Diseño conceptual de bases de datos en UML

Dogram Code

Diseño conceptual de bases de datos en UML

Jordi Casas Roma
Jordi Conesa i Caralt

EDITORIAL UOC

Diseño de la colección: Editorial UOC

Primera edición en lengua castellana: octubre de 2013
Primera edición en formato digital: febrero de 2014

Gran Via de les Corts Catalanes, 872, 3a Planta - 08018
Barcelona
www.editorialuoc.com

Realización editorial: Sònia Poch Masfarré
ISBN: 978-84-9064-122-4

Jordi Casas Roma

Licenciado en Ingeniería Informática por la Universitat Autònoma de Barcelona en 2002, Máster en Inteligencia Artificial Avanzada por la Universidad Nacional de Educación a Distancia en 2011 y actualmente finalizando sus estudios de doctorado sobre privacidad en redes en la Universitat Autònoma de Barcelona. Desde 2009 ejerce como profesor en los Estudios de Informática, Multimedia y Telecomunicación de la Universitat Oberta de Catalunya. Sus intereses de investigación incluyen la privacidad, la minería de datos y la teoría de grafos. Desde 2010 pertenece al grupo de investigación KISON (K-ryptography and Information Security for Open Networks).

Jordi Conesa i Caralt

Doctor en Informática por la Universitat Politècnica de Catalunya desde 2008, donde realizó una tesis doctoral relacionada con modelado conceptual. Desde el 2007 es profesor de la Universitat Oberta de Catalunya. Antes de eso fue profesor asociado del área de bases de datos de la Universitat de Girona. Actualmente ejerce como profesor en las áreas de bases de datos y de ingeniería del software. Sus intereses de investigación se enfocan en el uso del modelado conceptual, las ontologías y diferentes técnicas semánticas para crear aplicaciones que se comporten de manera más "inteligente". Profesionalmente, antes de su vida laboral en la universidad, trabajó como programador, analista y jefe de proyectos de aplicaciones web.

A mi padre, por iniciarme en el camino
que me ha traído hasta aquí.
Jordi Casas Roma

Para ti Neus, por todos los momentos
que este y otros proyectos nos ha robado.
Jordi Conesa i Caralt

Índice

Prólogo

Hoy en día estamos en una sociedad altamente informatizada que requiere del uso de bases de datos para almacenar y procesar la información que se genera. Según recientes estudios, la cantidad de datos digitales generados durante el 2012 alcanzó 2,8 Zettabytes, es decir 1.099.511.627.776 de Gibabytes. Y se prevé que la evolución será aún mayor. Las bases de datos son, hoy en día, el mecanismo más eficiente para almacenar y procesar estos datos. Por tanto, aunque el diseño de bases de datos no sea una nueva tendencia y pueda parecer poco glamurosa para algunos, es un área de vital importancia en el entorno actual. Pensad que un buen diseño de base de datos puede marcar la diferencia entre ser capaces de almacenar y explotar la información de forma adecuada o ser incapaces de hacerlo.

Este libro ataca esta problemática introduciendo al lector en el diseño conceptual de bases de datos. El propósito es que el lector aprenda qué es el diseño conceptual de una base de datos, cuál es su lugar dentro de la creación de la base de datos, y sepa crear el esquema conceptual de una base de datos a partir de un conjunto de requisitos dados. Las otras fases del diseño de base de datos (diseño lógico, diseño físico, implementación y optimización) quedan fuera del alcance de este libro.

El libro se centra en el diseño conceptual de bases de datos empleando el lenguaje UML, aunque su contenido puede

aplicarse fácilmente a otros lenguajes de modelado, como por ejemplo el modelo Entidad-Relación.

En la primera parte del libro el lector verá, de forma breve, las cinco etapas que integran el proceso de diseño de una base de datos, especificando de forma clara los objetivos de cada una de ellas. La segunda parte del libro explica claramente qué es un esquema conceptual, el rol que tiene en el diseño de bases de datos, y los lenguajes de modelado conceptual más usados actualmente. Posteriormente, el lector aprenderá los diferentes elementos de modelado que se utilizan en la modelización conceptual y cómo usarlos para realizar esquemas conceptuales de calidad. Estos son esquemas de alto nivel e independientes de la tecnología de implementación que representan la información que se va a almacenar en la base de datos. El libro utiliza más de medio centenar de ejemplos para facilitar la comprensión de los conceptos explicados. Versiones comentadas de los ejemplos pueden encontrarse en la wiki del libro (http://cv.uoc.edu/webapps/xwiki/wiki/bookdcbduml/).

Como conclusión, creemos que este libro es una buena herramienta para aprender y practicar la creación de esquemas conceptuales de bases de datos.

Introducción

En este libro se presenta la problemática del diseño conceptual de bases de datos. En principio, el libro pretende ser genérico y no limitarse a ningún tipo de base de datos concreto, pero debido a su amplia aceptación y uso, en algunos casos ha sido difícil evitar referencias al diseño de bases de datos relacionales.

El diseño de bases de datos es un proceso complejo, que permite obtener una implementación de una base de datos a partir de los requisitos iniciales de los usuarios del sistema de información. Este proceso guía al diseñador de bases de datos por diferentes etapas, con el objetivo de segmentar un problema de considerable complejidad en diferentes subproblemas de menor complejidad.

En la primera parte del libro veremos brevemente las cinco etapas que forman el proceso de diseño de una base de datos y especificaremos de forma clara los objetivos de cada una de ellas. Estas etapas definen el ciclo de vida de las bases de datos y pueden resumirse como la fase de análisis de requisitos, la creación de un esquema conceptual que permita almacenar la información relevante para la base de datos, la traducción de dicho esquema conceptual a un diseño lógico adaptado al tipo de base de datos a utilizar, la traducción del diseño lógico a un esquema físico que tenga en cuenta en qué máquinas se alojará la base de datos, qué sistema gestor de

base de datos la manejará y qué uso se realizará del sistema. Finalmente, la última etapa será la creación de la base de datos y su optimización.

A continuación, la segunda parte del libro se centra con más detalle en la etapa del diseño conceptual. Este texto se centrará en el diseño conceptual de bases de datos empleando el lenguaje UML. Este proceso permitirá obtener un esquema conceptual independiente de la tecnología que se utilizará en las etapas posteriores. Se usará un ejemplo de envergadura para ejemplificar todos los conceptos que se vayan explicando en el libro.

Al final del libro, el lector sabrá qué es el diseño conceptual de una base de datos, cuál es su lugar dentro de la creación de la base de datos y cómo crear un esquema conceptual de base de datos de un problema concreto. Las otras fases del diseño de base de datos (diseño lógico, físico, implementación y optimización) quedan fuera del alcance de este libro.

Somos conscientes de que la mejor manera de aprender a diseñar bases de datos es practicando. Por ese motivo, hemos creado una versión comentada de algunos de los ejemplos del libro en formato vídeo. Animamos al lector a consultar dichos vídeos para poder clarificar conceptos o aprender paso a paso cómo diseñar esquemas conceptuales. Los vídeos se encuentran en la wiki del libro:

http://cv.uoc.edu/webapps/xwiki/wiki/bookdcbduml/

Capítulo I

Introducción al diseño de bases de datos

1. Introducción

El diseño de bases de datos es un proceso complejo que permite obtener una implementación de una base de datos que satisface los requisitos informacionales de un sistema de información. El diseño se realiza a partir de los requisitos del sistema de información. Este proceso guía al diseñador de bases de datos por varias etapas con el objetivo de segmentar un problema de una complejidad considerable en diferentes subproblemas de menor complejidad hasta llegar a la solución final: la implementación de una base de datos que satisface los requerimientos funcionales y no funcionales de un sistema de información.

Cada uno de los subproblemas identificados corresponde a una de las etapas del proceso de diseño de bases de datos. En este capítulo se describe el proceso global de diseño de bases de datos y las diferentes etapas que lo forman. Este es sólo un texto introductorio al diseño de bases de datos. Así pues, en este capítulo se presenta una visión general de todo el proceso de diseño. Aunque el capítulo se centra en el diseño de bases de datos relacionales, las fases pueden ser

utilizadas en el diseño de bases de datos de cualquier tipo (orientadas a objetos, relacionales orientadas a objetos, etc.).

2. Proceso de diseño de una base de datos

El proceso de diseño de bases de datos consiste en definir la estructura lógica y física de una o más bases de datos para responder a las necesidades de los usuarios con respecto a la información que necesita un sistema de información.

Mediante un proceso de diseño de bases de datos, se define qué información es necesaria para un sistema de información y cómo se relaciona esta información entre sí. Aparte de definir la información necesaria, también se debe tener en cuenta cómo almacenar dicha información para que los sistemas de información puedan funcionar eficientemente. Todas estas tareas forman parte del proceso de diseño de bases de datos. Para poder tomar estas decisiones de la mejor manera, hay que tener en cuenta las necesidades de información de los usuarios en relación con un conjunto concreto de aplicaciones.

Por ejemplo, supongamos que se quiere crear una base de datos para dar soporte al proceso de extracción de dinero desde cajeros automáticos para un determinado banco. Las primeras etapas del diseño de base de datos se encargarán de garantizar que la base de datos contenga la información relevante: sobre las tarjetas de crédito, sus cuentas enlazadas y el saldo de estas. Las últimas etapas del diseño de base de datos

permitirán implementarlas de forma que puedan responder a la pregunta de si una cuenta tiene saldo suficiente en menos de 2 segundos, con independencia de que haya millones de cuentas en el banco.

> Por lo tanto, el diseño de una base de datos es el proceso en el que se define la estructura de los datos que debe tener la base de datos de un sistema de información determinado y cómo se deben almacenar y gestionar estos datos para permitir una explotación eficiente de los mismos por los sistemas de información.

Los requisitos que debe cumplir un sistema de información y la complejidad de la información que se presenta en él provocan que el diseño de una base de datos sea un proceso complicado. Para simplificarlo, es muy recomendable utilizar la estrategia de «divide y vencerás» *(divide and conquer)*[1].

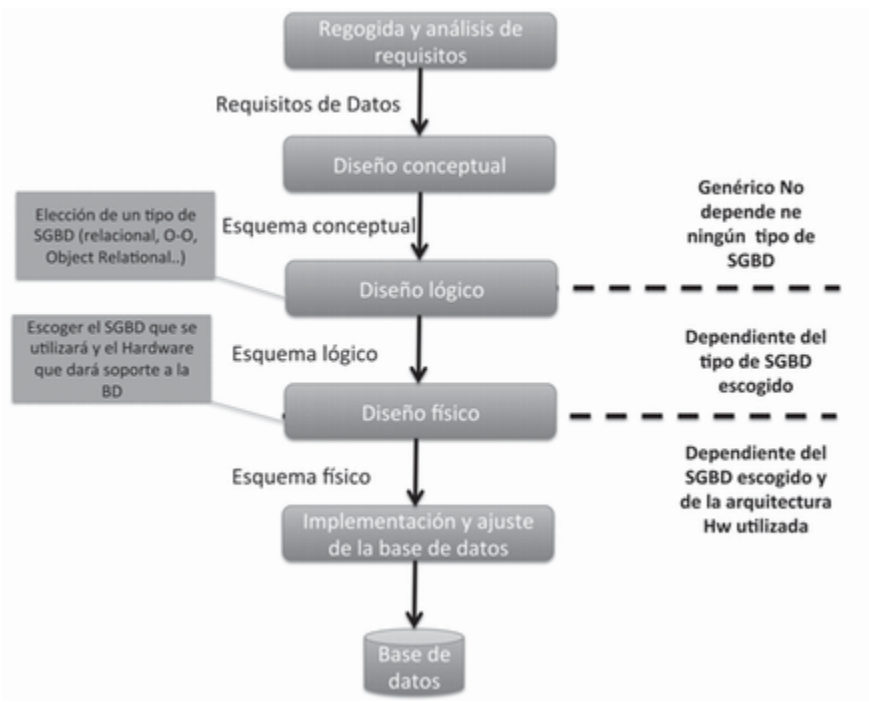
Si se aplica este concepto, se obtienen las diferentes etapas del diseño de bases de datos. Estas son secuenciales y el resultado de cada una sirve de punto de partida de la etapa siguiente. El resultado de la última etapa será la implementación de nuestra base de datos. De este modo, un proceso complejo se descompone en diferentes procesos de menor complejidad. La figura 1 muestra las distintas etapas del diseño de bases de datos.

En primer lugar, la recogida y análisis de requisitos. Esta etapa debe permitir obtener los requisitos y las restricciones

1. Dividir para vencer: La estrategia de «divide y vencerás» propone resolver un problema complejo mediante la subdivisión en un conjunto de problemas más sencillos donde la resolución de los diferentes subproblemas implica solucionar el problema inicial.

de los datos del problema. Para obtener esta información, será necesario mantener conversaciones con los diferentes usuarios de la futura base de datos y de las aplicaciones que estén relacionadas con esta. Solo si se cruzan los requisitos de los diferentes perfiles de usuarios será posible establecer un marco completo de requisitos y restricciones de los datos relacionados con la futura base de datos.

Figura 1. Etapas del diseño de bases de datos



A continuación, se inicia el diseño conceptual. En esta etapa se analizan los requisitos obtenidos en la etapa anterior para identificar la información necesaria para el sistema de información. Una vez detectada, se plasma en un esquema

conceptual. El esquema conceptual es una especificación gráfica de los datos necesarios para un sistema de información y las restricciones asociadas a dichos datos.

Hasta esta etapa del diseño de bases de datos todavía no ha sido necesario escoger el tipo de base de datos (relacional, orientada a objetos, documental, etc.) ni el sistema gestor de bases de datos (SGBD)[2] que se utilizará o el lenguaje concreto con el que se implementará la base de datos. Por tanto, el trabajo realizado hasta este punto será aprovechable sea cual sea el tipo de base de datos que se quiera crear.

En el momento en el que se inicia la tercera etapa del proceso de diseño, el diseño lógico, hay que determinar el tipo de base de datos[3] que se utilizará. Algunos ejemplos de tipos de bases de datos que se podrían tener en cuenta son los siguientes: Relacional, Orientada a Objetos, Object-Relational, NoSQL, Geográfica, XML y Multidimensional. Es importante destacar que en este paso NO se está escogiendo un producto de base de datos concreto, como pueden ser Oracle, Sql Server, MySQL, PostgreSQL, VoldDB o MariaDB, sino un tipo de base de datos. En esta etapa el esquema conceptual se convierte en un esquema lógico adecuado al tipo de bases de datos que se pretende usar.

---

2. Sistema gestor de bases de datos: un sistema gestor de bases de datos (SGBD; en inglés, *database management system*, DBMS) es un tipo de software específico que sirve de interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan.
3. Por *tipos de bases de datos* entendemos los diferentes grupos de bases de datos según el modelo de datos que aplican. Actualmente hay varios tipos de SGBD, entre los cuales los más utilizados son las bases de datos relacionales, orientadas a objetos, documentales, geográficas, NoSQL o multidimensionales. Por ejemplo, las bases de datos relacionales son el conjunto de todos los SGBD que aplican modelos de datos relacionales y son las más utilizadas a día de hoy.

En este punto, y antes de iniciar la etapa de diseño físico, hay que elegir un SGBD concreto sobre el que se pretende implementar la base de datos. La etapa de diseño físico adapta el esquema lógico a las necesidades específicas de un SGBD concreto y, posteriormente, ajusta algunos parámetros para el correcto funcionamiento de la base de datos. Por *base de datos concreta* o *SGDB concreto* entendemos una aplicación concreta de bases de datos. En el caso de bases de datos relacionales, ejemplos de SGBD concretos son Oracle Database, Mysql, SQL Server o IBM Informix, entre otros. También se deberá tener en cuenta la arquitectura hardware que dará soporte a la base de datos y el uso esperado de la base de datos, ya que no es lo mismo implementar una base de datos en un ordenador personal que en un *cloud* de ordenadores con discos RAID, ni crear una base de datos de escritorio que una base de datos multiusuario con acceso 7x24.

Finalmente, la última etapa es la implementación y optimización de la base de datos. Esta etapa permite cargar los datos y posteriormente ajustar algunos parámetros del modelo físico para optimizar el rendimiento de la base de datos.

Estas etapas de diseño no hay que seguirlas estrictamente de manera secuencial[4], y en muchos casos es habitual rehacer el diseño de la etapa anterior a partir de necesidades detectadas en fases posteriores. Estos bucles de retroalimentación son habituales y permiten afinar los diseños de las distintas etapas de una manera iterativa.

4. De hecho, según algunos autores, la frontera entre el diseño físico y la implementación y optimización es difusa e incluso inexistente.

El proceso que muestra la figura 1 se basa en el modelo de diseño orientado a datos. Este modelo se centra en el diseño de los contenedores de la información y en la estructura de la base de datos. Paralelamente a este modelo existe el modelo de diseño orientado a procesos, que se centra en las aplicaciones de bases de datos para determinar los datos y el uso que de estas hacen las aplicaciones. Tradicionalmente, el diseño de aplicaciones se ha basado en este segundo modelo, pero cada vez resulta más claro que ambas actividades son paralelas y que están estrechamente interrelacionadas. Las herramientas de diseño de bases de datos y de aplicaciones se combinan cada vez con mayor frecuencia.

3. Fases del diseño de una base de datos

A continuación veremos con algo más de detalle las fases que forman el proceso de diseño de una base de datos.

3.1. Fase 1. Recogida y análisis de requisitos

La primera fase en el diseño de una base de datos consiste en conocer y analizar con detalle las expectativas, las necesidades y los objetivos de los futuros usuarios de la base de datos. Este proceso se denomina *recogida y análisis de requisitos*.

La fase de recogida y análisis de requisitos se puede dividir en tres subfases secuenciales: la recogida de requisitos; la estructuración y el refinamiento de los requisitos, y la formalización de los requisitos.

3.1.1. Recogida de requisitos

Para determinar los requisitos, en primer lugar hay que establecer los actores del sistema de información que interaccionarán con la base de datos. Esto incluye a los usuarios y las aplicaciones, tanto si son nuevos como si no lo son. Normalmente, un grupo de analistas se encarga de hacer el análisis de requisitos. En la mayoría de los casos, este análisis suele ser informal, incompleto e incluso incoherente en algún punto. Por lo tanto, hay que dedicar muchos esfuerzos a trabajar esta información y convertirla en una especificación lo más formal posible, para evitar ambigüedades. La especificación resultante será la base que los diseñadores utilizarán para modelar e implementar el sistema de información.

En esta fase, no solo hay que recoger y analizar los requisitos referentes a la estructura o a la forma de la información (tipos de datos y relaciones entre ítems de datos), sino que hay que capturar y analizar cualquier tipo de requisito. En el caso particular de la base de datos, hay que recoger, analizar y documentar cualquier requisito que los usuarios esperen de la base de datos. Esto incluye identificar los procesos que se deben ejecutar sobre la base de datos y estimar con qué frecuencia se ejecutarán, las restricciones a implementar sobre los datos, las restricciones sobre el rendimiento del sistema de información, las restricciones relativas a la implementación (tanto en lo que

se refiere al hardware como en lo que se refiere al software), requisitos de seguridad o de rendimiento (por ejemplo, tiempo de respuesta) y volumen[5] esperado de los datos, entre otros.

Algunas de las actividades más habituales de esta fase son las siguientes:

- Identificar los grupos de usuarios y las principales áreas de aplicación que utilizarán de la base de datos. Dentro de cada grupo, hay que elegir usuarios clave y formar comités para llevar a cabo la recopilación y la especificación de requisitos.
- Estudiar y analizar la documentación existente relativa a las aplicaciones en uso.
- Estudiar el entorno actual y el uso que se quiere dar a la información. Esto incluye el estudio de las entradas, el flujo y las salidas de información, además de las frecuencias y los usos de las diferentes tareas dentro del sistema de información.
- Hacer entrevistas y encuestas a los futuros usuarios para que puedan manifestar su opinión y sus prioridades acerca del nuevo sistema de información.

