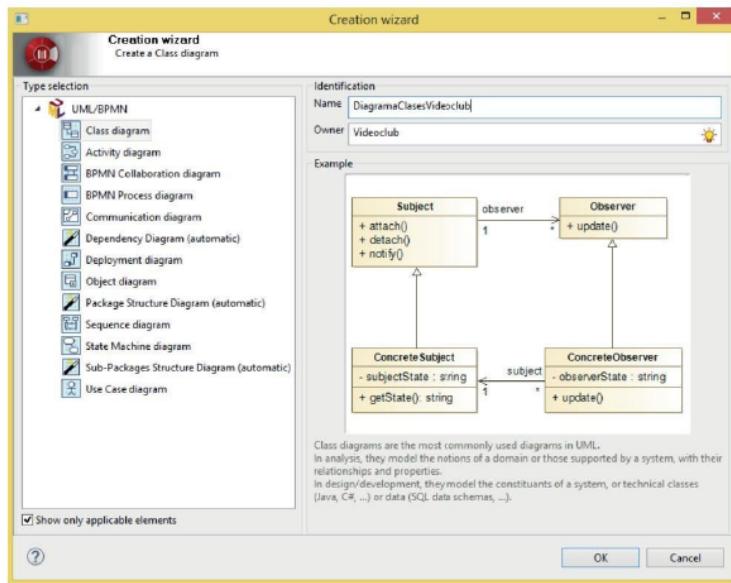


Figura 5.54. Ventana que se muestra antes de proceder a la creación de un diagrama en la que se selecciona el tipo de diagrama que se desea crear y se le da un nombre.

Tras clicar en el botón *OK*, aparecerá la pantalla principal de Modelio, donde se pueden distinguir varias áreas:

- **Explorador del modelo**, en la parte izquierda, donde se muestran todos los elementos del diagrama por los que se puede navegar haciendo doble clic sobre cada uno de ellos.
- **Área de paletas**, a continuación a la derecha, donde se muestran varias paletas para seleccionar los elementos que se desean incorporar a los diagramas.

- **Área de edición**, en la parte superior derecha de la pantalla, que es la zona reservada para la creación de los diagramas.
- **Área de detalles**, en la parte inferior; en esta área, se muestran diversos tipos de información en función de la pestaña seleccionada. Así, si se hace clic sobre la pestaña *Properties*, se muestran las propiedades del elemento seleccionado del diagrama; si se hace clic sobre *Diagrams*, se muestra un desplegable con los diagramas del proyecto.

Lo primero que se debe hacer es indicar que el paquete que se ha creado con el nombre *Videoclub* es un elemento Java para posibilitar posteriormente la generación de código Java a partir del diagrama de clases. A tal fin, seleccionado el paquete, en el área de detalles en la pestaña *Properties*, se marca la casilla de verificación *Java element*.

Para crear una clase en el diagrama, tan solo hay que pinchar sobre el icono  de la paleta *Class model* y hacer clic en la posición deseada del área de edición. O bien en la pestaña *Properties*, o bien seleccionando la opción del menú contextual de la clase *Edit element*, se pueden ver y modificar las propiedades de la clase. En la pestaña *Properties*, se le asigna a la clase el nombre *Película*. Para añadirle atributos a la clase, se pulsa sobre el icono  de la paleta *Class Model* y, luego, se clica sobre la clase. En la pestaña *Properties* del área de detalles, se puede asignar las propiedades necesarias al atributo: nombre, tipo de dato y visibilidad. En la Figura 5.55, se muestran las propiedades para el atributo *código* de la clase *Película*.

Property	
Name	código
Type	 integer
Visibility	Private
Multiplicity min	1
Multiplicity max	1
Value	
Access mode	Read/Write
Type constraint	
Abstract	<input type="checkbox"/>
Class	<input type="checkbox"/>
Derived	<input type="checkbox"/>
Ordered	<input type="checkbox"/>
Unique	<input type="checkbox"/>
Target is class	<input type="checkbox"/>

Figura 5.55. Ventana en la que se visualizan y editan las propiedades de un atributo en la pestaña *Properties*. En este caso, se muestran las propiedades del atributo *código* de la clase *Película*.

De igual modo, se añaden el resto de atributos a la clase *Película* (*título*, *añoProducción*, *género*, *país* y *stock*).

Para asignar métodos a una clase, se debe pulsar sobre el icono  de la paleta *Class Model* y luego se hace clic sobre la clase. Se puede observar que al llevar a cabo esta operación han aparecido automáticamente para la clase los métodos de selección (*get*) y acceso (*set*) para todos sus atributos. Esto es así porque el módulo *Java Designer* que

viene incorporado en Modelio tiene una propiedad por defecto que indica que se generen estos métodos de manera automática para todas las clases. Si se quiere eliminar esta posibilidad porque hace excesivamente densos los diagramas, se puede seleccionar la opción de menú *Configuration → Modules*, hacer clic sobre el módulo Java Designer, y en sus propiedades, que aparecen en la parte inferior de la ventana, asignar el valor *Never* a la propiedad *Accessor generation* dentro del apartado *Automation*, como se muestra en la Figura 5.56.

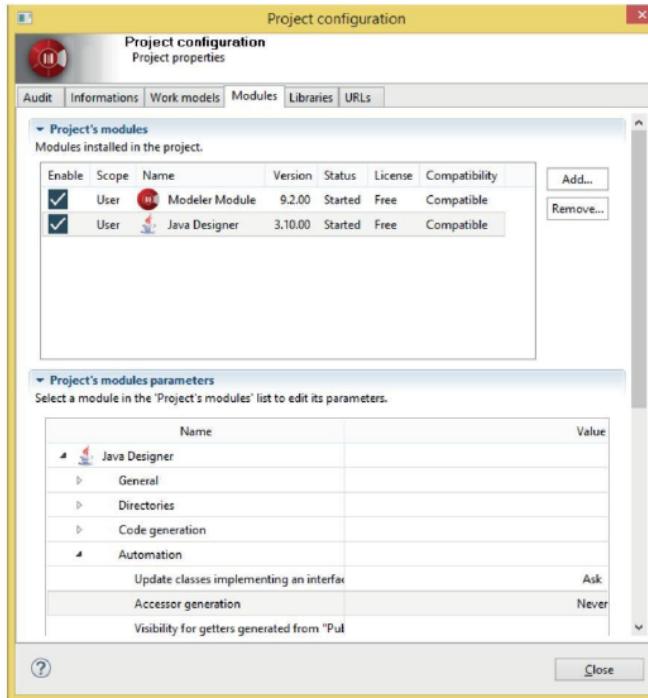


Figura 5.56 Se puede indicar a Modelio que no cree automáticamente métodos de consulta y acceso para todos los atributos de una clase modificando la propiedad *Accessor generation* del módulo Java Designer al valor *Never*.

No obstante, también es posible eliminar cualquier atributo o método asignado a una clase colocándose sobre él con el ratón y pulsando la tecla *Supr* o la opción del menú contextual *Delete selection*.

A la hora de asignar propiedades a un método u operación de una clase, por ejemplo, al método *CompararCódigo* de la clase *Película*, se debe hacer clic sobre el nombre asignado por defecto al método o elegir la opción del menú contextual *Edit element*. En la pestaña *Operation*, se le asigna un nombre al método (*CompararCódigo*) y se indica su tipo (operación, constructor o destructor) y su visibilidad (*public*). Además, en la sección *Operation parameters*, se indican los parámetros del método y su valor de retorno, si es el caso. El botón sirve para añadir un parámetro al método, y el botón , para indicar el valor de retorno del método. Con los iconos que aparecen a la derecha de la pantalla de la Figura 5.57, se ofrece la posibilidad de eliminar un parámetro, ascenderlo en la lista de parámetros o descenderlo, respectivamente. Se muestra a continuación toda la información para el método *CompararCódigo* de la clase *Película*. Este método presenta un parámetro de entrada *CodPel* de tipo *integer* y un valor de retorno de tipo *boolean*. Se puede observar el prototipo o la cabecera del método debajo de la lista de parámetros.

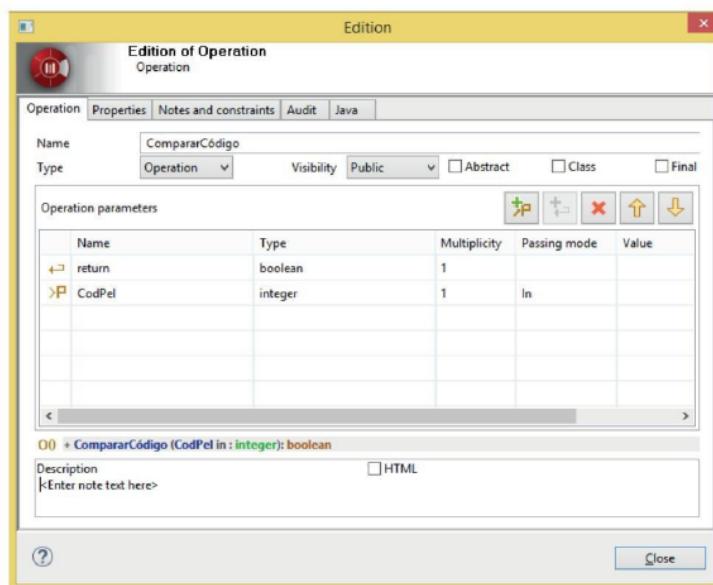


Figura 5.57. Ventana en la que se visualizan y modifican las propiedades de los métodos de una clase al seleccionar el método correspondiente y la opción del menú contextual *Edit element*. En la pestaña *Operation*, se debe asignar al método su nombre, tipo y visibilidad, y se deben definir sus parámetros y valor de retorno, si procede.

Después, se crean del mismo modo los restantes métodos de la clase *Película* (*dismuirStock*, *aumentarStock* y *getStock*) y, a continuación, la clase *Videoclub* con sus dos métodos *AlquilarPelícula* y *DevolverPelícula*.

Para crear relaciones entre clases, se debe clicar sobre el tipo de relación que nos interesa en la paleta *Class Model* y luego en la clase origen y en la clase destino. En el caso de la relación entre *Videoclub* y *Película*, se hará clic sobre el icono que se usa para una asociación o relación genérica. A continuación, se deben modificar las propiedades de la asociación (nombre, navegabilidad y multiplicidades mínimas y máximas al lado de cada clase), seleccionando la opción del menú contextual *Edit element*. En el caso de esta relación, sus propiedades deberán quedar como se muestra en la Figura 5.58.

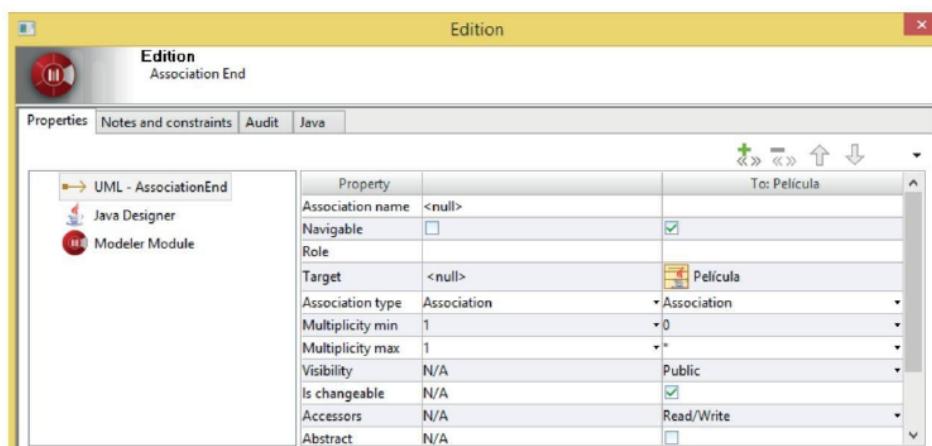


Figura 5.58. Ventana en la que se visualizan y modifican las propiedades de las relaciones establecidas entre clases, seleccionando la relación correspondiente y la opción del menú contextual *Edit element*. En la pestaña *Properties* se puede asignar un nombre a la relación, los roles, cambiar el tipo de asociación e indicar las multiplicidades mínimas y máximas al lado de cada clase.