Al final de esta etapa se tendrá una lista preliminar y no formalizada de requisitos del sistema a implementar.

5. Por volumen de los datos entendemos el número de registros que tendrá cada concepto almacenado en la base de datos. Un ejemplo podría ser «se estima que la base de datos contendrá un millón de clientes, dos millones de cuentas corrientes y un millón y medio de tarjetas de crédito».

3.1.2. Estructuración y refinamiento de los requisitos

Se debe tener en cuenta que algunos de estos requisitos, muy probablemente, cambiarán durante el proceso de diseño y que hay que estar atentos y en contacto permanente con los usuarios de la base de datos para detectar posibles problemas. Es una buena práctica incorporar a los usuarios de la base de datos durante el proceso de desarrollo, puesto que así se incrementa su grado de implicación y satisfacción. Hay algunas propuestas de metodologías para la recogida y el análisis de requisitos basadas en el trabajo conjunto de los desarrolladores con los usuarios de la base de datos, como, por ejemplo, el diseño conjunto de aplicaciones (JAD)[6].

3.1.3. Formalización de los requisitos

El paso siguiente es convertir los requisitos a un formato estructurado mediante técnicas de especificación de requisitos como, por ejemplo, el análisis orientado a objetos (OOA)[7], diagramas de flujo de datos (DFD)[8], el lenguaje i*[9] o la notación Z[10]. Estas técnicas utilizan diferentes tipos de recursos (diagramas, texto, tablas, gráficos, diagramas de decisión, etc.) para organizar y representar los requisitos de forma clara.

---

6. JAD es la sigla de la expresión inglesa *joint application design*.
7. OOA es la sigla de la expresión inglesa *object oriented analysis*.
8. DFD es la sigla de la expresión inglesa *data flow diagrams*.
9. i* es un lenguaje de modelado de requisitos. Se puede encontrar más información de i* en http://www.cs.toronto.edu/km/istar/
10. La notación Z es un lenguaje formal utilizado en ingeniería del software.

Esta fase puede representar un coste importante dentro del proceso de diseño de una base de datos, pero es muy importante y puede ser determinante para el éxito o el fracaso del sistema de información. Detectar y corregir los errores o problemas en las fases iniciales del proyecto es mucho menos costoso que arrastrar los errores hasta las fases finales, cuando corregirlos tendrá unos costes mucho más importantes. La satisfacción del usuario final vendrá determinada por la capacidad de recoger y captar sus necesidades e implementarlas de manera correcta en la solución final. Además, en caso de ser comprensibles por el cliente, los requisitos formalizados pueden ser un buen documento de compromiso entre el cliente y el diseñador de la base de datos, ya que contiene toda la información sobre lo que hay que hacer y, si están bien formalizados, dan poco margen (o ninguno) a ambigüedad.

3.2. Fase 2. Diseño conceptual

> La fase de **diseño conceptual** tiene como objetivo crear un esquema conceptual de alto nivel e independiente de la tecnología a partir de los requisitos, las especificaciones y las restricciones que se han recogido en la fase anterior.

En esta fase se parte de la recogida y el análisis de requisitos obtenidos en la fase anterior y tiene como objetivo diseñar un esquema conceptual de la base de datos que sea consistente con los requisitos, las especificaciones y las restricciones impuestas por la problemática que hay que resolver. Eso quiere decir que represente la información necesaria para la base de datos, junto con sus restricciones de integridad.

Un esquema conceptual es una representación gráfica que describe el conocimiento general sobre un dominio que un sistema de información debe saber para poder llevar a cabo sus funciones. Para crear un esquema conceptual, los diseñadores deben conocer muy bien el dominio del sistema de información (de aquí la necesidad de disponer de una buena lista de requisitos formalizados) y tener una gran capacidad de abstracción. Y aun cuando estas dos condiciones se cumplen, el éxito no está garantizado.

Los esquemas conceptuales describen conocimiento y, por tanto, son modelos de alto nivel y no incluyen detalles de implementación. Un esquema conceptual, además, debe servir de referencia para verificar que se han agrupado todos los requisitos y que no hay ningún conflicto entre ellos. De hecho, según el lenguaje utilizado para modelar el esquema conceptual, podemos encontrar herramientas que permiten analizar su validez y calidad, como por ejemplo la herramienta USE[11] que permite evaluar si un esquema UML es correcto y si sus restricciones de integridad son satisfactibles.

En esta fase del diseño todavía no se considera el tipo de base de datos que se utilizará. Y, por lo tanto, tampoco se considera el SGBD ni el lenguaje concreto de implementación de la base de datos. En esta etapa nos concentraremos en la estructura de la información, sin resolver de momento cuestiones relacionadas con la tecnología.

Hay varios modelos de datos de alto nivel que permiten modelar los requisitos, las especificaciones y las restricciones que se han obtenido en la primera fase del diseño de una base

11. Más información de la herramienta USE se puede encontrar en http://sourceforge.net/projects/useocl/

de datos. Estos modelos disponen normalmente de lenguajes gráficos para facilitar la representación y la comprensión de sus esquemas conceptuales. Quizás hoy en día los modelos de datos más utilizados en el diseño de bases de datos son ER y UML:

- **Modelo ER**[12]: Uno de los más conocidos y utilizados es el modelo entidad-interrelación, sobre todo en el diseño conceptual de bases de datos, principalmente debido a su simplicidad y facilidad de uso. Los elementos básicos que incluye el modelo son los tipos de entidad, los atributos y los tipos de relación entre entidades. El objetivo principal del modelo ER es permitir a los diseñadores reflejar en un modelo conceptual los requisitos del mundo real que sean de interés para la problemática a resolver. El modelo ER facilita el diseño conceptual de una base de datos y, como ya hemos comentado, es aplicable al diseño de cualquier tipo de bases de datos.
- **El lenguaje unificado de modelado**[13] (UML) es un lenguaje gráfico diseñado para especificar, visualizar, modificar, construir y documentar un sistema de información. El lenguaje UML incorpora una gran cantidad de diagramas que permiten representar el sistema desde diferentes perspectivas. En relación con el

12. En inglés se denomina *entity-relationship model*. Dada la ambigüedad de la traducción, algunos autores lo traducen como modelo entidad-relación y otros como modelo entidad-interrelación. Ambos conceptos se refieren al mismo modelo.
13. El lenguaje unificado de modelado (en inglés, *unified modeling language*) es un lenguaje de propósito general para modelar sistemas de software. Este estándar fue creado, y actualmente es mantenido, por el *Object Management Group* (OMG).

diseño conceptual de bases de datos, interesa especialmente el diagrama de clases, que permite representar información del dominio del sistema de información que se va a implementar. Los diagramas de clases son diagramas estáticos que describen la estructura de un sistema a partir de las clases o tipos de entidad del sistema, sus atributos y las asociaciones o tipos de relación que se establecen entre ellos. Estos diagramas han mostrado una capacidad excelente para el modelado de datos. Por este motivo, son cada vez más importantes en el diseño conceptual de bases de datos.

3.3. Fase 3. Diseño lógico

Previamente a la fase de diseño lógico, se debe elegir un tipo de base de datos. Es decir, no hay que escoger todavía un SGBD concreto, sino simplemente seleccionar el tipo de base de datos que se quiere implementar. Es importante que quede claro que el tipo de base de datos determina el esquema de diseño lógico. Una vez elegido el tipo de SGBD donde se quiere implementar la base de datos, ya se puede iniciar la fase del diseño lógico.

En la fase de **diseño lógico** se transforma el modelo conceptual, independiente del modelo de datos que se utilizará en la base de datos, en un modelo lógico dependiente del modelo de datos (o tipo de SGBD) en el que se implementará la base de datos.

La transformación traducirá el modelo considerando el tipo de SGBD en el que se quiere implementar la base de

datos. Por ejemplo, si se quiere crear la base de datos en un sistema relacional, esta etapa obtendrá un conjunto de relaciones con sus atributos, claves primarias y claves foráneas correspondientes. Mientras que si la base de datos se quiere crear en un sistema orientado a objetos, se identificarán las diferentes clases que se van a crear, sus atributos, relaciones, restricciones de integridad y la jerarquía de especializaciones entre clases y, posiblemente, entre relaciones.

El resultado de esta etapa será un modelo lógico de la estructura de la información. Generalmente, cuando este modelo lógico hace referencia a un SGBD relacional, se denomina modelo relacional.

El diseño lógico puede dividirse en tres subfases, que se aplican de manera secuencial:

- **Reconsideraciones del modelo conceptual**: en esta primera parte se realiza un análisis en profundidad del modelo conceptual obtenido en la fase anterior, con la intención de detectar y corregir algunos errores que se suelen producir en los modelos conceptuales y que conviene detectar y reparar lo antes posible para evitar que se propaguen en fases posteriores. Dichos errores se denominan *trampas de diseño*. Es conveniente conocer cada uno de estos errores y asegurarse de que el modelo conceptual que se quiere transformar está libre de ellos antes de continuar con el diseño lógico.
- **Transformación del modelo conceptual en el modelo lógico**: el esquema conceptual se reescribe al modelo de datos del tipo de SGBD escogido. Por ejemplo, en el caso de querer utilizar un SGBD relacional,

el esquema conceptual se reescribe utilizando tablas, atributos, claves primarias y claves foráneas.

- **Normalización**: la teoría de la normalización es conocida por su importancia en el diseño de bases de datos relacionales. En los modelos relacionales, la normalización aplica la teoría de conjuntos, la lógica y el álgebra relacional para formalizar un conjunto de ideas simples, que guían un buen diseño. La teoría de la normalización utiliza las formas normales (FN) para reconocer los casos en los que no se aplican buenos criterios de diseño. Una relación está en una forma normal determinada si satisface un conjunto de restricciones específicas que son propias de esta forma normal. La infracción de estas restricciones origina que la relación tenga un conjunto de anomalías y redundancias de actualización no deseables. Las formas normales son declarativas, es decir, cada forma normal indica las restricciones que se deben satisfacer, pero no describe ningún procedimiento para conseguirlo.
  Aunque la teoría de la normalización está muy ligada al modelo relacional, su filosofía es aplicable también a otros modelos. Por ese motivo existen algunas propuestas de normalización en otros modelos de bases de datos, como por ejemplo en los modelos de SGBD orientados a objeto.

3.4. Fase 4. Diseño físico

Previamente a la fase de diseño físico, hay que elegir un SGBD concreto. Deben estudiarse los diferentes sistemas

comerciales o libres que hay en el mercado y seleccionar un SGBD donde se pueda implementar la base de datos que se ha ido gestando en las fases anteriores del proceso de diseño. La elección del SGBD estará condicionada por el tipo de SGBD que se haya elegido en la etapa de diseño lógico.

> El **diseño físico** de una base de datos es un proceso que, a partir de un diseño lógico y de una estimación sobre el uso esperado de los datos de la base de datos, creará una configuración física de la base de datos adaptada al entorno donde se alojará y que permita el almacenamiento y la explotación de los datos con un rendimiento adecuado.

De la definición anterior se puede extraer que también deberá tenerse en cuenta el hardware donde se ubicará la base de datos y las características de los procesos que consultan y actualizan la base de datos, como por ejemplo las frecuencias de ejecución y los volúmenes que se espera tener de los diferentes datos que se quieren almacenar con el fin de conseguir un buen rendimiento de la base de datos.

El objetivo del diseño físico es obtener un buen rendimiento de la base de datos en un entorno real. El rendimiento se refiere, básicamente, a:

- **Tiempo de respuesta**. Es el tiempo que transcurre desde que se envía una petición al SGBD hasta que este devuelve los datos de la respuesta. Una parte importante de este tiempo está bajo el control del SGBD y hace referencia al tiempo de acceso por parte del SGBD a los datos requeridos para generar la respuesta. Otros aspectos no son controlados por el

SGBD, como por ejemplo la planificación del sistema operativo o los tiempos de acceso a los medios físicos de almacenamiento de los datos, aunque el modelo físico que definamos podrá condicionar fuertemente este último, ya que permite decidir en qué discos se deben alojar determinados datos, de qué forma (orientados a columna o a fila, ordenados o no, etc.) y el tamaño de las páginas de datos, entre otros.

- **Uso del espacio**. Es la cantidad de espacio de disco utilizado por los ficheros de la base de datos y las estructuras de rutas de acceso al disco, incluyendo índices y otras rutas de acceso.
- **Carga de transacciones**. Es la cantidad media de transacciones que se pueden procesar en un minuto de tiempo. Este factor puede ser crítico para sistemas transaccionales, como por ejemplo líneas aeronáuticas o entidades bancarias.
- **Disponibilidad de la base de datos**. En algunos casos es importante asegurar que la base de datos será resistente a caídas y fallos de funcionamiento. Por ejemplo, en el caso de una entidad bancaria se debe garantizar que los datos serán accesibles 7x24 y que si en algún momento la base de datos deja de funcionar, el tiempo de recuperación será mínimo. Cabe tener en cuenta que muchos negocios hoy en día deben su existencia al tratamiento automatizado de los datos y que sin ellos no pueden funcionar. Pensad, por ejemplo, qué afectaciones habría si un banco no pudiera acceder a sus datos durante 2 días seguidos: operaciones de oficina inoperables, cajeros automáticos no operativos, desconocimiento de qué cuentas corrientes hay y con qué saldos, etc.

Los componentes físicos que forman cada SGBD son específicos. Los fabricantes utilizan estrategias y tecnologías diferentes para maximizar el rendimiento de sus sistemas gestores de bases de datos. Incluso el lenguaje utilizado para definir el modelo físico varía de fabricante en fabricante. Eso es debido a que no existe ningún estándar para definir las construcciones adecuadas para este nivel, a diferencia a lo que pasa en los niveles anteriores. Por ejemplo, el estándar SQL incorpora la definición de todos los componentes del diseño lógico de la base de datos, más las operaciones de acceso a los datos de la base de datos, ya sea para consulta o para inserción/actualización. En cambio, no contiene ningún elemento del diseño físico. Sin embargo, existe un gran parecido entre las construcciones utilizadas por los diferentes gestores, ya que todos ellos permiten trabajar con los mismos elementos de diseño físico. Por ejemplo, en el caso relacional, los elementos de diseño físico que se utilizan son *tablespaces*, índices, vistas materializadas y particiones.

Por lo tanto, habrá que adaptar el esquema lógico obtenido en el paso anterior, teniendo presentes las características de cada sistema gestor. El diseñador debe considerar los aspectos de implementación física y de eficiencia que dependen específicamente del SGBD elegido. Por ejemplo, en el caso de utilizar un SGBD relacional, partiríamos de las definiciones de tablas (con toda la información relacionada; es decir, atributos, claves primarias, claves foráneas y claves alternativas). A continuación, se relacionaría cada elemento con un espacio adecuado en el nivel virtual y, finalmente, cada espacio virtual con un fichero físico, que constituye el nivel físico del sistema de información.

3.5. Fase 5. Implementación y optimización

La última etapa es la implementación y la optimización de la base de datos.

> La etapa de **implementación y optimización** consiste en realizar la carga de los datos y posteriormente ajustar algunos parámetros relacionados con el modelo físico de la base de datos para optimizar el rendimiento.

El objetivo principal de esta etapa es optimizar el rendimiento de la base de datos. En primer lugar, hay que realizar la carga de los datos, puesto que no es posible optimizar el acceso a los datos sin poder determinar el tamaño de las tablas, los tipos de accesos y consultas, la frecuencia de estas, etc.

Finalmente, también habrá que concretar los diferentes roles de usuarios y aplicaciones para poder determinar los permisos de los diferentes grupos. El componente de gestión de la seguridad y las vistas permiten limitar los accesos y de este modo reducir el riesgo de problemas derivados de accesos no autorizados.

3.5.1. Procesamiento y optimización de consultas

El objetivo de la optimización de consultas es crear las estructuras físicas necesarias para mejorar el tiempo de respuesta de una base de datos. Normalmente, la optimización de consultas se realiza mediante la creación de índices, que son estructuras que permiten mantener un índice ordenado

de acuerdo con uno o más campos de la base de datos. Los índices permiten reducir el tiempo de consulta cuando se filtra información de acuerdo con los campos indexados. Un ejemplo muy común de índices que se encuentran en los libros es el índice por capítulos, que a partir del número y el título del capítulo permite acceder rápidamente a su contenido.

Cuando la optimización de consultas se plantea en la etapa de diseño físico se puede abordar mediante otros medios, como por ejemplo el almacenamiento de determinados datos en discos de mayor velocidad, el almacenamiento ordenado de los datos, la creación de vistas materializadas o la segmentación de datos.

La optimización de consultas es un aspecto muy importante que hay que considerar cuando se diseña y se construye un SGBD relacional. Las técnicas que se utilizan para optimizar consultas condicionan el rendimiento global del sistema, puesto que determinan el tiempo que necesita el sistema gestor para resolver las consultas de los usuarios. No obstante, la mayoría de las técnicas utilizadas para la optimización de consultas son un arma de doble filo, que puede empeorar el rendimiento de la base de datos si no se utilizan convenientemente. Por ejemplo, los índices incrementan sustancialmente la velocidad de consulta, pero deben actualizarse cada vez que se añade un registro en la base de datos relacionado con el índice. Eso hace que, en determinados casos, cuando hay una frecuencia de actualización de datos muy alta, el uso de los índices puede ser contraproducente. Este sería el caso, por ejemplo, de los saldos de una cuenta corriente en una base de datos bancaria. Si se creara un índice sobre los saldos de una base de datos, cada vez que se modificara un saldo se tendría

que recalcular el índice. Por tanto, en este caso probablemente se descartaría dicho índice.

Las técnicas de optimización de consultas toman incluso mayor relevancia en SGBD donde las consultas utilizan lenguajes declarativos. Los lenguajes declarativos permiten definir qué información se quiere obtener, pero sin decir cómo debe obtenerse. Es decir, se especifica el resultado que se quiere obtener a partir de una consulta realizada, en lugar de determinar el algoritmo o el método que hay que usar para obtener el resultado. Por ejemplo, el lenguaje SQL empleado por las bases de datos relacionales es declarativo.

Cuando los SGBD utilizan lenguajes declarativos, deben evaluar sistemáticamente las posibles estrategias alternativas que se pueden utilizar para acceder a los datos, y elegir la que se considera óptima. El procesamiento de consultas recoge todo el conjunto de actividades realizadas por el SGBD, que tienen como objetivo la extracción de información de la base de datos para lograr la estrategia más eficiente y proporcionar un mejor rendimiento del sistema de información.

Los SGBD relacionales disponen de técnicas para consultar los planes de resolución de consultas que van a utilizar. Estos planes proporcionan información muy útil para el diseñador de la base de datos, ya que muestran cuántos accesos a disco se realizan, cómo se combinan los datos, y si se utilizan índices u otras estructuras de soporte. Con esta información, el diseñador de la base de datos puede identificar potenciales mejoras y crear estructuras alternativas de datos (índices, vistas materializadas, etc.) para optimizar la consulta.

Este punto es uno de los más importantes que se deben tener en cuenta cuando se diseña un SGBD relacional, pues-

to que la opción elegida afecta directamente al rendimiento global del sistema.

3.5.2. Administración de la seguridad

Finalmente, hay que tener en cuenta las técnicas que se emplean para proteger la base de datos de los accesos no autorizados y los mecanismos para asignar y revocar privilegios a los diferentes usuarios. De estas y otras acciones se encarga el componente de seguridad del SGBD. Este componente deviene cada día más importante, dado que en la actualidad una gran cantidad de ordenadores y otros tipos de dispositivos están interconectados y, por lo tanto, cualquier persona podría convertirse en usuario, y posible atacante, de una base de datos.