Después, se crea el resto del diagrama de clases siguiendo los pasos que se han indicado. Al final, se obtiene un diagrama de clases como el que se muestra en la Figura 5.59.

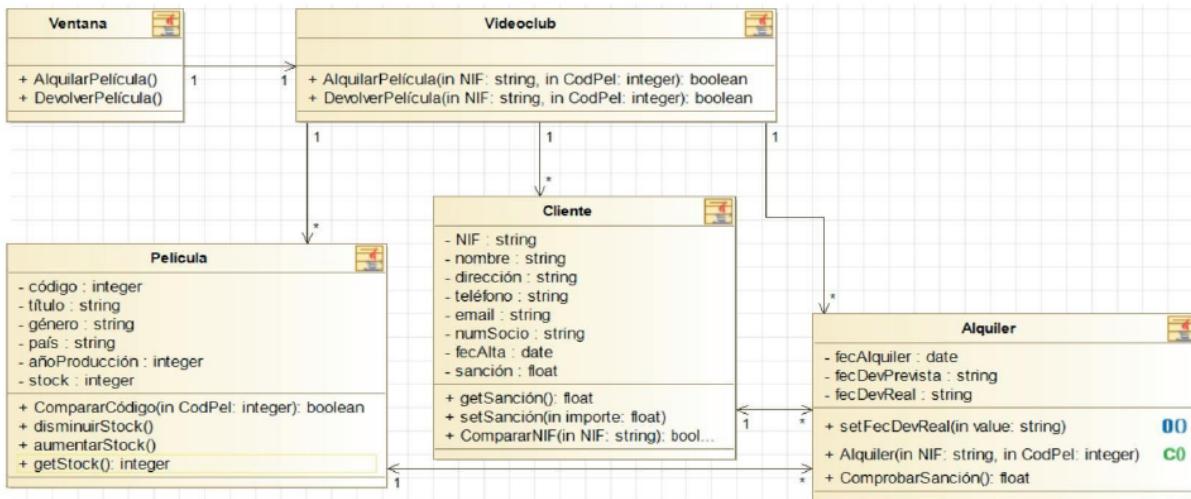


Figura 5.59. Diagrama de clases creado en Modelio que refleja la información que es necesario gestionar en un videoclub.

5.7. Generación de código a partir de diagramas de clases

Una vez generado un diagrama de clases, se puede crear de manera automática el código en Java mediante el empleo de un generador de código.

Para poner esto en práctica, aquí se usará Modelio, partiendo del diagrama de clases de la Figura 5.59. Esto se puede llevar a cabo gracias a que, al instalar Modelio, esta herramienta incorpora un módulo llamado *Java Designer*, una de cuyas utilidades es la generación de código Java. Para ello, es necesario haber especificado que las clases del diagrama, o bien el paquete que incluye todo el diagrama de clases, es un elemento Java. A tal fin, se selecciona el paquete o la clase correspondiente, y en la pestaña Java del área de detalles, se activa la casilla de verificación *Java element*.

Para generar el código Java correspondiente a una clase, se debe seleccionar dicha clase en el explorador del modelo y elegir la opción del menú contextual *Java Designer* → *Generate*, como se muestra para la clase *Película* en la Figura 5.60.

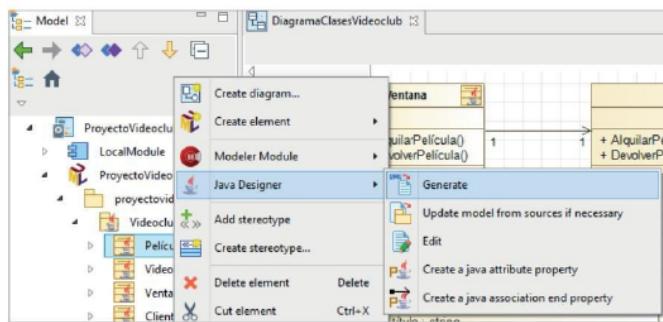


Figura 5.60. En Modelio es posible generar el código Java correspondiente a una clase activado la opción del menú contextual *Java Designer* → *Generate*.

Se habrá creado, en ese instante, un archivo con el código fuente Java correspondiente, llamado *Pelicula.java*. Este archivo se podrá encontrar en la carpeta *src/Videoclub* de la carpeta dedicada al proyecto, creada por Modelio en el equipo. Se puede visualizar el contenido de este archivo seleccionando para la clase la opción del menú contextual *Java Designer → Edit*. En la Figura 5.61, se puede observar el código generado para la clase *Película*, que debe ser revisado y modificado convenientemente. Como se puede ver en el código, Modelio ha añadido alguna importación y anotaciones propias. También es reseñable que ha creado un atributo de tipo lista para reflejar la navegabilidad de la relación entre *Película* y *Alquiler*, es decir, para registrar todos los alquileres que se han realizado para cada película.



```

DiagramaClasesVideoclub Película

package Videoclub;

import java.util.ArrayList;
import java.util.List;
import com.modeliosoft.modelojava.designer.annotations.objid;

@objid ("0ae7ce52-fd0f-4663-9f0f-f8652b91f153")
public class Película {
    @objid ("5eac3fc-b310-4790-b543-52759fc5108b")
    private int código;

    @objid ("29e1c7a0-899f-44b6-93b1-bb79c4c6b198")
    private String título;

    @objid ("a0864089-17c6-435f-b950-29b991a027ef")
    private String género;

    @objid ("8a25c2c8-1c3e-4417-9980-8f0555191f8d")
    private String país;

    @objid ("cc58639b-cb6d-4e5e-81c4-5d8d9b8b636d")
    private int añoProducción;

    @objid ("d62bce6a-1ef1-461b-97c0-2f253ed51fc0")
    private int stock;

    @objid ("8874f9ba-2127-43d3-929d-b2dbe028137c")
    public List<Alquiler> = new ArrayList<Alquiler>();

    @objid ("333be3d4-3e77-48d0-b7a9-67c2b0df7f47")
    public boolean CompararCódigo(int CodPel) {
    }

    @objid ("c51f22d1-b788-4298-b882-8e81cc9e7ca4")
}

```

Figura 5.61. En Modelio se puede visualizar el código Java correspondiente a una clase seleccionando la opción del menú contextual *Java Designer → Edit*.

Asimismo, existe la posibilidad de generar código para el paquete completo, seleccionándolo desde el explorador del modelo y eligiendo la opción del menú contextual *Java Designer → Generate*. Se generará un archivo con el código fuente correspondiente para cada clase del paquete en la carpeta *src/Videoclub* de la carpeta dedicada al proyecto, creada por Modelio.

■ 5.8. Generación de diagramas de clases a partir de código (ingeniería inversa)

Es posible definir la **ingeniería inversa** como «el proceso llevado a cabo con el objetivo de obtener información o un diseño a partir de un producto, con el fin de determinar cuáles son sus componentes y de qué manera interactúan entre sí y cuál fue el proceso de

fabricación» (https://es.wikipedia.org/wiki/Ingeniería_inversa). Como se puede deducir, este proceso no se aplica, por tanto, únicamente en el ámbito del software.

Argot técnico



Aplicada al software, la **ingeniería inversa** consiste en llevar a cabo las tareas de la **ingeniería del software** en un sentido contrario al habitual. Teniendo en cuenta que las fases del desarrollo más importantes de una aplicación informática son, en este orden, análisis, diseño y programación, se trata de partir del resultado de la tarea de programación, o sea, del código fuente, y obtener a partir de este la documentación que se debería haber obtenido durante las fases de diseño y análisis.

Aquí, se aplicará la ingeniería inversa para obtener a partir de una aplicación escrita en Java el diagrama de clases correspondiente. A tal efecto, se utilizará Modelio.

En primer lugar, se crea un proyecto en Modelio activando la casilla de verificación *Java project*. En el explorador del modelo, se selecciona la carpeta creada para el proyecto y se elige la opción del menú contextual *Java Designer* → *Reverse* → *Reverse Java application from sources* (Figura 5.62). A continuación, se seleccionan los archivos con el código fuente de las clases del proyecto para el que se desea aplicar ingeniería inversa. Para ello, se emplearán los archivos que vienen dentro de la carpeta Geometría que se encuentra disponible como material previo registro en www.paraninfo.es.

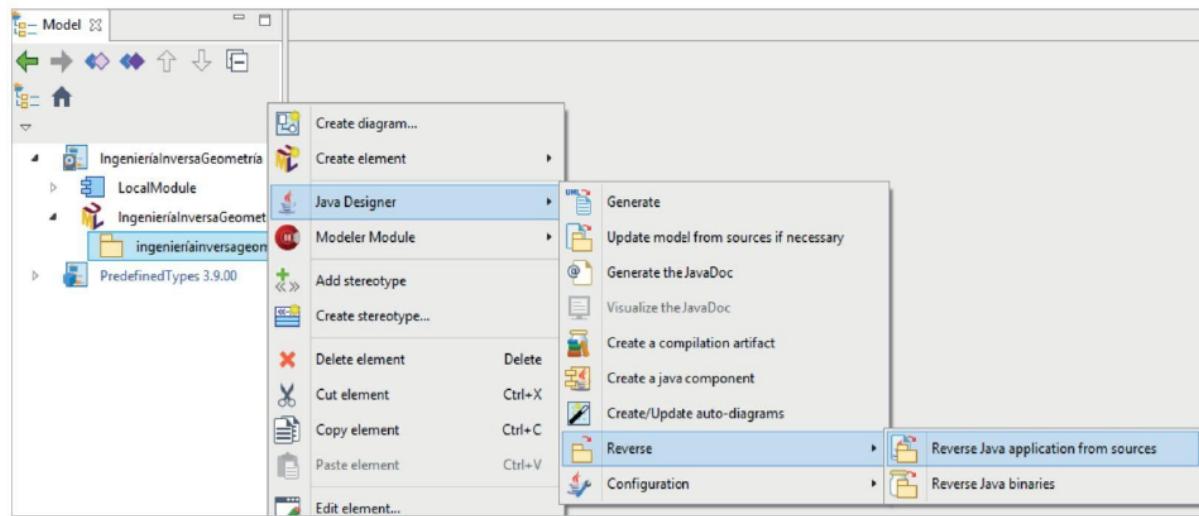


Figura 5.62. Para realizar ingeniería inversa a partir de archivos con código fuente de Java en Modelio, se debe crear un proyecto Java y activar la opción del menú contextual Java Designer → Reverse → Reverser Java application from sources.

En primer lugar, se pulsa en el botón para acceder a la ubicación donde se encuentran los archivos con el código fuente. Una vez seleccionada la carpeta, aparece una ventana como la de la Figura 5.63, en la que se deben marcar las clases deseadas.