Hacer un uso adecuado de las vistas es otro aspecto clave para gestionar la seguridad. Las vistas son estructuras lógicas que permiten acceder a la información de la base de datos a partir de una consulta predefinida. Por tanto, con las vistas pueden generarse diferentes visiones de los datos adaptadas a distintos usuarios, puesto que podemos ocultar información a determinados usuarios y mantener la visión del usuario independientemente de la evolución que tenga la base de datos. Así pues, utilizando vistas se podría hacer que los trabajadores del departamento de contabilidad de una empresa solo pudieran ver los datos de pago de un cliente (nombre, NIF y cuenta corriente), mientras que los trabajadores de logística pudieran ver su dirección y datos de entrega (nombre, persona de contacto, teléfono, dirección de entrega, etc.), así como que los comerciales del área

geográfica de Girona solo puedan acceder a los clientes de esa zona.

En muchas organizaciones, la información es un activo intangible y de naturaleza sensible, que tiene un valor muy importante. Para preservar esta información, hay que proteger el sistema de información y conocer las obligaciones legales que hay que cumplir.

Capítulo II

Diseño conceptual de bases de datos

Tal y como se ha descrito en el primer capítulo, el diseño conceptual es una de las etapas que se deben realizar en el diseño de una base de datos y tiene como objetivo la creación de un esquema conceptual que represente la información que debe almacenarse en la base de datos.

> Un **esquema conceptual** es una representación gráfica que describe el conocimiento general sobre un dominio que un sistema de información debe saber para poder llevar a cabo sus funciones. El esquema conceptual se crea a partir de un análisis de las especificaciones recogidas en la fase de análisis de requisitos e incluye descripciones detalladas de las entidades que están involucradas en el sistema de información, las relaciones entre estas entidades y las restricciones de integridad que se deben aplicar sobre los datos.

En esta etapa el foco de atención se centra en identificar la información relevante y plasmarla en un esquema conceptual, dejando por resolver las cuestiones ligadas a la tecnología para las etapas posteriores al diseño conceptual. Por tanto, en esta etapa se obtiene una estructura de la información genérica, es decir, independiente del tipo de base de

datos que se utilizará (relacional, orientada a objetos, etc.), del sistema gestor de bases de datos que se usará o del lenguaje concreto en que se implementará la base de datos.

El esquema conceptual, además, debe servir como referencia para verificar que se han considerado todos los requisitos y que no hay conflicto entre ellos. Si algunos de los requisitos iniciales no se pueden representar gráficamente en el esquema conceptual, deben añadirse de manera textual, para asegurarnos de que quedan recogidos en el modelo conceptual y de que se tendrán en cuenta en las fases siguientes del diseño de la base de datos.

El esquema conceptual resultante del proceso de diseño conceptual se representa mediante algún modelo de datos de alto nivel. Existen multitud de modelos de datos que pueden utilizarse para representar esquemas conceptuales, como por ejemplo ER, EER, UML y ORM. Estudiaremos más a fondo cómo deben ser estos modelos y los lenguajes de modelado más utilizados a día de hoy en las secciones posteriores.

Concretamente, en este capítulo explicaremos qué características deben cumplir los lenguajes de modelado conceptual y los más importantes para el diseño de bases de datos, posteriormente explicaremos los principios del diseño conceptual y mostraremos cómo diseñar esquemas conceptuales de bases de datos utilizando el lenguaje UML.

1. Lenguajes de modelado conceptual

Tal y como se ha comentado, hay diferentes lenguajes que permiten representar esquemas conceptuales. Algunas características importantes que deben satisfacer estos lenguajes son:

- **Expresividad**: deben ser suficientemente expresivos para permitir representar los diferentes conceptos del mundo real y las relaciones entre estos conceptos.
- **Simplicidad**: los esquemas resultantes deben ser simples y fáciles de entender. Los usuarios de la base de datos tienen que poder entender los conceptos que se expresan.
- **Representación diagramática**: el modelo debe utilizar una notación diagramática fácil de entender y útil para visualizar el esquema conceptual.
- **Formalidad**: la representación del modelo debe ser precisa, formal y no puede presentar ambigüedades.

Satisfacer estas características no es fácil y a menudo algunas de ellas entran en conflicto con las demás.

Quizás el modelo más conocido y utilizado hasta la fecha es el modelo entidad-interrelación, también conocido como modelo ER. No obstante, actualmente el uso del lenguaje unificado de modelado (UML) ha tomado gran fuerza en el diseño de esquemas conceptuales de bases de datos.

El lector se habrá dado cuenta de que estamos utilizando los términos modelo y lenguaje de forma indistinta. Es cierto que, en general, los términos modelo y lenguaje representan cosas distintas: los modelos representan abstracciones de la realidad

y los lenguajes son los artefactos usados para representar los modelos. En el área de base de datos, no obstante, los modelos (o modelos de datos) se definen como una colección de conceptos, relaciones y reglas utilizados para definir esquemas conceptuales. Por tanto, en el área de bases de datos podemos utilizar los términos modelo y lenguaje de forma indistinta.

A continuación estudiaremos los modelos ER y UML y más adelante se enumerarán las metodologías y las estrategias a seguir en la fase de modelado conceptual.

1.1. El modelo ER

El modelo entidad-interrelación, o modelo ER, es un modelo conceptual de datos de alto nivel e independiente de la tecnología. Este modelo y sus variaciones constituyen los modelos más utilizados por el diseño conceptual de las aplicaciones de bases de datos. Esto se debe, principalmente, a su simplicidad y facilidad de uso. Los principales elementos que incluye el modelo son las entidades, los atributos y las relaciones entre entidades.

> El objetivo principal del **modelo ER** es permitir a los diseñadores reflejar en un esquema conceptual los requisitos del mundo real que sean de interés para el problema a tratar.

El origen del modelo ER se remonta a los trabajos hechos por Peter Chen[14] en 1976. Posteriormente, otros muchos auto-

14. El científico y profesor de informática en la Universidad Estatal de Luisiana. En 1976, desarrolló el modelo ER, hecho por el que es conocido.

res han propuesto variantes y ampliaciones para este modelo. Por lo tanto, en la bibliografía se pueden encontrar variaciones en el modelo ER que pueden diferir simplemente en la notación diagramática o en algunos conceptos en los que se basan para modelar los datos.

El modelo ER permite reflejar aspectos relacionados con la estructura de los datos y la integridad de estos, pero no pueden reflejarse aspectos de comportamiento, es decir, qué operaciones se ejecutan sobre los datos.

Para representar el modelo ER se han empleado tradicionalmente los diagramas entidad-interrelación, o diagramas ER. Estos definen una nomenclatura específica para representar los diferentes conceptos de entidades y relaciones que requiere el modelo y representarlos en diagramas ER. A pesar del amplio uso de estos diagramas, últimamente se tiende a emplear el lenguaje UML para representar esquemas conceptuales de bases de datos. No obstante, hay que tener en cuenta que los conceptos que utilizan dichos lenguajes para crear esquemas conceptuales de bases de datos son parecidos: ambos representan los conceptos relacionados con las entidades, los atributos y las relaciones, pero utilizando estructuras distintas.

1.2. El lenguaje UML

El lenguaje unificado de modelado[15] (UML) es un lenguaje de propósito general diseñado para modelar sistemas de software. El estándar fue creado y es mantenido por el Object

15. En inglés, *unified modeling language* (UML).

Management Group[16]. Se añadió por primera vez a la lista de tecnologías empleadas por el OMG en 1997 y desde entonces se ha convertido en el estándar de la industria para modelar sistemas de software.

> **UML** es un lenguaje gráfico diseñado para especificar, visualizar, modificar, construir y documentar un sistema. Permite una visualización estándar de diferentes artefactos, entre otros, actividades, actores, lógicas de negocio y esquemas de bases de datos.

El lenguaje UML lo desarrolló la compañía Rational Software Corporation durante la década de 1990. Después de un período en el que coexistieron diferentes tendencias en el modelado orientado a objetos, la compañía unificó los esfuerzos de tres pioneros en esta área: James Rumbaugh, Grady Booch e Ivar Jacobson. En 1996, se creó el consorcio UML Partners con el objetivo de completar la especificación de UML. En enero de 1997, se publicó la versión 1.0 de UML. Durante el mismo año, se publicó la versión 1.1, que aceptó y adoptó el consorcio OMG. Finalmente, la versión 2.0 se publicó en el 2005. En el momento de escribir este material, la última versión es la 2.4.1, publicada en agosto del 2011.

El lenguaje UML define nueve tipos de diagramas que permiten representar el modelo de un sistema desde diferentes perspectivas. Estos diagramas se clasifican como estructurales

16. El Object Management Group (OMG) es un consorcio dedicado a establecer y promover varias especificaciones de tecnologías orientadas a objetos, como UML, XMI o CORBA. Es una organización sin ánimo de lucro que promueve el uso de la tecnología orientada a objetos mediante guías y especificaciones para estas guías.

o de comportamiento. Los estructurales describen información estática, es decir, definen la información relacionada con un sistema de información y sus interrelaciones, pero no cómo esta información es tratada por el sistema de información. Por otro lado, los diagramas de comportamiento describen las diferentes operaciones que pueden ejecutarse sobre los datos de los diagramas estructurales y entre los diferentes componentes del sistema.

Para el diseño conceptual de bases de datos nos interesa especialmente el diagrama de clases, que permite representar información del dominio de discurso[17]. Aun así, a continuación se describen de forma breve cada uno de los diagramas UML, a pesar de que el alcance de estos cae fuera de los objetivos de este texto.

Diagramas estructurales: permiten representar la información relevante de un sistema de información y la organización de sus componentes.

a) Los **diagramas de clases** son diagramas estáticos que describen la estructura de un sistema a partir de las clases del sistema, los atributos de estas clases y las relaciones que se establecen entre ellas (conocidas como asociaciones en terminología UML). Estos diagramas son uno de los principales bloques en el desarrollo orientado a objetos, pero también han demostrado una capacidad excelente para modelar datos. Por este motivo, han sido cada vez más importantes en el diseño conceptual de bases de datos. Por ejemplo, si quisiéramos diseñar un pequeño programa para realizar la

17. El dominio del discurso, universo del discurso, o simplemente dominio, es la información relevante en un determinado contexto.

autenticación de usuarios de una empresa, el esquema relacionado del diagrama de clases contendría la clase "*Usuarios*" y los atributos "*login* y *password*" de dicha clase.

b) Los **diagramas de objetos** muestran las instancias (u objetos del mundo real) y las relaciones entre estas conforme a un diagrama de clases. Las instancias de un sistema se modifican a lo largo del tiempo, por lo tanto, los diagramas de objetos se pueden ver como una fotografía que muestra las diferentes instancias de un sistema y cómo están relacionadas en un instante de tiempo determinado. Por ejemplo, en un momento determinado la clase "Usuario" podría tener dos instancias, una para representar el usuario "José María" con *password* "caf4" y otra para representar la usuaria "Ana" con el password "c0n14ch4".

c) Los **diagramas de componentes** ilustran la organización y las dependencias entre los componentes del sistema. En entornos de bases de datos, se utilizan para modelar los espacios de tablas o las particiones. No se tratan estas estructuras en este libro porque forman parte del diseño físico de bases de datos.

d) Los **diagramas de implantación** representan la distribución de componentes del sistema y su relación con los componentes del hardware disponible.

Diagramas de comportamiento: permiten representar las funciones que realiza un sistema de información y cómo se realizan desde diferentes niveles de abstracción. Es decir, definen la funcionalidad de los sistemas de información.

a) Los **diagramas de casos de uso** se utilizan para modelar las interacciones funcionales entre los usuarios y el sistema.

b) Los **diagramas de secuencia** describen las interacciones entre distintos objetos en el transcurso del tiempo. Muestran el flujo temporal de mensajes entre varios objetos.

c) Los **diagramas de colaboración** representan las interacciones entre objetos como una serie de mensajes en secuencia. Estos diagramas centran la atención en la organización estructural de los objetos que envían y reciben mensajes.

d) Los **diagramas de estado** describen cómo cambia el estado de un objeto en respuesta a diferentes acontecimientos externos.

e) Los **diagramas de actividad** presentan una vista dinámica del sistema y modelan el flujo de control de actividad a actividad.

2. Metodologías y estrategias de diseño conceptual

La creación de un esquema conceptual de una base de datos permite a los diseñadores concentrarse en expresar el significado, las propiedades, las relaciones y las restricciones de los datos sin preocuparse de los detalles de implementación y con independencia de las particularidades de los diferentes sistemas gestores de bases de datos. Además, el esquema conceptual generado tiene las siguientes ventajas:

- Se convierte en un elemento de descripción estable del contenido de la base de datos.

- Al ser independiente de las tecnologías específicas de cada SGBD, permite más expresividad y es más generalizable.
- Se puede utilizar como vehículo de comunicación entre los usuarios, los diseñadores y los analistas de la base de datos.

Para la creación de un esquema conceptual se pueden utilizar diversas metodologías y estrategias de diseño. A continuación se describen algunas de ellas.

2.1. Metodologías de diseño

A partir de los requisitos recopilados, trabajados y formalizados en la primera fase del proceso de diseño de una base de datos se pueden establecer dos metodologías básicas para abordar el diseño conceptual de bases de datos:

1. **Metodología centralizada**: antes de empezar la creación del esquema conceptual se fusionan todos los requisitos de los distintos grupos de usuarios y aplicaciones recopilados en la primera fase. A partir de este único conjunto de requisitos se desarrolla un único esquema conceptual para toda la base de datos.
2. **Metodología de integración de vistas**: se construye un esquema para cada conjunto de requisitos y se validan los esquemas de manera independiente. Estos esquemas se conocen también como vistas. Posteriormente, se fusionan las diferentes vistas en un único esquema conceptual global para toda la base de datos.

La metodología de integración de vistas permite crear esquemas conceptuales más pequeños que son validados por los diferentes grupos de usuarios de la base de datos, pero añade la subfase de integración de vistas para unificar los diferentes esquemas. Además, esta subfase añade una complejidad notable y requiere una metodología y unas herramientas específicas para poder llevarse a cabo de manera correcta.

La elección de una metodología u otra depende en gran medida de la complejidad y la envergadura de la base de datos a desarrollar. La metodología de integración de vistas es aconsejable para el diseño de grandes bases de datos o cuando la base de datos que se va a crear contiene temáticas muy dispersas. Mientras que la metodología centralizada se aconseja en bases de datos pequeñas y poco complejas.

En este libro nos centraremos en la metodología centralizada.

2.2. Estrategias de diseño

A partir de un conjunto de requisitos, recopilados para un único usuario o para un conjunto de ellos, hay que diseñar un esquema conceptual que los satisfaga. Para llevar a cabo esta tarea, se pueden utilizar diferentes estrategias. La mayoría utilizan un método incremental, es decir, empiezan por modelar algunas estructuras principales derivadas de los requisitos y las van modificando y refinando en diferentes iteraciones del mismo proceso. Algunas de las estrategias más utilizadas para el diseño conceptual de bases de datos son las siguientes:

1. **Estrategia descendente**: se empieza el proceso con un esquema que contiene abstracciones de alto nivel y se van aplicando sucesivamente refinamientos sobre estas abstracciones.
2. **Estrategia ascendente**: se empieza el proceso con un esquema que contiene los detalles de las abstracciones y se van combinando de manera sucesiva formando conjuntos de más entidad o conceptos complejos.
3. **Estrategia de dentro hacia fuera**: es un caso concreto de la estrategia ascendente. Esta centra la atención en un conjunto principal de conceptos que son muy evidentes y, poco a poco, incluye en el esquema nuevos conceptos relacionados directamente con los existentes.
4. **Estrategia mixta**: divide los requisitos según una estrategia descendente y entonces construye cada una de las divisiones siguiendo una estrategia ascendente. Finalmente, se combinan las partes para generar el esquema completo.

Capítulo III

Elementos básicos de modelado

Tal y como se ha descrito, un modelo conceptual representa una abstracción de la información de dominio que es relevante para un sistema de información.

El esquema conceptual utiliza los siguientes elementos estructurales para representar la información:

- **Tipos de entidad**: permiten modelar conceptos, es decir, agrupaciones de objetos con características comunes. Un ejemplo de tipo de entidad sería *Población*, que describiría las características más relevantes de las poblaciones en nuestra base de datos.
- **Atributos**: permiten definir las características de los tipos de entidad. Un atributo de *Población* sería, por ejemplo, el *código postal*.
- **Tipos de relación**: permiten modelar las relaciones que pueden existir entre los tipos de entidad. Un ejemplo sería '*tiene domicilio en*' que permite definir que los clientes de un negocio residen en una población.
- **Restricciones de integridad**: son condiciones que se deben satisfacer para cualquier estado de la base de datos. Su objetivo es asegurar que el esquema conceptual está en un estado consistente. Es decir, si hay

algún objeto que no cumple con una restricción de integridad, entonces puede decirse que dicho objeto no es válido en esta base de datos. Un ejemplo de restricción de integridad sería *"dos poblaciones no pueden tener el mismo código postal"*.

A continuación vamos a estudiar con más detalle cada uno de estos elementos estructurales y a poner ejemplos de cada uno de ellos utilizando el lenguaje UML.

1. Tipos de entidad

Entendemos por **entidad** un objeto del mundo real, que tiene identidad propia y que es distinguible del resto de los objetos.

Las entidades pueden ser objetos con existencia física (por ejemplo, *este libro en formato físico* o *Leonard Nimoy*, que es el actor que representa el papel de Spock en *Star Trek*) u objetos con existencia conceptual (por ejemplo, *este libro en formato eBook* o el *comandante Spock*).

Otros ejemplos de entidades tangibles son una manzana, un coche o una persona. Otros ejemplos, que no son tangibles pero que tienen identidad y son distinguibles del resto, son un pedido a un proveedor, una petición de préstamo en una biblioteca o una incidencia municipal.

> Entendemos por **tipo de entidad** la abstracción que permite definir un conjunto de entidades con las mismas propiedades, comportamiento común, semántica común e idéntica relación con los demás objetos. Como, por ejemplo, el tipo de entidad que describe el conjunto de lectores de este libro.

Filosóficamente hablando, es casi imposible encontrar dos entidades físicas en el mundo que cumplan la definición anterior, ya que siempre se podrán encontrar pequeñas diferencias entre ellas. Por eso es necesario un proceso de abstracción, que permite pasar de un conjunto de entidades "parecidas" a un tipo de entidad. Este proceso consiste en eliminar las diferencias o distinciones entre las entidades que no son relevantes para el problema que tratar con el objetivo de poder observar aspectos comunes a todas ellas. Por ejemplo, todos los lectores de este libro son distintos entre sí, pero mediante un proceso de abstracción podemos considerar un tipo de entidad llamado "*Lectores de este libro*" que describe todas aquellas personas que lo han leído. En este caso, las características comunes serían que dichos lectores son personas y que han leído este libro.

Por otro lado, la decisión de qué es una entidad y qué es un tipo de entidad no es trivial y depende del problema a tratar y, por lo tanto, qué información hay que representar en la base de datos. Por ejemplo, en una base de datos de una tienda de vinos, cada botella de vino será una entidad porque nos interesa controlar el *stock* de botellas que tenemos. No obstante, en una base de datos que pretenda almacenar los tipos de vino que existen, con qué uva se hace cada uno, de dónde provienen y con qué tipo de alimentos combinan mejor, entonces las entidades no serían las botellas de vino

concretas, sino las distintas marcas de vino o bien los tipos de vino existentes. El motivo es obvio, si queremos almacenar que los rioja utilizan uva de tipo tempranillo, no es necesario almacenar las distintas botellas físicas de tipo rioja que existen.

Existe una relación de pertenencia entre el tipo de entidad y las entidades que describe. Conceptualmente, podríamos ver el tipo de entidad como un conjunto, y las diferentes entidades como los miembros de ese conjunto. Esa relación de pertenencia se denomina instanciación y se representa normalmente mediante relaciones de instanciación que relacionan cada instancia con el tipo de entidad a la que pertenece. No entraremos en más detalle en este tipo de relación ya que no se hace un uso exhaustivo de ella en el diseño de bases de datos, tan solo se utiliza a veces para ejemplificar un esquema conceptual mediante diversas instanciaciones del mismo.