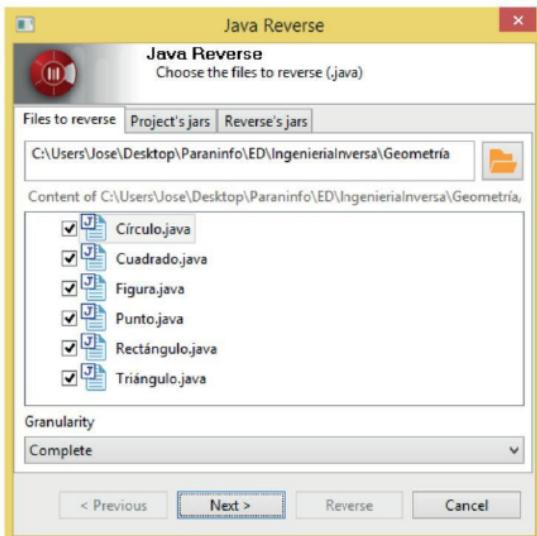


Figura 5.63. Ventana en la que se seleccionan las clases para las que se desea aplicar la ingeniería inversa.

En esta ventana, hay que pulsar la tecla *Next* y, también, en la siguiente, y finalmente la tecla *Reverse*. Entonces, aparece un aviso indicando que el tipo *String* no es reconocido, pero la ingeniería inversa se lleva a cabo y, de hecho, se muestran inmediatamente en el explorador del modelo todas las clases incluidas en el paquete. A continuación, es posible desplegar cada clase y visualizar todos sus elementos (atributos, operaciones y relaciones), como se puede observar en la Figura 5.64 para la clase *Círculo*.

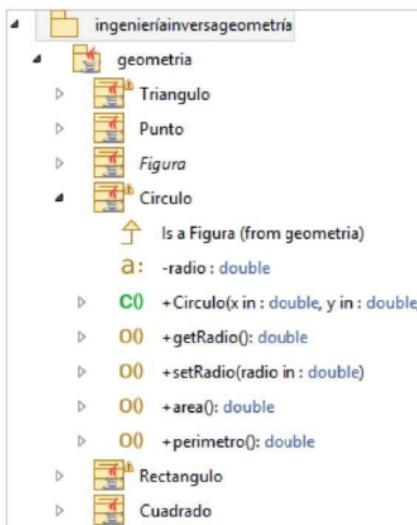


Figura 5.64. En el explorador del modelo se muestran todas las clases que se han creado a partir del código fuente. Estas se pueden desplegar para visualizar sus elementos (atributos, operaciones y relaciones).

Luego, es necesario crear un diagrama de clases, para lo que se activa la opción del menú contextual de la carpeta *Create diagram*, y se le asigna un nombre. El siguiente paso será arrastrar, desde el explorador del modelo hasta el área de edición del diagrama, las clases y los elementos de estas que se deseen (atributos, métodos y relaciones).

Si se arrastran todas las clases con sus atributos y las relaciones existentes entre ellas, se obtiene un diagrama de clases, como el de la Figura 5.65, en el que, no obstante, se han modificado las cardinalidades para la asociación entre las clases *Figura* y *Punto*. Se mostraban a ambos lados las cardinalidades 0..1, pero deben ser, por un lado, 1 al lado de *Punto* porque toda figura tiene un punto como centro y, por otro lado, * al lado de *Figura*, porque un punto puede ser el centro de más de una figura.

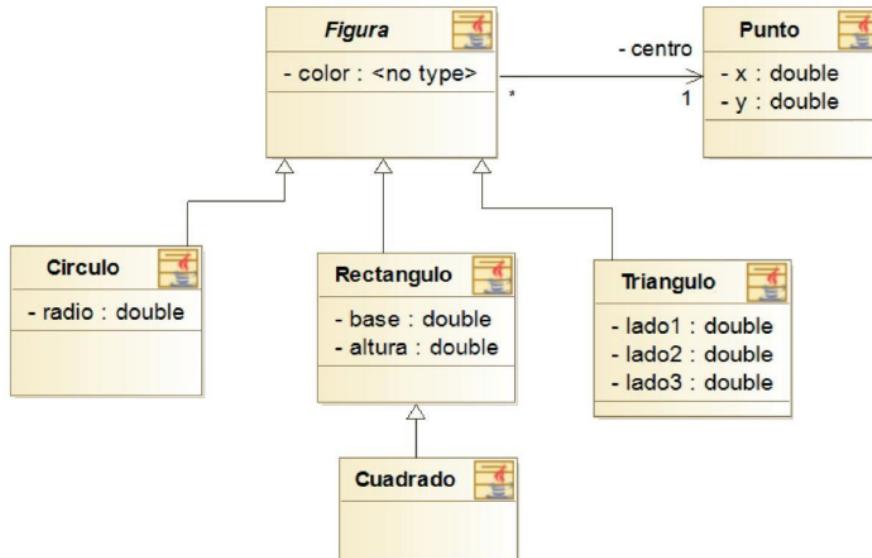
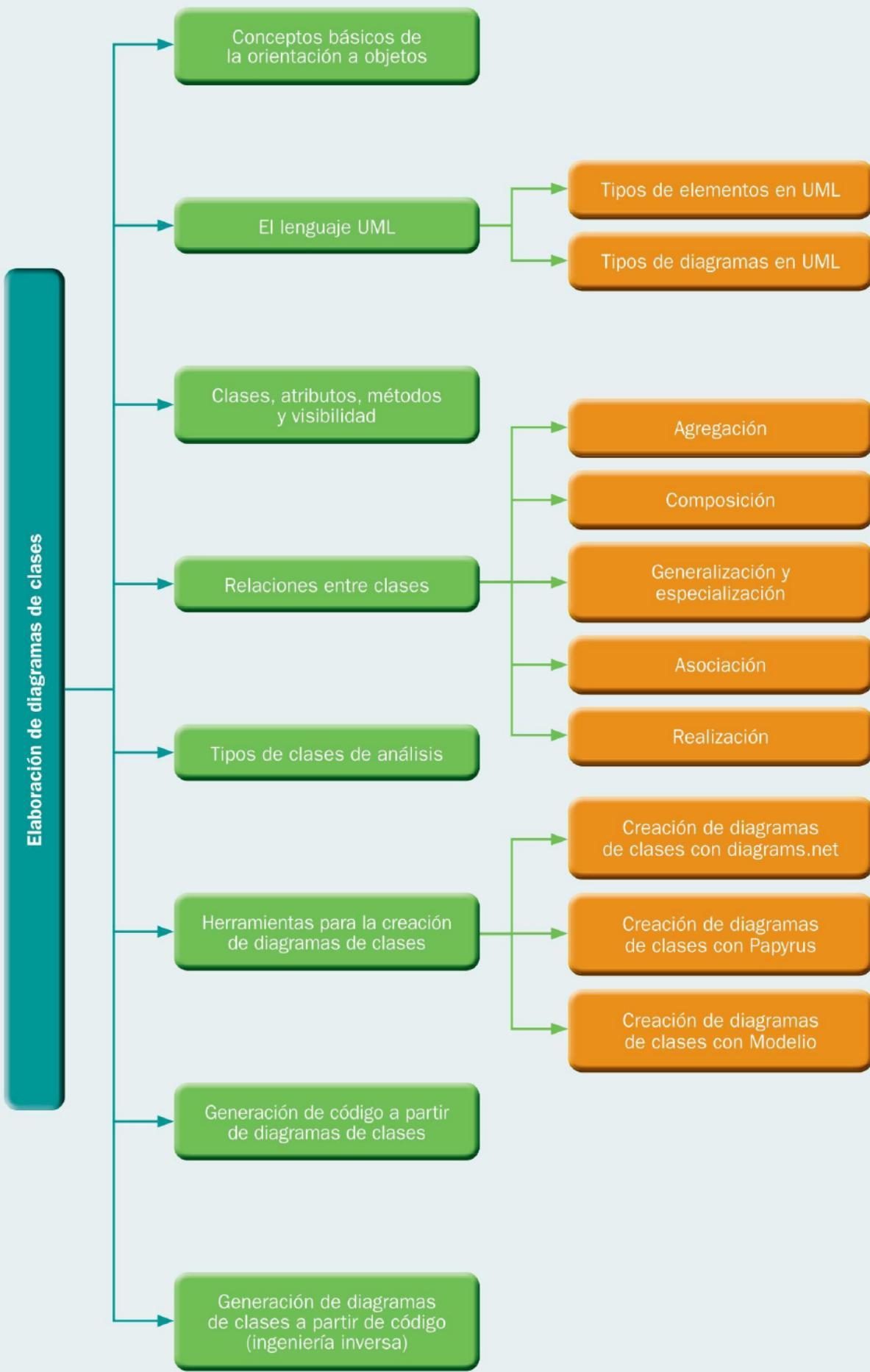


Figura 5.65. Diagrama de clases sin métodos resultado del proceso de ingeniería inversa en Modelio.

Se puede observar que, en el diagrama se ha establecido una relación entre *Figura* y *Punto*, a la que se ha asignado el nombre *centro*, que es el nombre del atributo de la clase *Punto* dentro de la clase *Figura* que indica el punto que constituye el centro de la figura. Además, la punta de flecha al lado de *Punto* indica que esta relación es navegable solo desde *Figura* hasta *Punto*. Esto quiere decir que, dada una figura, es posible conocer el punto que constituye su centro, pero dado un punto, no es posible conocer las figuras para las cuales ese punto es su centro. Esto sería posible si en la clase *Punto* hubiese un atributo de tipo lista con objetos de la clase *Figura*, pero esto no ocurre en este ejemplo, como se puede observar en el código de la clase *Punto*.



Actividades de comprobación

- 5.1.** ¿Cuál es la característica de un lenguaje de programación por la que es posible definir varios métodos para la misma clase con el mismo nombre, pero diferente número y/o tipo de parámetros y/o tipo de valor devuelto?
- a) Polimorfismo.
 - b) Modularidad.
 - c) Encapsulamiento.
 - d) Ocultamiento.
- 5.2.** ¿Cuál de las siguientes características no es imprescindible para que un lenguaje de programación se pueda considerar orientado a objetos?
- a) Herencia.
 - b) Abstracción.
 - c) Recolección de basura.
 - d) Polimorfismo.
- 5.3.** La siguiente definición se corresponde con un componente estructural de UML, ¿de qué elemento se trata?
- «Es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional que, por lo general, dispone de algo de memoria y, con frecuencia, capacidad de procesamiento».
- a) Nodo.
 - b) Artefacto.
 - c) Componente.
 - d) Caso de uso.
- 5.4.** Un diagrama de secuencia es un diagrama de:
- a) Interacción.
 - b) Comportamiento.
 - c) Estructural.
 - d) Las respuestas a y b son correctas.
- 5.5.** ¿Qué diagramas muestran la funcionalidad del software desde el punto de vista del usuario?
- a) Diagramas de casos de uso.
 - b) Diagramas de clases.
 - c) Diagramas de actividades.
 - d) Diagramas de componentes.
- 5.6.** Si a la hora de definir una clase, se quiere que sus atributos sean accesibles desde la propia clase y sus subclases, se tendrá que asignar a los atributos el modificador de acceso:
- a) *Public*.
 - b) *Protected*.
 - c) *Private*.
 - d) *Package*.

- 5.7.** La propiedad por la cual una subclase tiene, además de sus atributos y métodos propios, los atributos y métodos de su superclase, recibe el nombre de:
- a) Jerarquía.
 - b) Herencia.
 - c) Especialización.
 - d) Generalización.
- 5.8.** El tipo de relación entre una clase de tipo compuesto y una clase componente tal que el tiempo de vida de los componentes tiene que coincidir con el del compuesto, recibe el nombre de:
- a) Asociación.
 - b) Agregación.
 - c) Composición.
 - d) Generalización.
- 5.9.** ¿Es posible en un diagrama de clases UML reflejar información que en lugar de ser propia de una clase pertenezca a una asociación entre clases?
- a) No, no es posible.
 - b) Sí, sí es posible y esto se puede llevar a cabo simplemente asignando atributos a la relación.
 - c) Sí, sí es posible y esto se puede llevar a cabo creando una clase asociativa vinculada a la relación, de manera que esta clase contenga los atributos propios de la relación.
 - d) Ninguna de las respuestas anteriores es correcta.
- 5.10.** El proceso de obtener un diagrama de clases a partir del código fuente recibe el nombre de:
- a) Ingeniería.
 - b) Reingeniería.
 - c) Ingeniería inversa.
 - d) Ingeniería invertida.