El tipo de entidad *coche* se define como el concepto de *vehículo* que describe las partes comunes de diferentes coches con independencia de la marca, el color, el modelo y otras características concretas. Un coche concreto, por ejemplo *mi coche*, es una instancia u ocurrencia del tipo de entidad *coche*.

Utilizaremos la nomenclatura siguiente para diferenciar estos dos conceptos:

- El concepto que hemos definido como *entidad* también puede ser referenciado por los términos *objeto*, *instancia* u *ocurrencia*.
- El concepto que hemos definido como *tipo de entidad* también puede ser referenciado por los términos *clase*, *entidad tipo* o *tipo*.

En la bibliografía se pueden encontrar autores que utilizan las diferentes terminologías y, por lo tanto, hay que tenerlas todas presentes para identificar de manera rápida y unívoca a qué concepto se hace referencia cuando se utiliza cualquiera de estos términos.

En la representación mediante diagramas UML, los tipos de entidad se denominan *clases* y las entidades, *objetos*.

2. Atributos

> Un **atributo** es una propiedad que tienen todas las entidades de un mismo tipo de entidad y que permite representar alguna de sus características.

Los atributos son muy importantes en el modelado conceptual porque nos permiten definir las características que tienen los elementos que estamos modelando.

A continuación utilizaremos los diagramas de clases de UML para representar gráficamente los tipos de entidad y sus atributos[18].

Por ejemplo, supongamos que tenemos un tipo de entidad denominado '*Coche*' con un conjunto de atributos; por ejem-

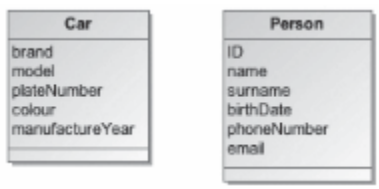
18. Atributos y relaciones binarias. En teoría, un atributo no es más que un tipo de relación binaria en la que uno de los participantes es un tipo de dato (se discutirá más a fondo en el apartado siguiente). No obstante, para simplificar el modelo y facilitar su creación y su legibilidad, se tratan de manera diferente y más compacta.

plo: marca, modelo, matrícula, color y año de fabricación. Una entidad de *coche* tiene valores específicos para cada uno de estos atributos. Por ejemplo, un coche puede ser de la marca *Fiat*, modelo *Punto*, matrícula *3621-GHV*, color *negro* y fabricado en el *2010*.

De manera similar, el tipo de entidad *'Persona'* tiene un conjunto de atributos; por ejemplo: *identificación* (en el caso de España es el documento nacional de identidad o DNI), *nombre y apellidos*, *fecha de nacimiento*, *número de teléfono* y *correo electrónico*. Una entidad de '*Persona*' puede ser, por ejemplo, *Fred Smith* con DNI *33551058D*, nacido el *1 de marzo de 1968*, con número de teléfono *85542154* y correo electrónico *fredsmith@mydom.com*.

En los diagramas de clases del modelo UML representaremos los tipos de entidad (denominados clases en los diagramas UML) mediante un rectángulo con tres secciones. En la primera sección indicaremos el nombre del tipo de entidad. Los atributos se representan como una lista en la segunda sección, y la tercera sección permite representar ciertas operaciones o restricciones sobre los datos, que no utilizaremos en el diseño conceptual de bases de datos. La figura 2 muestra un ejemplo de representación de los tipos de entidad *'Coche'* (*Car*) y *'Persona' (Person)* en UML.

Figura 2. Ejemplos de los tipos de entidad *'Coche' (Car)* y *'Persona' (Person)*



La tabla 1 muestra un posible ejemplo de un conjunto de entidades de persona.

Tabla 1. Ejemplo de representación de entidades del tipo de entidad *'Persona' (Person)*

ID	Name	Surname	Birth date	Phone number	Email
33941857B	John	Smith	12/10/1987	938541524	johnsmith@example.com
77854623Q	Mary	Jane	24/02/1979	658421569	maryjane@domain.com
96725466Z	Fred	Sánchez	17/10/1992	658417841	freddy@my.me

2.1. Representación de los atributos

Los diagramas UML presentan una nomenclatura concreta para especificar las propiedades de cada uno de los atributos[19]:

[visibilidad] nombre [etiquetas] [: tipo] [multiplicidad] [= valor inicial]

Dónde:

- Visibilidad: define la visibilidad del atributo (*public*, *protected*, *package* o *private*). Normalmente este concepto no se aplica al diseño conceptual de bases de datos.

19. Los corchetes [ ] denotan opcionalidad. Por tanto, el uso de los elementos entre corchetes no es obligatorio en la definición de atributos.

- Nombre: denominación del atributo, el único valor obligatorio.
- Etiquetas: etiquetas opcionales para indicar ciertos conceptos relacionados con el atributo.
- Tipo: tipo o dominio de los datos, por ejemplo *numérico* o *String*.
- Multiplicidad: número de ocurrencias del atributo.
- Valor inicial: valor que toma el atributo por defecto.

En apartados posteriores veremos cómo utilizar la representación de los atributos para definir atributos multivalor, de enumeración, atributos derivados, atributos clave, etc.

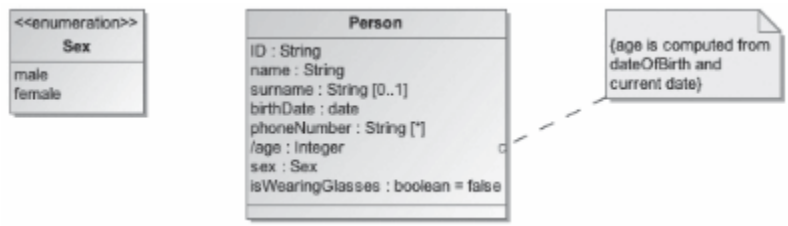
En UML se pueden asociar notas para indicar restricciones de integridad o cualquier casuística que el diseñador considere oportuno remarcar. Podemos ver un ejemplo en la figura 3, donde una nota indica que el atributo *age* se calcula a partir de la diferencia entre la fecha actual y el valor del atributo *dateOfBirth*.

La figura 3 muestra un tipo de entidad que tiene un atributo opcional *(surname)*, un atributo derivado[20] *(age)*, un atributo de tipo enumeración[21] *(sex)*, un atributo con valor inicial especificado *(isWearingGlasses)* y un atributo multivalor *(phoneNumber)*. Una nota asociada al atributo *age* indica cómo calcular los valores de este atributo derivado.

---

20. Un atributo derivado es un atributo cuyo valor se calcula automáticamente. Los atributos derivados se estudian con más detalle en apartados posteriores.
21. Un atributo de tipo enumeración tiene restringidos sus posibles valores a un conjunto cerrado de valores predefinido. Por ejemplo, se pueden definir los días de la semana como un atributo de tipo enumeración, donde el conjunto de posibles valores es lunes, martes, miércoles, jueves, viernes, sábado y domingo.

Figura 3. Ejemplo del tipo de entidad *'Persona' (Person)*



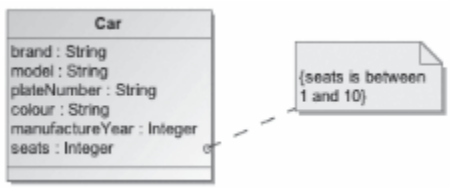
2.2. Dominio de los atributos

El conjunto de posibles valores que puede tomar un atributo se denomina dominio del atributo o tipo de datos (datatype en inglés). Generalmente, se especifican a partir de los tipos de datos básicos disponibles en la mayoría de los lenguajes de programación, como entero, real, cadena de texto, booleano o tipo enumerado. No obstante, algunos lenguajes de modelado conceptual permiten definir tipos de datos propios.

La figura 4 muestra el tipo de entidad *'Coche'* (*Car*), en el que se especifica el dominio para cada uno de sus atributos. En este ejemplo hemos definido un nuevo atributo para indicar el número de plazas de un coche *(seats)*, que corresponde al dominio de los enteros y que está limitado por un valor mínimo de 1 plaza y un valor máximo de 10 plazas (para este ejemplo se considera que no hay coches que tengan un número de plazas superior a 10). En UML esta condición se puede crear utilizando notas textuales o recurriendo a lenguajes que permitan definir restricciones

de integridad sobre los modelos UML, como por ejemplo OCL[22].

Figura 4. Ejemplo de los tipos de entidad *'Coche' (Car)* con el dominio de los atributos



2.3. Atributos compuestos y atómicos

> Un **atributo compuesto** es aquel que se puede dividir en atributos más básicos con significado independiente. Un **atributo atómico** o **simple** es aquel que no es divisible sin perder el significado.

Un ejemplo de atributo compuesto es el concepto de 'dirección postal': el atributo de *dirección postal* con valor "Avenida Santa Cristina, 75, 08280 Barcelona" se puede descomponer en los atributos *calle*, *número*, *código postal* y *ciudad*. De este modo, cada nuevo atributo mantiene significado independiente y el conjunto de todos ellos identifica una

22. OCL es un lenguaje declarativo que permite definir reglas que se aplican a modelos UML. Entre otras, se pueden describir restricciones sobre el dominio de los atributos. En este módulo no utilizaremos este lenguaje, puesto que supone una dificultad añadida, y utilizaremos las notas textuales para indicar restricciones en el dominio y consideraciones similares.

dirección postal. Un ejemplo de atributo simple es el nombre de una población. Aunque este atributo pueda parecer compuesto, por ejemplo *"Arenys de Mar"*, no se puede dividir sin perder el significado original.

2.4. Atributos monovalor y multivalor

Un **atributo monovalor** es aquel que solo puede tener un valor para cada entidad. En contraposición a este concepto, un **atributo multivalor** es aquel que puede tomar más de un valor para la misma entidad.

Un ejemplo de atributo monovalor es el atributo '*fecha de nacimiento*' de una persona, puesto que una persona solo puede haber nacido una única vez. Un ejemplo de atributo multivalor es el atributo 'teléfono' de una persona, puesto que se puede dar el caso de que una persona tenga cero, uno o más números de teléfono (por ejemplo, un teléfono fijo y un teléfono móvil).

Para indicar la cardinalidad[23] de un atributo en UML se utiliza:

- Un número para indicar el número exacto de valores que debe tener el atributo.
- El intervalo mín..máx para indicar el número mínimo y máximo de valores que puede tener el atributo. Por ejemplo, para indicar que el atributo '*teléfono*' *(phone-*

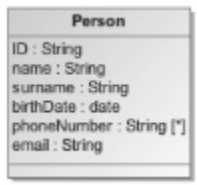
23. La cardinalidad de un atributo define el número de posibles valores que puede tener el atributo para cada entidad.

Number) del tipo de entidad '*Persona*' *(Person)* puede tener entre 0 y 5 valores hay que indicar "*phoneNumber: String[0..5]*" en la definición del atributo.

- El símbolo * para indicar que el atributo puede tener un número indefinido valores (cero o más).

Por defecto, los atributos son de tipo monovalor en UML. Por tanto, los atributos *ID*, *name*, *surname*, *birthdate* y *email* de la clase *Person* (ver figura 5) son monovalor porque no se indica lo contrario. El atributo '*teléfono*' es multivalor y puede tener un número indefinido de valores.

Figura 5. Ejemplo del tipo de entidad 'Persona' (Person) con un atributo multivalor

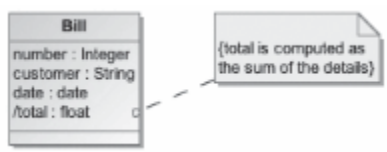


2.5. Atributos derivados

Un **atributo derivado** es aquel cuyo valor se puede calcular a partir de los valores de otros atributos, y lleva asociado un procedimiento de cálculo que indica cómo calcular su valor a partir del valor de otros atributos. Un atributo no derivado, denominado básico, es aquel al que hay que asignar un valor explícitamente.

Supongamos la clase *'Factura' (Bill)* con los atributos *'número de factura' (number), 'cliente' (customer), 'fecha de la factura' (date)* e *'importe total de la factura' (total)*. Para obtener la información del importe total de la factura, podemos crear el atributo derivado *'importe total' (total),* que se calcula a partir de la suma de cada una de las líneas de los detalles de la factura, tal como se muestra en la figura 6.

Figura 6. Ejemplo de atributo derivado



2.6. Atributos opcionales

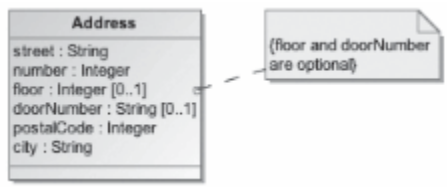
El **valor NULL** (o nulo) es un valor especial diseñado para indicar que el valor de un atributo es desconocido o no tiene sentido en una entidad concreta. Un atributo que acepte el valor nulo se considera un atributo opcional.

En caso de que se especifique que el atributo no acepta el valor nulo, quiere decir que el atributo es obligatorio y, por tanto, todas las entidades deben tener un valor en dicho atributo.

La figura 7 muestra el tipo de entidad *'Dirección' (Address)*, que permite definir la dirección física de una casa, un piso o un local. Hay que señalar que en las casas unifamiliares no tiene sentido el atributo de *'número de piso y puerta'*. Los

atributos que indican el *'número de piso' (floor)* y *'puerta de escalera' (doorNumber)* especifican una multiplicidad de 0 o 1, es decir, son opcionales y pueden tener un valor nulo.

Figura 7. Ejemplo de atributo con valor NULL



2.7. Atributos de clave

> Una **clave candidata** es un conjunto de atributos mínimo que permite identificar de manera única todas las entidades de un tipo de entidad.

Un tipo de entidad puede tener una o más claves candidatas. Pero es necesario que tenga al menos una clave candidata que permita identificar de manera unívoca todas las entidades[24].

24. Esta restricción solo es aplicable en función del tipo de base de datos con el que trabajaremos. Si trabajamos con bases de datos relacionales, es necesario que haya un atributo de clave para cada clase, pero si trabajamos con bases de datos orientadas a objeto, por ejemplo, esto no es necesario, ya que cada clase tiene un identificador interno (llamado *oid*) que permite identificar todas sus instancias. Aunque no sea un elemento necesario en cualquier base de datos, consideramos una buena práctica definir una clave principal para cada clase con el fin de facilitar el diseño y la posterior consulta de los datos de la base de datos.

Una clave candidata puede estar formada por uno o más atributos. En el caso de estar formada por varios atributos, recibe el nombre de clave candidata compuesta. En este caso, la combinación de los diferentes valores de los atributos debe ser única para identificar de manera unívoca a todas las entidades de un mismo tipo de entidad. Una clave candidata compuesta debe ser mínima, es decir, debe incluir el número mínimo de atributos que permitan identificar de manera unívoca las entidades. La definición de clave candidata se puede ver como una restricción sobre los datos, puesto que prohíbe que dos entidades tengan el mismo valor en los atributos que forman la clave candidata.

El diseñador de la base de datos elige una de las claves candidatas para que sea la clave primaria. La **clave primaria** es la clave elegida para identificar las entidades de manera unívoca. Como clave candidata, no puede contener valores duplicados pero además, de manera implícita, tampoco puede contener valores nulos en ninguna entidad.

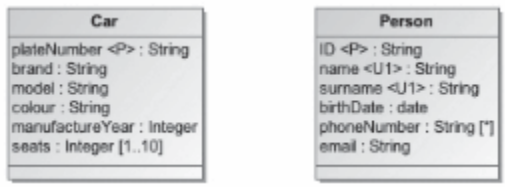
En UML no hay ninguna notación para indicar las claves candidatas ni la clave primaria. En este texto tomamos un convenio para indicar las claves candidatas y las claves primarias. Las claves candidatas se marcarán con la etiqueta <*Un*>, donde n (n > 0) agrupa cada uno de los conjuntos de claves candidatas. Por lo tanto, si dos o más atributos tienen la misma etiqueta, por ejemplo <*U1*>, entendemos que se trata de una clave candidata compuesta por todos los atributos con esta etiqueta. La clave primaria se marca con la etiqueta <*P*> para distinguirla del resto de claves.

Otros autores prefieren indicar las claves con la etiqueta {*Un*} y {*P*} o utilizar restricciones textuales como método de notación. Aunque el significado es el mismo para las diferentes notaciones, en este texto utilizaremos la notación mencionada inicialmente.

Notad que decir que un atributo es una clave candidata es lo mismo que decir que el atributo en cuestión es único. O sea, que no puede haber dos entidades que tengan el mismo valor para el atributo.

La figura 8 muestra ejemplos de los tipos de entidad *Car* y *Person* con las marcas de la clave primaria y de las claves candidatas. El tipo de entidad *'Persona' (Person)* tiene dos claves candidatas: por un lado, la clave formada por el atributo *'ID'* y por el otro, la clave formada por los atributos *'nombre' (name)* y *'apellidos' (surname)*. Para este ejemplo, suponemos que no pueden existir dos personas con el mismo nombre y apellidos. Aun así, elegimos como clave primaria el atributo 'ID'. El tipo de entidad *'Coche' (Car)* solo tiene una clave candidata, formada por el atributo *'matrícula' (plateNumber)*. Por tanto, esta es elegida como clave primaria.

Figura 8. Ejemplos de los tipos de entidad *'Coche' (Car)* y *'Persona' (Person)* con la clave primaria y las claves candidatas marcadas



3. Tipos de relación

Definimos las relaciones como asociaciones entre entidades. Por ejemplo, si *Leonard Nimoy* es una entidad y *Comandante Spock* es otra entidad, entonces *"representó el papel de"* es una relación que nos permite decir que *Leonard Nimoy* representó el papel del *Comandante Spock*. Al igual que en las entidades, la abstracción de un conjunto de relaciones con características comunes puede dar lugar a un tipo de relación.

> Un **tipo de relación** es una asociación entre dos o más tipos de entidad, llamados tipos de entidad participantes. Los tipos de relación son abstracciones que indican que las instancias de los tipos de entidad participantes pueden relacionarse entre sí teniendo en cuenta la semántica y las restricciones definidas en el tipo de relación.

Siguiendo con el ejemplo anterior, si *Leonard Nimoy* es instancia de *Person* y *Comandante Spock* es instancia de la clase *Fiction Character*, entonces podemos definir un tipo de relación llamado *"plays the role of"* entre *Person* y *Fiction Character* y que signifique que las personas pueden representar caracteres de ficción.

Los tipos de relación pueden relacionar cualquier número de tipos de entidad, pero normalmente, por simplicidad, se acostumbra a utilizar tipos de relación binarios, es decir, tipos de relación que solo relacionan dos tipos de entidad. Un ejemplo de tipo de relación ternario (con tres tipos de entidad participantes) podría ser *"X interpretó el Papel de Y en la*

Película Z", donde X, Y y Z serían los tipos de entidad *Person* (X), *Fiction Character* (Y) y *Film* (Z).

Los tipos de relación indican las relaciones en las que pueden participar las entidades de un tipo de entidad, y en qué condiciones.

A partir de ahora, utilizaremos la nomenclatura siguiente para diferenciar estos dos conceptos:

- El concepto que acabamos de definir como tipo de relación también puede ser referenciado por los términos interrelación, relación o asociación[25].
- El concepto que acabamos de definir como relación también puede ser referenciado por los términos instancia u ocurrencia de la relación, interrelación o asociación.

En la figura 9 podemos ver un ejemplo de cómo definir un tipo de relación en UML. Entre los tipos de entidad *'Coche' (Car)* y *'Persona' (Person)* podemos establecer un tipo de relación de propiedad que indique que una persona posee un coche. Para representar esta información, debemos crear en el diagrama conceptual un nuevo tipo de relación denominado *Has*, para indicar que una persona tiene un coche.

25. En la representación en diagramas UML, los tipos de relación se denominan asociaciones y las relaciones, instancias de las asociaciones.