Actividades de aplicación

- 5.11.** ¿Cuál es la diferencia entre los diagramas de clases y los diagramas de objetos?
- 5.12.** ¿Para qué se usan los diagramas de casos de uso? ¿De qué elementos consta un diagrama de casos de uso?
- 5.13.** ¿En qué consiste la visibilidad de un atributo o de un método? ¿Qué valores puede tomar la visibilidad de un atributo o de un método?
- 5.14.** ¿Cómo se representan en un diagrama de clases las relaciones de grado mayor que 2 (ternarias, cuaternarias, etc.)? ¿Pueden tener estas relaciones atributos asociados? En caso afirmativo, ¿cómo se representa esta información?

- 5.15.** ¿A qué se refieren los conceptos de generalización y especialización?
- 5.16.** ¿Cómo se deben interpretar las multiplicidades o cardinalidades en las relaciones de los siguientes diagramas de clases?

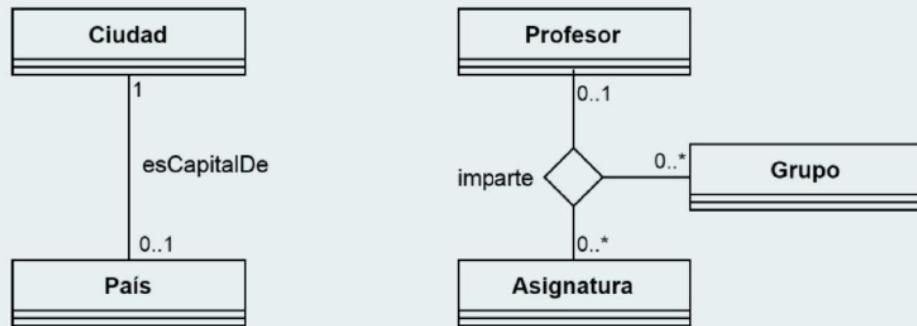


Figura 5.66. Diagrama de clases que refleja, por un lado, la ciudad capital de cada país, y por otro, las asignaturas que imparte cada profesor y el grupo al que imparte cada asignatura.

- 5.17.** ¿Por qué en el siguiente diagrama de clases la línea que une la relación *vende* con la clase *Venta* esta dibujada con un trazado discontinuo? ¿Qué información proporcionan los atributos *stock* y *precio* de la clase *Venta*? ¿Por qué se ha creado la clase *Venta*?

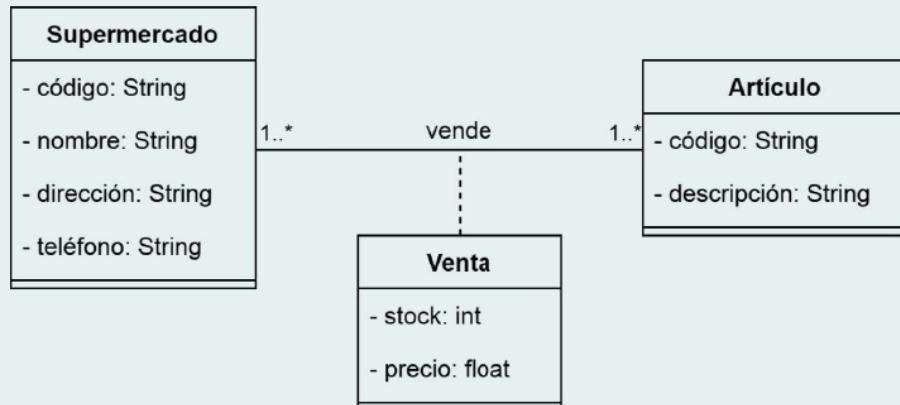


Figura 5.67. Diagrama de clases que refleja los artículos que vende cada supermercado.

- 5.18.** ¿Qué tipos de clases de análisis se pueden identificar en una aplicación? ¿Para qué se usa cada uno de estos tipos de clases?
- 5.19.** Realiza un diagrama de clases sin métodos para una empresa dedicada al alquiler de automóviles, teniendo en cuenta los siguientes requisitos:
- Un determinado cliente puede tener, en un momento dado, varias reservas hechas.
 - De cada cliente, se desea almacenar su DNI, nombre, dirección y teléfono. Además, los clientes se diferencian por un código único.
 - Cada cliente puede ser avalado por otro cliente de la empresa. Un cliente puede avalar a varios.

- Una reserva la realiza un único cliente, pero puede involucrar varios coches.
- Es importante registrar la fecha de inicio y fin de la reserva, el precio de alquiler de cada uno de los coches, los litros de gasolina en el momento de realizar la reserva, el precio total de la reserva y un indicador que informe acerca de si cada coche ha sido entregado o no.
- Todo coche tiene siempre asignado un determinado garaje. De cada coche, se requiere registrar su matrícula, modelo, color y marca.
- Cada reserva se realiza en una determinada agencia. De cada agencia, se requiere registrar su nombre y dirección.
- Sobre cada garaje, es necesario conocer su código único, ubicación y capacidad o número máximo de coches que caben en el garaje.

5.20. Elabora un diagrama de clases sin métodos para un sistema de monitorización de sensores, teniendo en cuenta los siguientes requisitos:

- El sistema monitoriza sensores y también informa sobre posibles problemas.
- Cada sensor lo describe su fabricante y número de modelo, secuencia de iniciación (que se envía al sensor para iniciarla), factor de escala y unidad de medida, intervalo de muestreo, ubicación, estado (encendido, apagado, en reserva), valor actual y umbral de alarma. Hay un tipo especial de sensores, que son los *sensores críticos*. Estos tienen una característica adicional, que es su tolerancia del intervalo de muestreo.
- Los sensores están instalados en edificios. El sistema hace un seguimiento de los sensores de cada edificio. De cada edificio, se conoce su dirección y el número de contacto en caso de emergencia.
- El sistema activa determinados dispositivos de alarma siempre que se supera el umbral de un sensor. Los dispositivos de alarma tienen dos características de las cuales depende su funcionamiento: el estado del dispositivo y la duración de la señal de alarma.
- El sistema registra información de la fecha, hora, gravedad, tiempo de reparación y estado de cada suceso de alarma.

5.21. Realiza un diagrama de clases sin métodos para una entidad bancaria, de la que es necesario saber su CIF, nombre y dirección. Tras una entrevista se ha obtenido la siguiente información:

- *Listado de sucursales*: código de la sucursal, dirección y teléfono.
- *Listado de cuentas corrientes*: número de la cuenta, saldo y, además, NIF, nombre y apellidos y dirección de las personas que pueden utilizar esa cuenta.
- *Listado de operaciones realizables*: código de la operación y descripción.
- *Listado de recibos domiciliados*: número de recibo, número de cuenta de pago, importe del recibo, entidad emisora (empresa que emite el recibo) y fecha.

Las reglas que regulan este sistema son las siguientes:

- El banco tiene muchas sucursales.

- Cada sucursal tiene una serie de cuentas corrientes.
- Cada cuenta corriente tiene asociados uno o más clientes. Sin embargo, es posible que varios clientes con la misma cuenta (por ejemplo, personas de la misma familia: padres e hijos menores) no puedan realizar las mismas operaciones.
- Cada cliente puede tener varias cuentas. En cada una de ellas puede realizar operaciones distintas.
- Cada cuenta puede o no tener uno o varios recibos domiciliados.

5.22. Elabora un diagrama de clases sin métodos para una empresa que pertenece a un grupo nacional de agencias que se dedican a la venta al por mayor, de acuerdo con las siguientes especificaciones:

- Los clientes de la empresa pueden ser mayoristas, minoristas e, incluso, clientes del propio grupo de empresas. Acerca de todos los clientes, se debe conocer su NIF, nombre y apellidos, dirección y teléfono. Además, para los mayoristas se necesita saber el almacén y el número de tarjeta bancaria. En el caso de los minoristas, interesa el número de cuenta bancaria; y en el caso de clientes del propio grupo, la actividad a la que se dedican.
- Cuando un cliente hace un pedido, al que se asigna un número y del que se registra asimismo su fecha, se genera un documento con la relación de los artículos solicitados. De cada artículo se precisa saber su código, nombre, precio, existencias actuales y el número mínimo y máximo de existencias recomendables, así como el número de unidades de ese artículo solicitadas en el pedido. Cada artículo lleva asociado un tipo de IVA, que puede ser del 4, del 10 o del 21 %. Por otra parte, algunos artículos tienen asociado un envase, del que se desea almacenar su código, descripción y precio.
- El pedido puede ser modificado como consecuencia de devoluciones, siniestros, cambios de última hora en el criterio del cliente, etcétera.
- Al final de cada mes, la empresa pone en marcha el proceso de facturación que consiste en crear una factura por cada cliente, en la que se incluyen los pedidos correspondientes a ese mes. Las facturas se numeran y llevan una fecha asociada.
- En los pedidos intervienen los llamados *vendedores* o *comisionistas*, que son empleados de la empresa cuyo propósito es promocionar y vender los artículos. En cada pedido interviene un solo comisionista. Al final de cada mes, reciben sus liquidaciones correspondientes. Cada vendedor-comisionista tiene un porcentaje sobre el total de sus ventas y una determinada antigüedad. Aparte de estos datos, se precisa registrar su NIF, nombre y apellidos, teléfono y NSS (número de Seguridad Social). Se considera venta solamente si se ha facturado.

Actividades de ampliación

- 5.23.** A la hora de representar una asociación entre clases, además de asignar un nombre a la relación, se pueden escribir roles sobre la línea que representa la relación y al lado de cada una de las clases. Busca información sobre los roles y responde a las siguientes preguntas: ¿para qué se usan los roles?, ¿en qué tipos de relaciones se suelen especificar con más frecuencia?
- 5.24.** Además de los tipos de relaciones entre clases que se han estudiado en el Apartado 5.4 de esta unidad, en un diagrama de clases, se pueden reflejar relaciones de dependencia entre clases. Busca información sobre este tipo de relaciones y responde a las siguientes preguntas: ¿qué significado tienen las relaciones de dependencia entre clases?, ¿cómo se representan?
- 5.25.** Encuentra información sobre la restricción *xor* en un diagrama de clases y contesta a las siguientes preguntas: ¿para qué sirve esta restricción?, ¿cómo se representa?
- 5.26.** Busca también información sobre la restricción *subset* en un diagrama de clases. ¿Con qué fin se emplea esta restricción? Indica cómo se representa.
- 5.27.** Averigua en qué consiste la restricción *ordered* en un diagrama de clases. ¿Para qué se emplea? Describe cómo se representa.
- 5.28.** Busca información sobre los tipos de relaciones de generalización y especialización. ¿Qué diferencia a una generalización disjunta de una solapada? ¿En qué se diferencia una generalización total de una parcial?

Enlaces web de interés

-  **DiagramasUML** - <https://diagramasuml.com/>
(Tutorial sobre diagramas UML con numerosos ejemplos)
-  **UML-diagrams.org** - <https://www.uml-diagrams.org/>
(Sitio web en inglés con información sobre todos los diagramas UML)
-  **Diagrams.net** - <https://www.diagrams.net/>
(Sitio web sobre la herramienta de modelado de código abierto app.diagrams.net)
-  **Modelio** - <https://www.modelio.org/>
(Sitio web sobre la herramienta de modelado UML de código abierto Modelio)