Figura 9. Ejemplo de tipo de relación Has entre los tipos de entidad 'Persona' (Person) y 'Coche' (Car)

Tal como se puede ver en la figura 9, en un tipo de relación pueden aparecer diferentes etiquetas, todas con el fin de facilitar la comprensión de la relación entre las entidades:

- Los tipos de relación casi siempre tienen una etiqueta de tipo de relación que indica el significado de la asociación entre los dos tipos de entidad. Esta etiqueta indica la acción entre los dos tipos de entidad, y se suele expresar mediante un verbo. Para especificar mejor su significado, de manera opcional se puede indicar la dirección en que debe leerse la relación. Así, en la figura 9 podemos leer que «una persona tiene un coche». Aunque no es muy habitual, cuando esta relación es lo bastante evidente se puede considerar que no es necesario indicarlo explícitamente y se puede obviar la etiqueta o la dirección.
- También se puede dar la situación en que sea necesario definir el papel que tienen las entidades en el tipo de relación. En estos casos se puede definir una etiqueta en cada extremo del tipo de relación, que indica el papel que desempeña cada una de las entidades en la relación y que ayuda a explicar el significado de esta. Estas etiquetas se denominan etiquetas

de rol o nombres de rol. Para definir los roles se suelen utilizar nombres. En la figura 9 podemos ver los nombres de roles para los participantes del tipo de relación *"has"*, que serían *"propietario" (owner)* y *"coche de propiedad" (owned car)*. Utilizando los nombres de rol podemos leer los participantes del tipo de relación como *"la persona que participa en la relación es propietaria"* y *"el coche que participa en la relación es propiedad de alguien"*.

En UML es necesario que todo tipo de relación tenga definidos los roles de los participantes o el nombre del tipo de relación.

Se define el **grado de un tipo de relación** como el número de tipos de entidad que participan en la asociación.

Los tipos de relación se clasifican, según su grado, en:

- **Binarias**: asociaciones en las que intervienen dos tipos de entidad.
- **Ternarias**: asociaciones en las que intervienen tres tipos de entidad.
- **N-arias**: asociaciones en las que intervienen *n* tipos de entidad. A pesar de que los tipos de relación binarias y ternarias son un caso concreto de los tipos de relación *n*-arias, las relaciones *n*-arias se usan normalmente para denotar relaciones con un grado superior a tres. En la práctica es difícil encontrar modelos conceptuales con relaciones cuaternarias o de superior grado.

3.1. Tipos de relación *vs.* atributos

En este punto seguro que algunos lectores se han preguntado: ¿qué diferencia hay entre un tipo de relación y un atributo? ¿no son parecidos? La respuesta es sí. No solo son parecidos, sino que uno es un caso particular del otro.

Un atributo podría verse como un caso particular de tipo de relación binario. De hecho, un atributo es un tipo de relación binario que tiene un tipo de datos (o *datatype*) como participante. Sorprendentemente, hoy en día aún hay muchas definiciones de «tipo de datos» y no parece que haya un consenso claro sobre su semántica. En este libro, y para simplificar las cosas, definiremos un tipo de datos como un tipo de entidad cuyas instancias identifican unívocamente al objeto que representan en el mundo real a partir de sus valores. Por ejemplo, los enteros son un tipo de datos porque dos instancias distintas de la clase entero con el mismo valor son el mismo número entero: 3 igual a 3. Lo mismo pasa con un String o con una fecha, ya que la cadena "Hola" y "Hola" son la misma y las fechas "3 de julio de 1974" y "3 de julio de 1974" también son la misma fecha. No obstante, el tipo de entidad *Libro,* por ejemplo, no es un tipo de datos. El motivo es que podríamos tener dos instancias de libro con los mismos datos (ISBN, idioma, título, autor…) pero aun así ser dos ejemplares físicos distintos. Básicamente, y para simplificar, podemos considerar como tipos de datos a los tipos básicos disponibles en la mayoría de los lenguajes de programación: enteros, reales, carácter, String, etc.

Aunque un atributo se pueda representar como una relación binaria entre el tipo de entidad que lo contiene y el tipo de datos asociado, normalmente se representa mediante atri-

butos porque así se mejora su compresión y permite agrupar mejor la información relacionada.

En algunos casos los tipos de relación, en primera instancia, pueden modelarse como atributos. Un análisis posterior con más detalle ayuda a ver que lo que se había modelado como un atributo en un primer momento es realmente un tipo de relación que debe relacionar la clase actual con un tipo de entidad nuevo, o ya existente.

Por ejemplo, supongamos que queremos modelar el tipo de entidad empleado *(Employee)* para almacenar los datos de los empleados de una empresa. Entre otros atributos, nos interesa indicar a qué departamento de la empresa pertenece cada empleado. Una primera aproximación sugiere un diseño en el que se incluye el atributo *'departamento'* en el tipo de entidad *'Empleado'*, tal como muestra la figura 10. Pero refinando este modelo, es posible que nos interese tener categorizados los diferentes departamentos, e incluso podríamos querer almacenar cierta información relativa al departamento. En este nuevo diseño se crea un nuevo tipo de entidad para los departamentos y se establece un tipo de relación entre los empleados y los departamentos (tipos de entidad *Department*) que indica a qué departamento pertenece cada empleado. La figura 11 muestra el nuevo diseño.

Figura 10. Ejemplo de diseño del departamento como un atributo del tipo de entidad *'Empleado' (Employee)*

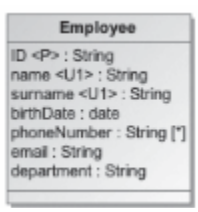
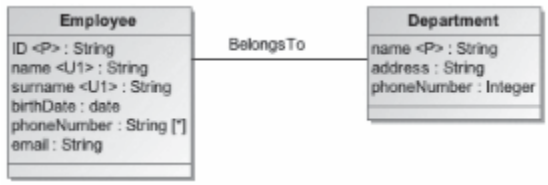


Figura 11. Ejemplo de diseño del departamento como un nuevo tipo de entidad *'Departamento' (Department)* relacionado con el tipo de entidad *'Empleado' (Employee)*



3.2. Tipos de relación binarias

Se define un **tipo de relación binaria** como la asociación entre dos tipos de entidad.

La figura 12 muestra el tipo de relación *'Matrícula' (Enrollment)* entre los tipos de entidad *'Estudiante' (Student)* y *'Asignatura' (Subject)*. Esta relación indica en qué asignaturas se ha matriculado cada uno de los estudiantes[26].

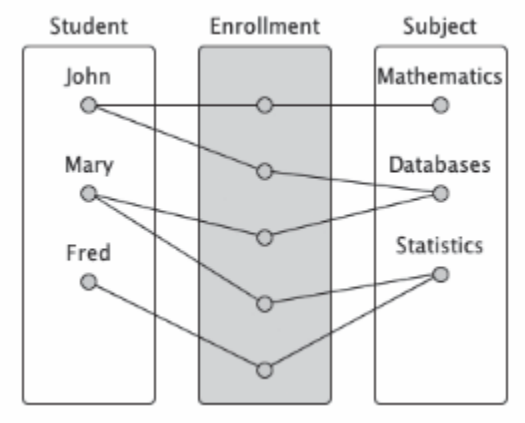
Figura 12. Ejemplo del tipo de relación binaria *'Matrícula' (Enrollment)* entre los tipos de entidad *'Estudiante' (Student)* y *'Asignatura' (Subject)*

Siguiendo con el ejemplo de la figura 12, la figura 13 muestra la misma relación, pero a escala de entidades. Es decir,

26. En algunos diagramas de ejemplo de los tipos de relación obviaremos los atributos y las características que no sean necesarias para explicar el concepto que nos interese, con objeto de simplificar los diagramas y centrarnos en los tipos de relación.

este ejemplo muestra una posible asociación entre los datos de los dos tipos de entidad mediante la relación *Enrollment*. Se puede ver, por ejemplo, que el estudiante *John* está matriculado en las asignaturas *Mathematics* y *Databases*, pero que el estudiante *Fred* solo se ha matriculado en *Statistics*.

Figura 13. Ejemplo en el nivel de entidades de la relación *'Matrícula' (Enrollment)*



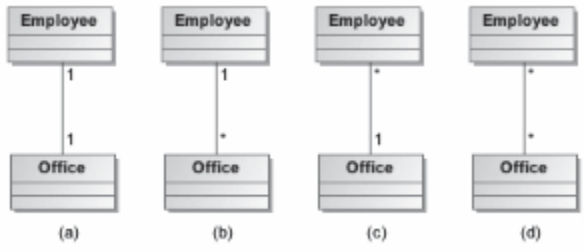
3.2.1. Conectividad

La **conectividad de un tipo de relación** expresa el tipo de correspondencia que hay entre los tipos de entidad que participan en un tipo de relación. En el caso de los tipos de relación binarios, expresa el número de ocurrencias de un tipo de entidad con el que se puede asociar una ocurrencia del otro tipo de entidad según el tipo de relación.

Un tipo de relación binaria puede tener tres tipos de conectividad:

1. **Conectividad uno a uno (1:1):** indica que cada entidad de un tipo de entidad se puede relacionar solo con una de las entidades del otro tipo. En la figura 14a se puede ver un ejemplo de conectividad 1:1, donde, según el modelo, cada empleado tiene un despacho y en cada despacho hay un solo empleado. La conectividad 1:1 se denota poniendo un 1 en cada parte del tipo de relación.
2. **Conectividad uno a muchos (1:N o 1..*):** indica que cada entidad de un tipo de entidad se puede relacionar con varias entidades del otro tipo, pero las entidades de este segundo tipo solo se pueden relacionar con una única entidad del primer tipo. En la figura 14b se puede ver un ejemplo de conectividad 1:N, donde la N está en la parte del tipo de entidad *'Despacho'*. Según el modelo, cada empleado tiene varios despachos, pero en cada despacho hay un solo empleado. En la figura 14c se puede ver la relación inversa. En este caso, la N está en la parte del tipo de entidad *'Empleado'*. Según este modelo, cada empleado tiene un único despacho, pero un mismo despacho lo pueden compartir varios empleados. La conectividad 1:N se denota poniendo un 1 en una parte del tipo de relación y una N o * en la otra parte.
3. **Conectividad muchos a muchos (M:N o *..*):** indica que cada entidad de un tipo de entidad se puede relacionar con varias entidades del tipo de entidad relacionado. En la figura 14d se puede ver un ejemplo de conectividad M:N, donde, según el modelo, cada empleado tiene varios despachos y en cada despacho hay varios empleados. La conectividad M:N se denota poniendo una M o * en una parte del tipo de relación y una N o * en la otra parte.

Figura 14. Ejemplos de conectividad en los tipos de relaciones binarias entre los tipos de entidad *'Empleado' (Employee)* y *'Despacho' (Office)*



Estas restricciones, denominadas también restricciones de cardinalidad, se determinan en función del problema a resolver. Por tanto, la conectividad de las relaciones dependerá en gran medida de la información que se quiera representar, es decir, de los requisitos del sistema de información recogidos.

Cuando se habla de «conectividad a muchos» se puede indicar de diferentes maneras. Según el matiz del problema, usaremos:

- Un número para indicar el número exacto de entidades que pueden intervenir en la relación.
- El intervalo *mín..máx* para indicar el número mínimo y máximo de entidades que pueden intervenir en la relación.
- La etiqueta *N* o * para indicar que un número indefinido (cero o más) de entidades pueden participar en la relación.

La figura 15 muestra el tipo de relación *'Matrícula' (Enrollment)* entre los tipos de entidad *'Estudiante' (Student)* y *'Asignatura' (Subject)*, donde se indica que un estudiante tiene que estar matriculado en un mínimo de dos asignaturas y en

un máximo de cinco. La relación no pone límite al número de estudiantes que puede tener cada asignatura, por tanto podríamos tener asignaturas sin estudiantes, otras con 10 estudiantes y otras con 100.000 estudiantes.

Figura 15. Ejemplo del tipo de relación *'Matrícula' (Enrollment*) donde se indica el número mínimo y máximo de instancias que pueden intervenir en la relación

3.2.2. Participación total o parcial en un tipo de relación

En algunos casos, las entidades de un tipo de entidad tienen que estar relacionadas con entidades de otro tipo de entidad de forma obligatoria. En estos casos, se dice que el primer tipo de entidad es un tipo de entidad obligatorio[27] en la relación. En caso contrario, se dice que es un tipo de entidad opcional[28] en la relación.

En los diagramas UML se utiliza la cardinalidad del tipo de relación para expresar la obligatoriedad de los tipos de entidad que participan en la asociación. Por tanto, para decir que el participante de un tipo de relación es obligatorio debemos colocar un número *N* en su cardinalidad mínima, donde

27. Para designar este concepto también se puede decir que el tipo de entidad tiene una participación total en la asociación o que presenta una dependencia de existencia respecto a la asociación.
28. Para designar este concepto también se puede decir que el tipo de entidad tiene una participación parcial en la asociación.

N>0. Por ejemplo, 1..1[29], 2..3, 1..*, etc. Colocar un 0 en la cardinalidad mínima de un tipo de entidad participante en una asociación indicará que sus entidades no tienen la obligación de participar en la relación. Ejemplos de cardinalidades que denotan optatividad son 0..1, 0..*[30], etc.

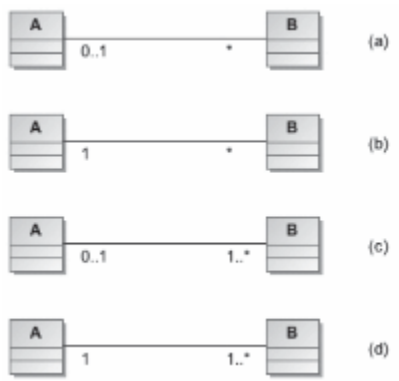
Debido a la manera en que se representan los tipos de relación en UML, los tipos de entidad obligatorios se definen colocando una cardinalidad mínima diferente a cero en el extremo opuesto de la asociación al de la clase obligatoria. La figura 16 muestra un modelo en el que aparece un tipo de relación binaria que expresa la relación de *'Dirige un departamento' (Heads)* entre los tipos de entidad *'Profesor' (Professor)* y *'Departamento' (Department)*. Podemos ver que el tipo de entidad *'Departamento'* es obligatorio en la relación ya que, según el diagrama, todo departamento deberá tener un profesor que lo dirija. Dicha restricción hará que todos los departamentos participen en la relación. Por tanto, podemos decir que la clase *'Departamento' es* obligatoria en la relación. No obstante, la clase *'Profesor'* es opcional en la relación ya que hay profesores que no dirigen ningún departamento y, por tanto, no todos los profesores participan en la relación.

Figura 16. Ejemplo de dependencia de existencia en los tipos de relación binarias

29. La cardinalidad 1 es una manera abreviada de referirse a la cardinalidad 1..1.
30. La cardinalidad * es una manera abreviada de referirse a la cardinalidad 0..*.

En una relación binaria con participantes A y B, tenemos cuatro posibilidades de expresar la obligatoriedad entre las clases participantes. En la figura 17, vemos en 17a el caso en el que los dos tipos de entidad son opcionales, en 17b el caso en el que el tipo de entidad A es obligatorio, en 17c el caso en el que el tipo de entidad B es obligatorio y en 17d el caso en el que los dos tipos de entidad son obligatorios.

Figura 17. Ejemplos de obligatoriedad en la dependencia de existencia



Igual que las restricciones de conectividad, estas restricciones son inherentes a los problemas que se quieren resolver y a partir de la compilación de requisitos de los usuarios de la base de datos se podrá determinar cómo se debe proceder en cada caso.

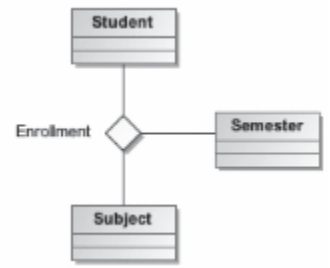
3.3. Tipos de relación ternarias

Se define un **tipo de relación ternaria** como la asociación entre tres tipos de entidad.

Hay situaciones en las que no basta con modelar una asociación entre dos tipos de entidad para representar un concepto y es necesario modelar la asociación entre tres tipos de entidad para representar el concepto deseado.

Siguiendo un ejemplo visto antes, no es suficiente modelar el tipo de relación *'Matrícula' (Enrollment)* entre los tipos de entidad *'Estudiante' (Student)* y *'Asignatura' (Subject)*. Dicha relación no permite representar la realidad de los estudiantes que suspenden una asignatura y tienen que volver a cursarla en un semestre posterior. En estos casos, una entidad *'Estudiante'* puede tener cero, una o más de una matrícula para una misma asignatura. El requisito es que solo es posible matricularse de una asignatura una vez por semestre. Por lo tanto, hay que introducir el tipo de entidad *'Semestre' (Semester)* y modificar el modelo para que la relación *'matrícula'* asocie los tres tipos de entidad. La figura 18 muestra el nuevo modelo.

Figura 18. Ejemplo de tipo de relación ternaria entre los tipos de entidad *'Estudiante' (Student)*, *'Asignatura' (Subject)* y *'Semestre' (Semester)*



3.3.1. Conectividad

En la conectividad de un tipo de relación ternaria hay implicados tres tipos de entidad, donde cada uno puede tomar la conectividad «a uno» o «a muchos».

En UML, podemos representar hasta cuatro tipos de conectividad para un tipo de relación ternaria[31]:

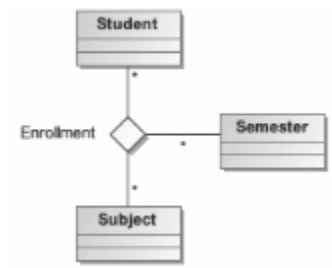
- Conectividad M:N:P o *..*..*.
- Conectividad M:N:1 o *..*..1.
- Conectividad M:1:1 o *..1..1.
- Conectividad 1:1:1 o 1..1..1.

Para decidir cómo se debe conectar cada uno de los tipos de entidad a la asociación, hay que fijar «a una» las otras entidades y entonces preguntarse si es posible conectar con la relación «a una» o «a muchas» entidades del primer tipo de entidad.

Veamos cómo definir la conectividad en un tipo de relación ternaria mediante el ejemplo que ilustra la figura 19, donde podemos ver los tres tipos de entidad, *'Estudiante' (Student), 'Asignatura' (Subject)* y *'Semestre' (Semester)*, y el tipo de relación *'Matrícula' (Enrollment)*.

31. En un tipo de relación ternaria se podrían representar hasta 12 tipos de restricciones de cardinalidad distintas. No obstante, la dificultad de mostrar gráficamente todas las restricciones de cardinalidad para los tipos de relación con un grado mayor a dos hace que los lenguajes de modelado muestren solo algunas de ellas. Si el lector está interesado, puede encontrar más información al respecto en el capítulo 4 del libro de Antoni Olivé listado en el apartado de bibliografía.

Figura 19. Ejemplo de conectividad en un tipo de relación ternaria



En primer lugar, nos preguntamos si dados una asignatura y un semestre concretos, cuántos estudiantes se pueden tener matriculados: «uno» o «muchos». La respuesta es que puede haber «muchos» estudiantes matriculados, puesto que varios estudiantes pueden matricularse de una misma asignatura en el mismo semestre. Por lo tanto, el tipo de entidad *'estudiante'* participa con grado N en la relación *'matrícula'*.

En segundo lugar, nos preguntamos, si fijados un estudiante y una asignatura concretos, puede existir matrícula en «uno» o «muchos» semestres. La respuesta es que pueden existir «muchos» semestres, puesto que un estudiante se puede matricular de una asignatura en diferentes semestres hasta superarla. Por lo tanto, el tipo de entidad *'semestre'* participa con grado N en la relación *'matrícula'*.

Finalmente, nos preguntamos si fijados un estudiante y un semestre concretos, pueden tener matrícula de «una» o «muchas» asignaturas. La respuesta es que se pueden tener «muchas» asignaturas matriculadas, puesto que un estudiante se puede matricular en varias asignaturas en un mismo semestre. Por lo tanto, el tipo de entidad *'asignatura'* también participa con grado N en la relación matrícula.

Así pues, vemos que el tipo de relación *'matrícula'* tiene cardinalidad M:N:P o *..*..*.

3.4. Tipos de relación n-arias

> Se define un **tipo de relación *n*-aria** como la asociación entre *N* tipos de entidad.

Aunque las relaciones binarias son relaciones *n*-arias por definición, normalmente se usa el término relación *n*-aria para referirse a las relaciones cuaternarias o de grado superior.

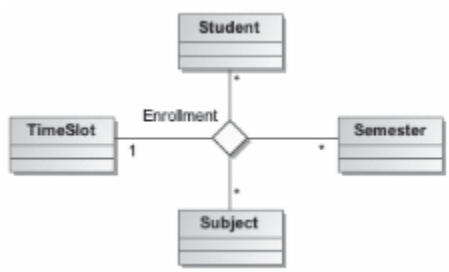
El tipo de relación ternaria es un caso concreto de tipo de relación *n*-aria. Lo que hemos visto en el subapartado anterior se puede generalizar para los tipos de relación *n*-arias. Por lo tanto, en una relación *n*-aria puede haber $n + 1$ tipos de conectividad, puesto que cada una de las *n* entidades puede estar conectada «a uno» o «a muchos» tipos en la asociación.

La figura 20 presenta una ampliación del ejemplo anterior, en el que se presentaban los tipos de entidad *'Estudiante' (Student), 'Asignatura' (Subject)* y *'Semestre' (Semester)* asociados mediante el tipo de relación *'Matrícula' (Enrollment)*. En este caso queremos añadir el concepto de turno o franja horaria en el esquema para indicar si el estudiante se matricula en el turno de mañana o de tarde. Para modelar este concepto, hemos creado el tipo de entidad *'Franja horaria' (TimeSlot)*.

Para determinar el grado de conectividad del nuevo tipo de entidad seguiremos el mismo proceso visto para el caso del tipo de relaciones ternarias. En este caso, nos preguntamos si, con referencia al tipo de relación *'Matrícula'* y fijados un estudiante, un semestre y una asignatura con-

cretos, puede haber «una» o «muchas» franjas horarias. La respuesta es que solo puede haber «una» franja horaria. Por lo tanto, el tipo de entidad de *'Franja horaria'* o *'Turno' (TimeSlot)* participa con grado 1 en este tipo de relación. Después de hacerse estas preguntas para cada tipo de entidad participante en la relación, el resultado es el que se muestra en la figura 20.

Figura 20. Ejemplo de conectividad en un tipo de relación cuaternaria



3.5. Tipos de relación reflexivas o recursivas

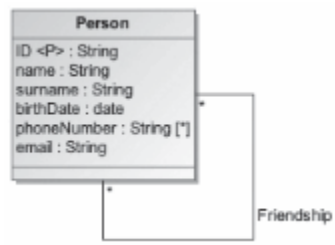
> Un **tipo de relación reflexiva o recursiva** es un tipo de relación binaria en la que un tipo de entidad se relaciona consigo misma.

Se presenta una relación entre dos entidades del mismo tipo de entidad. Igual que en el resto de los tipos de relación binarias, pueden tener conectividad 1:1, 1:N o M:N y pueden expresar dependencia de existencia.

Un claro ejemplo de un tipo de relación recursiva binaria es el concepto de amistad entre dos personas. En este

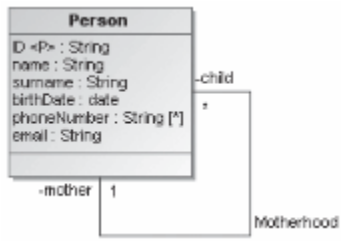
caso, tenemos el tipo de entidad *'Persona' (Person)* y el tipo de relación *'Amistad' (Friendship)*, que une dos entidades de *'Persona'*. La figura 21 muestra su modelo conceptual. La conectividad de la asociación es M:N, puesto que una persona puede tener varios amigos y, a la vez, puede haber varias personas que la tengan de amiga.

Figura 21. Ejemplo de tipo de relación recursiva



En las relaciones recursivas se aconseja definir el rol de cada tipo de entidad participante para clarificar mejor la semántica de la relación y qué rol desempeña cada participante en la misma. Por ejemplo, la figura 22 muestra un tipo de relación recursiva que describe la relación de maternidad biológica. Podemos ver que una persona puede tener cero o más hijos, y que toda persona debe tener una madre y solo una. Las etiquetas o nombres de rol, en este caso, pueden ayudar a comprender el rol de cada entidad en esta relación recursiva. Sin poner los nombres de rol sería difícil entender quién es el descendiente y quién la madre, pero después de añadir los nombres de rol se entiende perfectamente el papel que desempeña cada participante.

Figura 22. Ejemplo de tipo de relación recursiva con nombres de rol



3.6. Tipos de entidad asociativas

En algunos casos, las relaciones pueden requerir atributos que permitan describir propiedades relevantes sobre los tipos de relación. Estos atributos se definen y siguen las mismas reglas que los atributos de las entidades pero se definen en el contexto de un tipo de relación. En UML se usan clases asociativas (o tipos de entidad asociativos) para solucionar dicha problemática.

Un **tipo de entidad (o clase) asociativa** es el resultado de considerar una relación entre entidades como si fuera una entidad. Su nombre se corresponde con el nombre de la relación sobre la que se define.

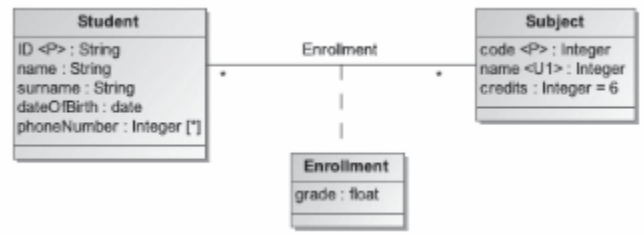
Una clase asociativa se representa como la unión de un tipo de entidad con un tipo de relación. El tipo de entidad de una clase asociativa se utiliza como contenedor de los atributos del tipo de relación.

Las clases asociativas añaden nuevas funcionalidades a las relaciones. Las principales son las siguientes:

1. Permiten definir atributos que hacen referencia a la relación. Es decir, atributos que son específicos de la asociación que hay entre las dos instancias de los dos tipos de entidad que forman la relación.
2. Permiten que un tipo de relación participe en otro tipo de relación. Esto puede hacerse porque la clase asociada a la relación puede participar en otros tipos de relación.

Supongamos el tipo de relación *'Matrícula' (Enrollment)* entre los tipos de entidad *'Estudiante' (Student)* y *'Asignatura' (Subject)*. Esta relación indica en qué asignaturas están matriculados cada uno de los estudiantes. Supongamos que queremos almacenar la nota que ha sacado cada estudiante en las asignaturas matriculadas. En un primer momento, podríamos pensar en situar el atributo *'nota' (grade)* en la entidad *'Estudiante' (Student)*, pero ello obligaría a que el estudiante tuviera siempre la misma nota en todas las asignaturas matriculadas. Otra opción sería situar el atributo en la entidad *'Asignatura' (Subject)*, pero en este caso estaríamos obligando a que todos los estudiantes de una misma asignatura tengan la misma nota. Por tanto, está claro que el atributo *'nota'* es un atributo de la relación *'Matrícula'*. En este caso, es necesario convertir la asociación *'Matrícula'* en una clase asociativa y añadir el atributo *'nota' (grade)* a dicha clase. El esquema conceptual resultante puede verse en la *figura 23*.

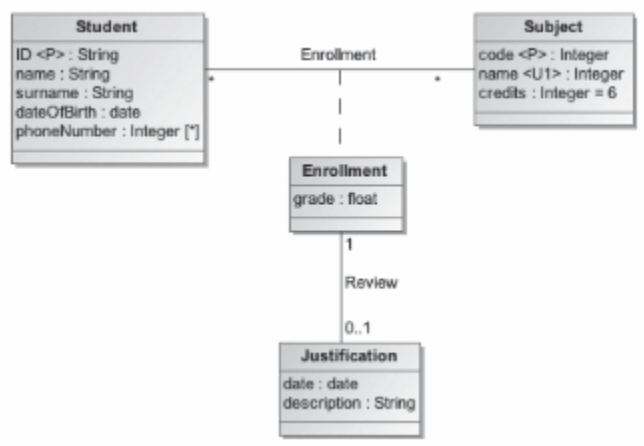
Figura 23. Ejemplo del tipo de entidad asociativa 'Matrícula' (Enrollment)



Tal y como se ha comentado, los tipos de entidades asociativas pueden relacionarse con otras entidades y también con otras entidades asociativas, es decir, permiten crear relaciones entre relaciones.

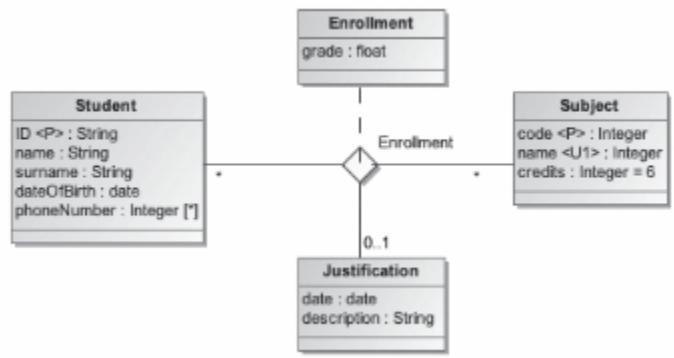
Por ejemplo, supongamos que queremos añadir en el ejemplo anterior el concepto de *justificación* o *retorno personalizado* de la nota. Es decir, queremos expresar que en algunos casos, por ejemplo cuando el estudiante lo solicite, el estudiante recibirá una descripción que justifique la puntuación obtenida y los errores que ha cometido en una asignatura determinada. La figura 24 muestra el modelo conceptual con el nuevo requisito. Podemos ver cómo el nuevo tipo de entidad *'Justificación' (Justification)* indica la fecha y la descripción de la justificación y se relaciona con algunas matriculaciones. Este punto es relevante, puesto que, dada la cardinalidad del tipo de relación *'Revisión' (Review)*, se puede deducir que pueden existir parejas estudiante-asignatura sin una justificación.

Figura 24. Ejemplo de tipo de relación entre una entidad y una entidad asociativa



Cabe señalar que esta misma situación no se puede modelar mediante un tipo de relación ternaria, vamos a ver con un ejemplo el porqué. Supongamos que creamos un diagrama UML para representar la relación *Matrícula* mediante una clase asociativa de tipo ternaria (ver figura 25). En este caso, el significado del modelo es diferente del que planteábamos en el ejemplo anterior.

Figura 25. Ejemplo de error de expresión sobre el ejemplo de la figura 24



El nuevo modelo nos dice que para cada estudiante *(Student)* y asignatura *(Subject)* puede haber ninguna o una justificación *(Justification)*. Pero también indica que fijadas una justificación y una asignatura puede haber muchos estudiantes, afirmación que es falsa según los requisitos iniciales. Igualmente, el modelo indica que fijados un estudiante y una justificación puede haber muchas asignaturas, afirmación que también es falsa. Si modificamos las cardinalidades de los tipos de entidad *'estudiante'* y *'asignatura'* y las establecemos a 1, entonces solo podríamos representar los estudiantes que tienen asociada una justificación en su matrícula.

El problema reside en el hecho de que intentamos representar con un tipo de relación ternaria un modelo que es binario. La *justificación* no es necesaria para todas las matrículas, pero al ser incluida como participante en el tipo de relación, obligamos a que toda matrícula tenga una justificación. Obviamente no es lo que queríamos. Eso se debe a que el concepto de *justificación* no hace referencia a las entidades *'Estudiante'* y *'asignatura'*, sino a *'Matricula'* y, para ello, es necesario que sea modelado como un tipo de entidad que se relaciona con la clase asociativa *'Matricula'*.

Es importante que se entienda que la información representada en los diagramas de las figuras 24 y 25 es distinta y que, en términos generales, no se representa la misma información con un tipo de relación ternaria que con una clase asociativa binaria.

4. Tipos de entidad débiles

> Se define un **tipo de entidad fuerte**[32] como un tipo de entidad cuyos atributos propios permiten formar claves candidatas y, por tanto, identificar de manera unívoca todas sus instancias.

La existencia de las instancias de un tipo de entidad fuerte no depende de ningún otro tipo de entidad o tipo de relación. Es decir, la semántica detrás de una entidad fuerte no necesita de ninguna relación con otras entidades que complementen su significado.

> Se define un **tipo de entidad débil**[33] como un tipo de entidad cuyos atributos no permiten identificar sus entidades completamente, sino que solo las identifican de manera parcial. Estas entidades deben participar en una relación con otras entidades que ayuden a identificarlas de manera unívoca.

Un tipo de entidad débil, por lo tanto, necesita una asociación con otro tipo de entidad que le permita identificar de manera única sus instancias. Esta asociación recibe el nombre de asociación identificativa del tipo de entidad débil. El tipo de entidad débil debe tener una restricción de participación total respecto a su asociación identificativa

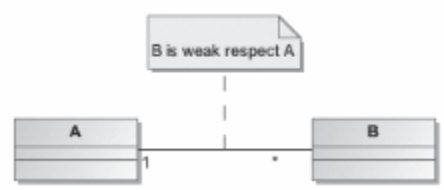
32. Los tipos de entidad fuertes también se pueden denominar tipos de entidad propietarias, dominantes o padres.
33. Los tipos de entidad débiles también se pueden denominar tipos de entidad subordinadas o hijas.

puesto que de no ser así, habría instancias que no se podrían identificar de manera única. El tipo de relación identificativa tiene una conectividad 1:N, con la entidad débil en el lado de la N.

Una entidad débil no tiene sentido si se elimina la entidad con la que está relacionada. Por lo tanto, si se elimina una entidad, hay que eliminar las entidades débiles que dependen de ella.

En los diagramas UML un tipo de entidad débil se indica mediante una nota en su asociación identificativa. La figura 26 muestra la notación para modelar un tipo de entidad débil utilizando diagramas UML.

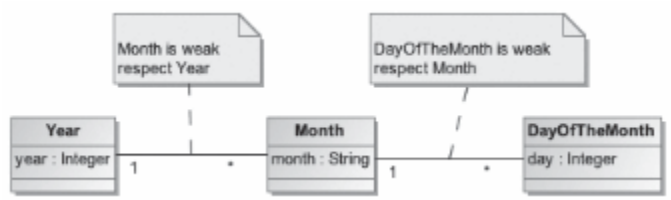
Figura 26. Notación UML para indicar un tipo de entidad débil (B)



Un tipo de relación identificativa no suele tener atributos, puesto que en las relaciones 1:N siempre es posible mover sus atributos al tipo de entidad débil, es decir, en la parte con cardinalidad N.

La figura 27 muestra el tipo de entidad débil *'Día del mes' (DayOfTheMonth)* relacionada con *'Mes del año' (Month)*, que a su vez también es una entidad débil relacionada con el tipo de entidad fuerte *'Año' (Year)*.

Figura 27. Ejemplo de un conjunto de tipos de entidad débiles interrelacionados



La clave primaria de un tipo de entidad débil está formada por la combinación de la clave primaria del tipo de entidad relacionado y la clave parcial del tipo de entidad débil.

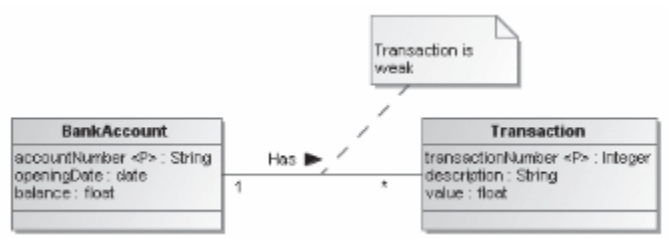
La **clave parcial** de un tipo de entidad débil es el conjunto de atributos que permiten identificar de manera única todas sus instancias que están relacionadas con una misma instancia del tipo de entidad relacionada.

En el peor de los casos, la clave parcial es el conjunto de todos los atributos del tipo de entidad débil.

Por ejemplo, podemos considerar el concepto *cuenta corriente* de una entidad bancaria como un tipo de entidad fuerte, puesto que posee el atributo *'número de cuenta' (accountNumber)* que permite identificar de manera única cada una de sus instancias. En cambio, si consideramos el concepto de *transacción*, nos damos cuenta de que solo tiene sentido y puede existir si está asociado a una cuenta bancaria. Por lo tanto, para modelar esta situación se puede crear un tipo de entidad fuerte, *BankAccount*, y un tipo de entidad débil, *Transaction*, como se muestra en la figura 28. La clave primaria del tipo de entidad *BankAccount* es el atributo *accountNumber*. En el tipo de entidad *Transaction* la clave parcial es el atributo

transactionNumber, que indica el número de transacción. Por lo tanto, la clave primaria del tipo de entidad débil es compuesta y está formada por los atributos {*accountNumber*, *transactionNumber*}.

Figura 28. Ejemplo de tipo de entidad débil *(Transaction)* y la relación identificativa con el tipo de entidad fuerte *(BankAccount)*



5. Opciones de diseño

Hasta este punto se han mostrado los elementos de modelado conceptual más comunes en el diseño de bases de datos. Conocer qué elementos podemos utilizar para modelar un dominio, y modelarlo son dos cosas muy diferentes. Crear esquemas conceptuales tiene mucho de arte y la práctica es la mejor manera de convertirse en un buen modelador conceptual. Por otro lado, hay que tener en cuenta que el proceso de modelado conceptual no es determinista, es decir, no existe un único diagrama conceptual para un problema concreto y, por tanto, diferentes diseñadores pueden generar diferentes esquemas conceptuales que representen la misma

información. Por ejemplo, en algunas situaciones no es trivial determinar si un concepto debe ser modelado como un tipo de entidad, un atributo o un tipo de relación.

A continuación, mencionamos algunos consejos que pueden ser útiles durante el diseño conceptual de una base de datos:

- El proceso de diseño conceptual de una base de datos se puede enfocar como un proceso iterativo de refinamiento, donde se crea un primer diseño inicial y se va refinando de manera iterativa hasta conseguir el nivel de diseño deseado.
- Es frecuente modelar un concepto como un atributo en un primer momento, y en refinamientos posteriores convertirlo en un tipo de relación. Modelar el atributo como una relación permite categorizar los valores de este concepto y, además, que se puedan añadir nuevos atributos relacionados con el dominio del atributo.
- Es habitual que un atributo existente en varios tipos de entidad sea convertido en un nuevo tipo de entidad. A continuación hay que establecer relaciones desde los tipos de entidad que contenían este atributo hacia el nuevo tipo de entidad[34].
- También puede existir el refinamiento inverso. En este caso, se puede eliminar un tipo de entidad y represen-

34. De atributo a tipo de entidad: Si los tipos de entidad *Estudiante*, *Profesor* y *Curso* tienen el atributo departamento, es interesante crear el nuevo tipo de entidad *Departamento*, con un único atributo que indica el nombre del departamento, y crear tres tipos de relación binarias entre *Estudiante*, *Profesor* y *Curso*, y el nuevo tipo de entidad *Departamento*.

tar el concepto como un atributo si solo afecta a un tipo de entidad (o a muy pocos) y el tipo de entidad eliminado solo tiene un atributo.

6. Criterios de asignación de nombres

En el diseño del esquema conceptual de una base de datos es importante elegir adecuadamente los nombres de los tipos de entidad, atributos y tipos de relación que aparecen y usar criterios homogéneos en todo el esquema. El objetivo es escoger nombres que eviten confusión y que transmitan de la mejor manera posible el significado de las diferentes estructuras del esquema.

No hay una normativa clara a la hora de bautizar los nombres de los diferentes elementos de un esquema conceptual. A continuación se presentan un conjunto de criterios basados en la experiencia y el gusto particular de los autores de este libro.

El primer criterio es utilizar el inglés para definir los nombres de los elementos del esquema conceptual. Es importante escoger qué «dialecto» o tipo de lenguaje utilizaremos. Una palabra en castellano de España, por ejemplo, puede tener un significado diferente en Argentina y dar lugar a confusiones y errores. En nuestro caso, utilizamos el inglés americano. Antes de asignar un nombre a un elemento, es importante buscar en el diccionario para comprobar que el nombre candidato es correcto y no ofrece ambigüedad. También es

aconsejable utilizar tesauros para ver si podemos encontrar un nombre mejor.

A continuación estableceremos los criterios utilizados para cada elemento de modelado UML:

- Etiquetar los tipos de entidad utilizando nombres simples en singular siempre que sea posible. Si es necesario, se pueden utilizar hasta dos o tres nombres. En nuestro caso, empleamos la grafía *Pascal*[35] para escribir el nombre de los tipos de entidad.
- Etiquetar los atributos utilizando nombres simples en singular y evitando que aparezca el nombre del tipo de entidad. Si es necesario, se pueden utilizar hasta dos o tres nombres o la combinación de un nombre y uno o dos adjetivos. En este libro se ha empleado la grafía *Camel*[36] para escribir el nombre de los atributos.
- En el caso de atributos booleanos, se suele utilizar el nombre o nombres deseados precedidos del verbo "es" (de la forma verbal *es*, o *is* en la versión inglesa). El objetivo es enfatizar el valor booleano, que responde con un sí o un no ante una pregunta. Por ejemplo: *esNegrita* o *isBold*.
- Etiquetar los tipos de relación utilizando verbos en tercera persona del singular. En caso de requerir etiquetas compuestas, se suele usar un verbo seguido de

35. Grafía *Pascal*: la primera letra del identificador y la primera letra de las siguientes palabras se escriben en mayúscula. El resto, en minúscula. Por ejemplo: *BackColor*.
36. Grafía *Camel*: la primera letra del identificador se escribe en minúscula y la primera letra de las siguientes palabras concatenadas, en mayúscula. El resto, en minúscula. Por ejemplo: *backColor*.

una preposición y otro verbo. Al igual que en los tipos de entidad, las etiquetas de los tipos de relación siguen la grafía *Pascal*.

- Las etiquetas para identificar los roles de las entidades en las relaciones siguen la grafía *Camel*.

Como práctica general, a partir de una descripción funcional de los requisitos de la base de datos, los nombres que aparecen tienden a ser candidatos de nombres para los tipos de entidad que se vayan a definir, mientras que los verbos tienden a ser utilizados para los tipos de relación del esquema conceptual. Como criterio general, los nombres de los tipos de relación se eligen para que se puedan leer de izquierda a derecha y de arriba abajo. De este modo, se facilita la lectura de las relaciones. Puede haber excepciones a estas normas, y en estos casos nos podemos ayudar de los nombres de roles o indicar la dirección de lectura tal y como se ha explicado en el apartado de definición de tipos de relación. También es aconsejable, en algunos casos, utilizar nombres de roles para facilitar la comprensión de las relaciones y el papel de las entidades en cada una de estas relaciones. Los atributos suelen ser nombres extraídos de las descripciones de los objetos definidos como tipos de entidad.

7. Ejemplo completo

En este capítulo hemos visto los elementos básicos del diseño conceptual según el modelo UML. Pero antes de continuar con los elementos avanzados de modelado en el siguiente capítulo, queremos ilustrar cómo se utilizan en la práctica los conceptos vistos hasta ahora para crear esquemas conceptuales de bases de datos. Para facilitar el análisis de un ejemplo más o menos real, a continuación plantearemos una descripción y un conjunto de requisitos y restricciones que deben permitir implementar un modelo conceptual de una base de datos destinada a la gestión universitaria. Este modelo es una simplificación de un modelo real y, por tanto, no tiene como objetivo ser capaz de dar soporte a la gestión real de una universidad. Solo pretende servir como modelo de ejemplo para los conceptos vistos hasta ahora.

A continuación se enumeran los diferentes aspectos de los requisitos de los usuarios que hay que tener en cuenta al realizar el diseño conceptual de la futura base de datos:

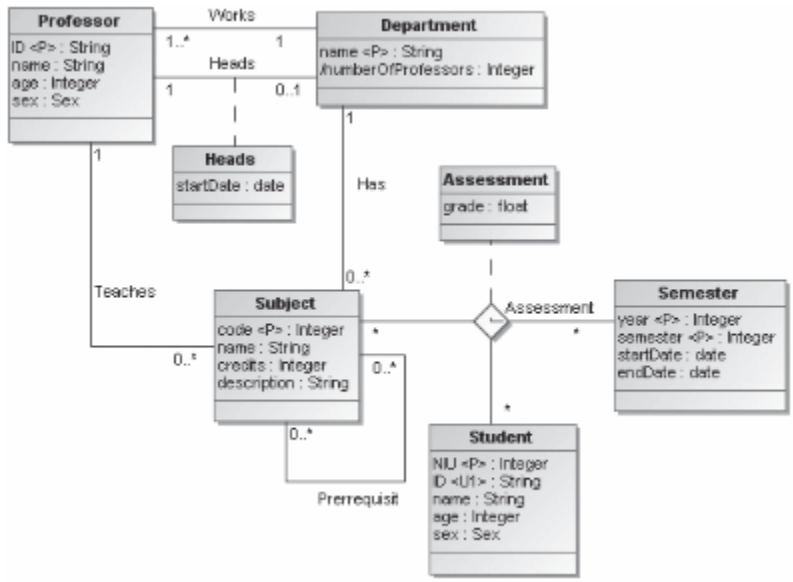
1. La universidad está formada por diferentes departamentos. Cada departamento está compuesto por un conjunto de profesores y está dirigido por un único profesor. Nos interesa conocer el nombre de los departamentos y el número de profesores adscritos a cada departamento.
2. De cada profesor nos interesa almacenar sus datos personales, como, por ejemplo, nombre, DNI, edad y si es un hombre o una mujer. De los profesores que son

directores de departamento, nos interesa poder identificar la fecha en la que empezaron a ejercer este cargo.

3. En la universidad se imparten un conjunto de asignaturas. Nos interesa conocer el código de cada una de estas, el nombre, el número de créditos, la descripción y los prerrequisitos de cada una de ellas. Entendemos como prerrequisitos de una asignatura el conjunto de otras asignaturas que es necesario haber cursado para poder realizar la asignatura con éxito. Además, hay que tener en cuenta que una asignatura pertenece a un único departamento y es impartida por un único profesor.
4. Los estudiantes son una parte muy importante de la base de datos. De ellos, se desea almacenar información sobre sus datos personales (nombre, DNI, edad, sexo) y número de identificación universitario (conocido como NIU).
5. Cada estudiante puede estar matriculado en más de una asignatura en cada semestre. Y para cada una de las asignaturas en las que está matriculado cada semestre debemos poder almacenar su nota final.
6. También se quiere dejar constancia de las fechas de inicio y final de cada semestre.

A partir de los requisitos expresados, la figura 29 muestra un posible esquema conceptual que representa la información expresada en los puntos anteriores. Como ya se ha descrito, este modelo no es único, y pueden existir diferentes aproximaciones y conceptualizaciones válidas para una misma representación del mundo real.

Figura 29. Diagrama del modelo conceptual para la base de datos de gestión universitaria



A continuación se describen los aspectos más relevantes de este modelo conceptual:

- En el tipo de entidad *'Departamento' (Department)*, el atributo *'número de profesores' (numberOfProfessors)* es un atributo derivado, puesto que se calcula a partir del número de profesores que están vinculados a cada departamento.
- Los tipos de entidad *'Departamento' y 'Profesor' (Professor)* tienen dos tipos de relación, una para indicar en qué departamento trabaja cada profesor, y otra para determinar qué profesor es el director de cada departamento. La conectividad de la primera asociación indica que un profesor debe pertenecer a un

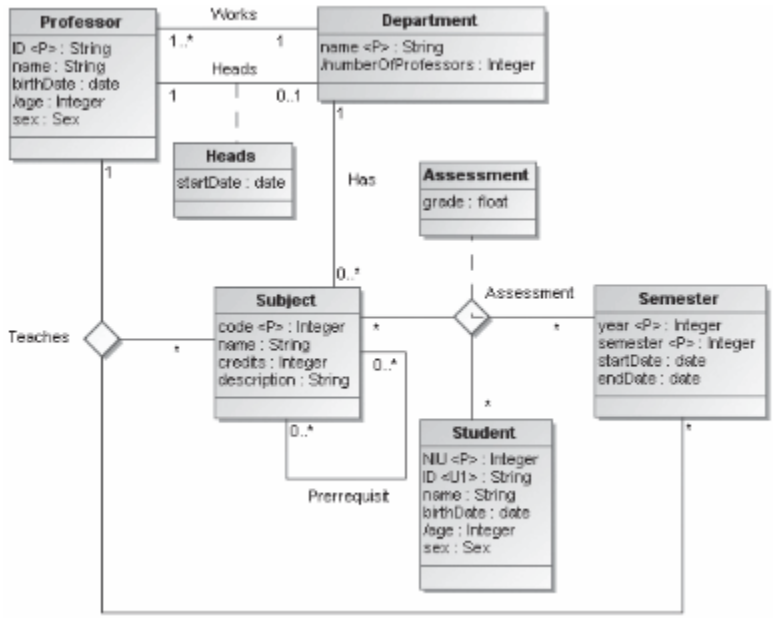
departamento y que los departamentos deben tener como mínimo un profesor. La segunda relación es un tipo de entidad asociativa y contiene el atributo *'fecha de inicio' (startDate)* para indicar la fecha en la que el profesor empieza a dirigir el departamento. Un profesor puede impartir cero, una o más de una asignatura, como indica la conectividad del tipo de relación con el tipo de entidad *'Asignatura' (Subject)*.

- El concepto prerrequisito de una asignatura se ha modelado como un tipo de relación recursiva que asocia diferentes entidades de asignaturas. Tal como se desprende de la conectividad, una asignatura puede tener ninguno o múltiples prerrequisitos y ser a la vez prerrequisito de ninguna o de múltiples asignaturas.
- El semestre se ha modelado como un tipo de entidad *(Semester)* con una clave primaria compuesta formada por el año académico *'year'* y el número semestre *'semester'*.
- El tipo de relación ternaria entre los tipos de entidad *'Asignatura'*, *'Semestre'* y *'Estudiante' (Student)* determina la nota obtenida por los estudiantes en las asignaturas en que se matriculan cada semestre. Tal como se expresaba en los requisitos, un estudiante puede estar matriculado de múltiples asignaturas en un semestre, y se puede matricular de la misma asignatura en varios semestres. Como hay un atributo ligado al tipo de relación, se ha creado el tipo de relación como una clase asociativa para poder alojar el atributo *'grade'*.

Pero este modelo conceptual también presenta algunas deficiencias, que comentamos a continuación:

- Según el modelo, un profesor imparte un conjunto de asignaturas, pero no sabemos qué profesor impartió una asignatura en un semestre concreto. Es decir, el tipo de relación *'imparte' (Teaches)* indica el profesor que imparte una asignatura, pero no permite saber qué profesor la había impartido en semestres anteriores.
- En los tipos de entidad *'Profesor'* y *'Estudiante'* se almacena la edad mediante un atributo numérico que contiene el valor para cada profesor o estudiante en un momento determinado. Esto implica que cada año deberá actualizarse la base de datos y sumar 1 a los valores de este atributo. Lógicamente, este modelo no es el adecuado y habría que almacenar la fecha de nacimiento de los profesores y los estudiantes y crear un atributo derivado que indique su edad *(age)* de manera automática.

Figura 30. Diagrama del modelo conceptual con las correcciones aplicadas



La figura 30 muestra el modelo conceptual modificado para corregir las deficiencias detectadas. El tipo de relación *'Imparte' (Teaches)* se ha convertido en un tipo de relación ternaria. De este modo, podemos indicar qué profesor es responsable de cada asignatura en cada semestre. También se han creado los atributos derivados *'age'* para calcular automáticamente la edad de los profesores y estudiantes.

Capítulo IV

Elementos avanzados de modelado

Los elementos de modelado vistos hasta ahora permiten representar cualquier esquema de bases de datos de aplicaciones tradicionales. Pero para poder representar de manera más precisa y directa algunas características del mundo real, se utilizan otros elementos de modelado que permiten más potencia y flexibilidad. En este apartado veremos algunos de dichos elementos que permiten representar fácilmente realidades difícilmente expresables utilizando los elementos de modelado vistos en el capítulo anterior.

1. Generalización/especialización

En algunos casos, puede ser interesante representar la información de un dominio de forma incremental. Es decir, definiendo los tipos de entidad según distintos niveles de abstracción y relacionando los tipos de entidad más concretos con los de carácter más general. De esta forma se puede reutilizar en las nuevas clases el conocimiento general que está definido en las clases generales.

Por ejemplo, supongamos el tipo de entidad *'Persona'* en un entorno universitario. Sus atributos son su *nombre*, *apellidos*, *DNI*, *fecha de nacimiento*, *sexo*, *teléfono* y *correo electrónico*. En la base de datos queremos tratar con diferentes tipos de personas: profesores, administrativos y estudiantes. Los estudiantes tienen algunas características propias que son relevantes para la base de datos y que no comparten los profesores y los administrativos, como por ejemplo el *número de identificación de estudiante* y el *año de primera matrícula*. De los profesores queremos almacenar sus *áreas de especialidad*. Y del personal administrativo queremos almacenar su *cargo* y la *sección donde trabajan*. Es decir, todas las entidades tienen un conjunto de características comunes, pero también tienen algunas características diferentes entre sí que hay que poder representar en el modelo de base de datos. Además, podemos ver que cada uno de los tres colectivos tiene una semántica muy definida y distinta a los demás.

Cuando nos encontramos con casos como el del ejemplo anterior, se puede crear una jerarquía de tipos de entidad que permita definir una clase en diferentes niveles de abstracción. Por ejemplo, en el caso anterior podríamos crear una jerarquía de tipos de entidad con *Persona* como clase más genérica y *Estudiante*, *Profesor* y *Administrativo* como clases específicas.

La **generalización/especialización**[37] es una relación entre un tipo de entidad general, que denominamos *superclase*

37. La relación de generalización/especialización también se denomina relación *'es-un'* o *'es-una'* (o *'is a'* en inglés) por la manera en que se hace referencia al concepto: un estudiante *'es una'* persona, un profesor *'es una'* persona.

(o tipo de *entidad padre*), y un tipo de entidad más específico, denominado *subclase* (o tipo de *entidad hijo*).

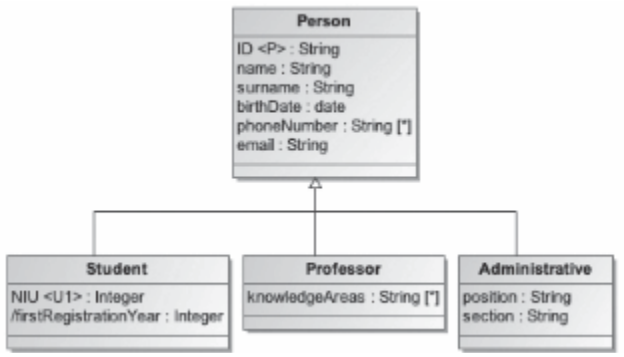
En notación UML, esta relación se representa mediante una flecha vacía que sale de las subclases y que se dirige hacia la superclase.

La superclase define la información más general y acostumbra a incluir las características comunes a todas sus subclases, como por ejemplo los atributos *nombre*, *apellidos*, *DNI*, *fecha de nacimiento*, *sexo*, *teléfono* y *correo electrónico* del ejemplo anterior, que son genéricos a todas las subclases y se definen en la superclase. Las subclases, por otro lado, definen información más concreta y acostumbran a definir características que no son comunes al resto de subclases, como podría ser el atributo *áreas de especialidad* de la clase *Profesor*. Las subclases se denominan especializaciones de la superclase y la superclase se denomina generalización de las subclases.

Los atributos o relaciones de la superclase se propagan hacia las subclases. Es lo que se denomina herencia de propiedades. Por tanto, cualquier entidad que pertenezca a cualquiera de las subclases puede tener valor para los atributos que se definen en la superclase.

Los atributos de las subclases reciben el nombre de atributos específicos o atributos locales. Solo las entidades que pertenecen a una subclase determinada pueden tener un valor asignado a estos atributos.

Figura 31. Ejemplo de generalización/especialización



Siguiendo el ejemplo anterior, la figura 31 muestra un esquema conceptual en UML que utiliza generalización/especialización para representar los diferentes tipos de personas que podemos encontrar en el entorno universitario descrito. La superclase es el tipo de entidad *'Persona' (Person)*, que incluye los datos básicos que nos interesa almacenar para cualquier persona del modelo. A continuación encontramos las tres subclases descritas antes, que contienen la información relevante de los estudiantes *(Student)*, profesores *(Professor)* y personal de administración *(Administrative)*. Por lo tanto, cualquier entidad de la subclase *'Estudiante'* tiene disponibles todos los atributos de la superclase *'Persona'* más los atributos específicos o locales de la clase *'Estudiante'*.

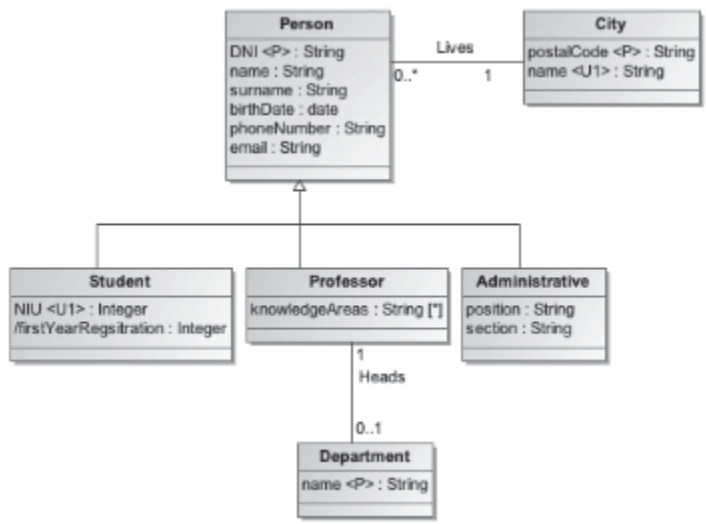
Al ser tipos de entidad normales, tanto las superclases como las subclases pueden intervenir en relaciones sin ningún tipo de restricción y pueden participar en otras relaciones de generalización/especialización. Por otro lado, al igual como pasa con los atributos, las subclases heredan los tipos de relación definidos en sus superclases.

Tal y como se ha comentado, una jerarquía de clases permite definir las clases en diferentes niveles de abstracción. Es

importante escoger adecuadamente los posibles participantes de un tipo de relación dentro de una jerarquía de clases.

Por ejemplo, supongamos el ejemplo de la figura 32, que extiende el ejemplo anterior con información de donde viven las personas y de los directores de departamento de la universidad. En el esquema se puede ver cómo la superclase *'Persona' (Person)* participa en la relación *'vive' (Lives)* con la entidad *'Población' (City)* y la subclase *'Profesor' (Professor)* participa en la relación *'dirige' (Heads)* con la entidad *'Departamento' (Department)*. Evidentemente no tiene sentido considerar que cualquier entidad de la superclase *'Persona'* puede dirigir un departamento y hay que indicar en el modelo conceptual que solo aquellas personas que son entidades de la subclase *'Profesor'* pueden dirigir un departamento. Definiendo la relación a nivel de *'Profesor'* garantizamos que solo los profesores puedan dirigir departamentos.

Figura 32. Ejemplo de generalización/especialización con tipos de relación



Por lo tanto, un proceso de generalización/especialización permite[38]:

- Crear una taxonomía de tipos de entidad, definiendo un conjunto de entidades tipo específicas (subclases) a partir del tipo de entidad genérica (superclase).
- Establecer atributos específicos adicionales a cada tipo de entidad específico (subclase).
- Establecer tipos de relación adicionales entre los tipos de entidad específicos (subclases) y otros tipos de entidad.

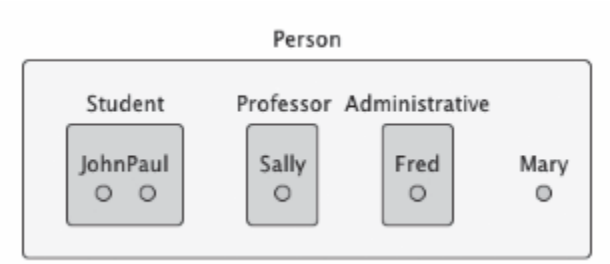
1.1. ¿Cómo afecta la jerarquía de clases a las instancias?

Conceptualmente, la herencia (o jerarquía) se utiliza para especificar tipos de entidad más específicos y que, por tanto, contienen menos entidades. Formalmente, la generalización/especialización entre dos clases (superclase y subclase) define una restricción de integridad de pertinencia entre las dos clases, que indica que toda instancia de la subclase es también instancia de la superclase. Así pues, si definimos que *Jordi Casas* es una instancia de la clase *Profesor*, automáticamente podremos inferir que *Jordi Casas* es también una *Persona*, porque *Profesor* es subclase de *Persona*. Las relaciones de herencia entre clases se modelan utilizando relaciones de generalización/especialización.

38. Observación: a partir de este punto, denominamos superclase a los tipos de entidad genéricos, y subclase a los tipos de entidad específicos.

La figura 33 muestra una posible relación entre las entidades del ejemplo anterior. Podemos ver que hay cinco entidades de la superclase *'Persona' (Person)*, con los nombres *John*, *Mary*, *Paul*, *Fred* y *Sally*, dos de las cuales son estudiantes y pertenecen a la subclase *'Estudiante' (Student)*, una es un profesor *(Professor)* y otra es administrativo *(Administrative)*. La entidad *Mary* no pertenece a ninguna de las subclases y solo pertenece a la superclase *Persona*.

Figura 33. Ejemplo de la relación entre las entidades en una generalización/especialización



1.2. Factores a tener en cuenta en la creación de jerarquías de clases

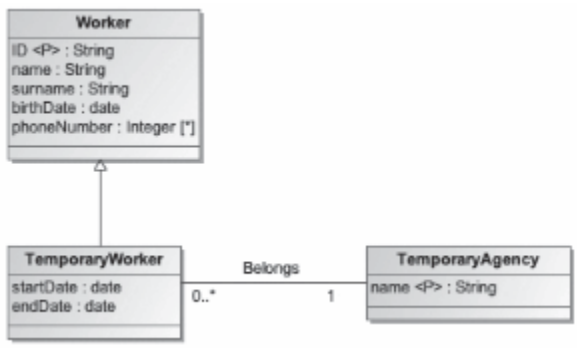
Hay dos motivos principales que inducen al uso de las relaciones de generalización/especialización entre tipos de entidad:

1. El primer motivo es debido a que, en determinados casos, hay que modelar situaciones en que algunos atributos solo son aplicables a algunas de las entidades, pero no a toda la clase. En este caso, se define una relación de generalización/especialización que permite agrupar estas entidades en una subclase y añadirles

los atributos necesarios, mientras continúan formando parte de la superclase y compartiendo el resto de los atributos con todas las entidades de la superclase.

2. Otro motivo se produce cuando hay que modelar tipos de relación que solo tienen sentido en algunas entidades de la clase. Por ejemplo, supongamos que hemos creado un tipo de entidad *'Trabajador' (Worker)* para almacenar a todos los trabajadores de una fábrica y queremos establecer un tipo de relación que indique de qué empresa de trabajo temporal provienen los trabajadores. Lógicamente, esta relación solo es aplicable a los trabajadores que sean temporales y no a los fijos de la empresa. Para modelar esta situación, podemos crear una subclase para identificar a los trabajadores temporales *(TemporaryWorker)* y relacionarlos con el tipo de entidad de las empresas temporales *(EmploymentAgency)*. De este modo, solo los trabajadores que pertenezcan a la subclase podrán estar relacionados con la empresa de trabajo temporal. La figura 34 muestra un esquema que ejemplifica este caso.

Figura 34. Ejemplo de generalización/especialización que incorpora relaciones con la subclase

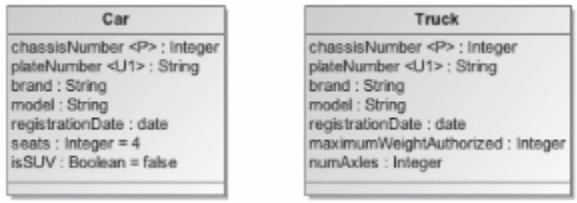


En el momento de realizar el diseño del modelo, se puede optar por una estrategia descendente *(top-down)* y empezar diseñando la superclase como tipo de entidad principal, y posteriormente refinar el diseño añadiendo las características específicas de las subclases. Esta metodología de diseño recibe el nombre de proceso de especialización y ha sido la empleada en los ejemplos presentados hasta ahora.

La estrategia contraria, basada en una metodología ascendente *(bottom-up)*, consiste en diseñar las subclases y posteriormente, en el momento de darse cuenta de las características comunes entre estas subclases, definir la superclase que las une. Esta metodología recibe el nombre de proceso de generalización y se emplea en el ejemplo siguiente.

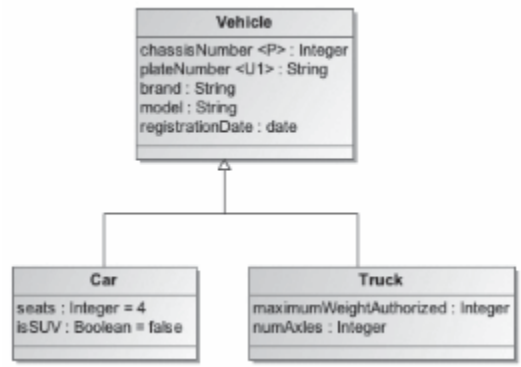
Supongamos que queremos modelar una base de datos para mantener información sobre los vehículos que tiene una empresa. En un momento del diseño aparece el tipo de entidad *'Coche' (Car)*, del que queremos almacenar el número de bastidor, la matrícula, la marca, el modelo, la fecha de matriculación, el número de plazas disponibles y si es un vehículo todoterreno o no. Más adelante, en la misma fase de diseño, aparece el tipo de entidad *'Camión' (Truck)*, del que queremos almacenar el número de bastidor, la matrícula, la marca, el modelo, la fecha de matriculación, el peso máximo autorizado de carga y el número de ejes del camión. La figura 35 muestra el diseño conceptual de los dos tipos de entidad.

Figura 35. Ejemplo de los tipos de entidad *'Coche' (Car)* y *'Camión' (Truck)*



Pero en este punto el diseñador se da cuenta de que los dos tipos de entidad comparten parte de los atributos y de que sería interesante generalizar esta parte común para obtener una nueva superclase que permita identificar un vehículo, sea camión o coche. Por lo tanto, se crea la superclase *'Vehículo' (Vehicle)*, que contiene todos los atributos comunes (el número de bastidor, la matrícula, la marca, el modelo y la fecha de matriculación). Después se crean dos subclases con los atributos propios de cada una de las clases anteriores. La figura 36 muestra el diseño conceptual después del proceso de generalización.

Figura 36. Ejemplo de generalización de los tipos de entidad *'Coche' (Car)* y *'Camión' (Truck)*



1.3. Restricciones en la generalización/especialización

Hay tres tipos de restricciones de integridad básicas que pueden aplicarse en los procesos de generalización/especialización. En particular, dada una superclase y N subclases, las restricciones de integridad vienen definidas por:

- La manera en que se clasifican las instancias de la superclase en las subclases.
- Si se permite que diferentes subclases compartan instancias.
- Si las instancias de la superclase son iguales a la unión de las instancias de sus subclases.

A continuación se describen cada una de estas restricciones, se muestra cómo se representan en UML y se ejemplifican mediante la creación de algunos esquemas conceptuales.

1.3.1. Atributos discriminadores

En algunos casos la superclase contiene un atributo (o un conjunto de atributos) que permite identificar, para cada entidad, a qué subclase pertenece. Por ejemplo, continuando con el ejemplo de la figura 36 (vehículos), podríamos crear en la superclase un atributo que defina el tipo de vehículo *(vehicleType)* con dos valores posibles: *Car* y *Truck*. En este caso, el valor de este atributo para una entidad de la superclase determinaría a qué subclase pertenece y diremos que el valor del atributo determina la semántica de las subclases.

En caso de que el valor del atributo de una superclase determine a qué subclase pertenece cada entidad de la superclase, este atributo recibe el nombre de discriminador o atributo definitorio y se dice que la relación de generalización/especialización es de atributo definido. En caso de que no exista ningún atributo que permita identificar la pertenencia de las entidades a las diferentes subclases, se dice que es una generalización/especialización definida por el usuario.

En UML, el discriminador, si lo hay, se indica en la flecha de la relación de generalización/especialización, como podemos ver en figura 38.

1.3.2. Restricción de exclusividad

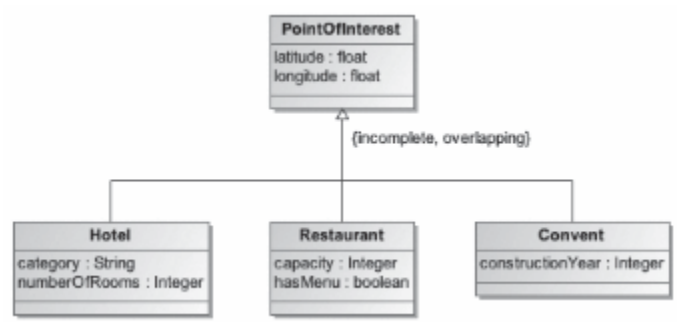
Otra restricción que se puede aplicar a las generalizaciones/especializaciones es la restricción de exclusividad, que indica si las subclases han de ser disjuntas entre sí. Dos tipos de entidad son disjuntos si no pueden compartir entidades. Dicho de otro modo, expresa si una misma entidad perteneciente a una superclase, puede pertenecer a más de una subclase. En este sentido, la generalización/especialización puede ser:

- **Disjunta**[39]: una entidad solo puede pertenecer a una única subclase.
- **Solapada**[40]: una entidad puede pertenecer a más de una subclase al mismo tiempo.

39. En inglés, *disjoint*.
40. En inglés, *overlapping*.

La figura 37 muestra un ejemplo de generalización/especialización no disjunta. En este modelo, podemos ver la superclase *'Punto de interés' (PointOfInterest)*, que permite almacenar información sobre diferentes puntos de interés turístico de una zona. Este concepto se puede especializar en diferentes subclases, como por ejemplo hoteles *(Hotel)*, restaurantes *(Restaurant)* y conventos *(Convent)*. Como hay hoteles que también son restaurantes o conventos que hacen funciones de hotel, nos encontramos con que una misma entidad podría pertenecer a más de una subclase al mismo tiempo. Por lo tanto, podemos decir que esta generalización/especialización es de tipo solapada. En UML se indica poniendo la palabra *overlapping* entre llaves.

Figura 37. Ejemplo de generalización/especialización solapada



1.3.3. Restricción de participación

La última de las restricciones referentes a la generalización/especialización se denomina restricción de participación e indica la obligatoriedad de las entidades de la supercla-

se a pertenecer a alguna de las subclases. De acuerdo con su participación, la herencia entre una superclase y sus subclases puede ser:

- **Total**[41]: toda entidad de la superclase debe pertenecer a alguna de las subclases.
- **Parcial**[42]: puede haber entidades de la superclase que no pertenezcan a ninguna de las subclases.

En una generalización/especialización con restricción de participación total no puede haber entidades que pertenezcan exclusivamente a la superclase, ya que todas las entidades deben pertenecer a alguna de sus subclases. En estos casos se dice que la superclase es un tipo de entidad abstracta, puesto que no contiene ninguna entidad propia. Los tipos de entidad abstractos en UML se indican poniendo el nombre del tipo de entidad en letra cursiva.

Si en una generalización/especialización no se indican las restricciones, por defecto se asume que se trata de una relación definida por el usuario, solapada y parcial. La figura 38 muestra la nomenclatura para identificar los dos casos en UML. En la misma relación se indica entre llaves {} los conceptos de total o parcial *(complete/incomplete)* y disjunta o solapada *(disjoint/overlapping)*.

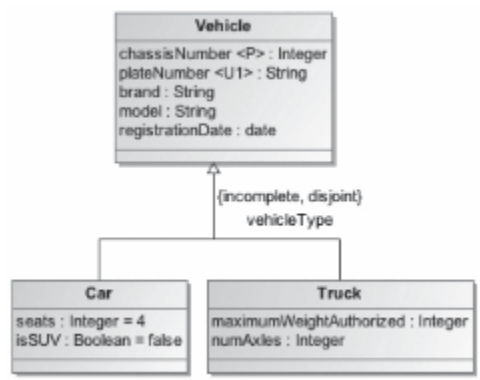
Las figuras 38 y 39 muestran esquemas similares, pero muy diferentes. La base de datos del esquema de la figura 38 permitirá almacenar información de cualquier vehículo, mientras que el esquema de la figura 39 solo permitirá alma-

41. En inglés, *complete*.
42. En inglés, *incomplete*.

cenar información de coches y camiones. A continuación describimos brevemente el porqué.

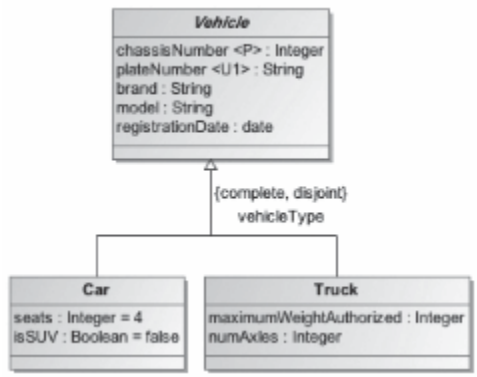
En la figura 38 la generalización/especialización entre la superclase *'Vehículo' (Vehicle)* y las subclases *'Coche' (Car)* y *'Camión' (Truck)* es disjunta e incompleta. Esto nos indica, en primer lugar, que una entidad no puede pertenecer a las dos subclases al mismo tiempo, es decir, que no puede haber un vehículo que sea un coche y un camión a la vez, y, en segundo lugar, que pueden haber entidades del tipo de entidad *'Vehículo'* que no sean ni un coche ni un camión.

Figura 38. Ejemplo de generalización/especialización disjunta *(disjoint)* y parcial *(incomplete)*



En la figura 39, en cambio, se presenta una herencia similar, pero en este caso es completa. El hecho de que sea completa implica que la superclase no podrá contener entidades propias y, por lo tanto, será un tipo de entidad abstracta. Es decir, que cualquier entidad de *'Vehículo'* deberá ser un coche o un camión. En ambos diagramas se indica que el atributo *vehicleType* es el discriminante de la generalización/especialización.

Figura 39. Ejemplo de generalización/especialización disjunta *(disjoint)* y completa *(complete)*



Los conceptos de restricciones de *exclusividad* y de *participación* son independientes entre sí, y, por lo tanto, se pueden combinar para dar lugar a cuatro posibles restricciones de generalización/especialización:

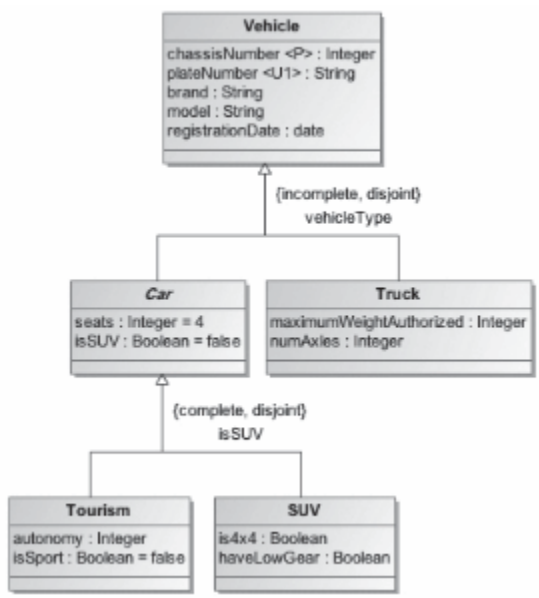
- Disjunta y total
- Disjunta y parcial
- Solapada y total
- Solapada y parcial

1.4. Herencia simple y múltiple

Decimos que un esquema conceptual utiliza **herencia simple** cuando todos sus tipos de entidad tienen, como máximo, una superclase. Como resultado de esta condición, el conjunto de relaciones de generalización/especialización generan una estructura de clases en forma de árbol, normalmente llamado **taxonomía**.

La figura 40 muestra un esquema conceptual que usa herencia simple. El árbol de la jerarquía de clases empieza con la superclase raíz *'Vehículo' (Vehicle)*. El atributo discriminador *'Tipo de vehículo' (vehicleType)* se utiliza para especializar los vehículos en coches *(Car)* o camiones *(Truck)*. Al mismo tiempo, el tipo de entidad *'Coche'* se especializa en dos subclases, según si es un turismo *(Tourism)* o un vehículo todoterreno *(SUV)*. El tipo de entidad *'Coche'* es superclase en esta nueva relación y a la vez subclase en la relación con el tipo de entidad 'vehículo'. Hay que señalar, además, que el tipo de entidad 'coche' se convierte en abstracta, puesto que la especialización es completa. Como en el esquema resultante no hay clases que tengan dos superclases, podemos decir que el esquema presentado es de herencia simple.

Figura 40. Ejemplo de herencia simple



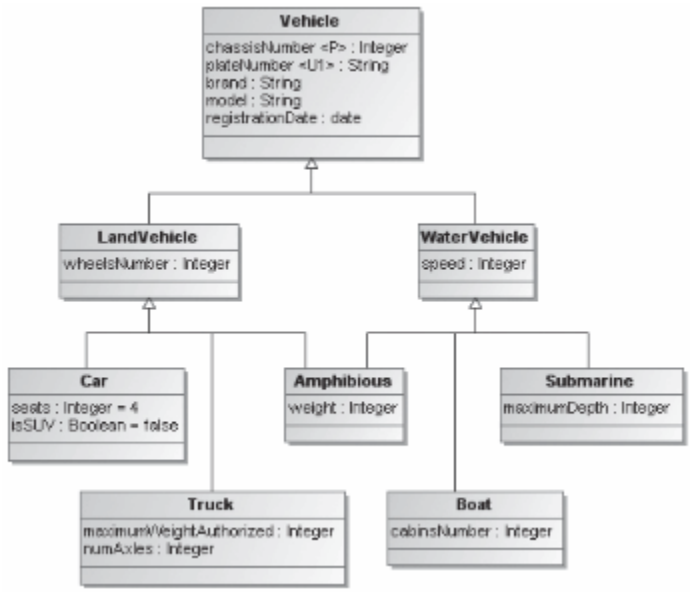
El tipo de entidad origen de la jerarquía, en este caso el tipo de entidad *'Vehículo'*, se llama superclase raíz o tipo de entidad base de la taxonomía. Mientras que los tipos de entidad que no son superclases, en este caso las clases *'Turismo'*, *'Todoterreno'* y *'Camión'*, se denominan clases hoja.

Diremos que un esquema conceptual utiliza herencia múltiple cuando alguno de sus tipos de entidad tiene más de una superclase, es decir, hereda información de más de una clase. En estos casos, se obtiene una estructura de tipo grafo dirigido.

La figura 41 muestra un esquema conceptual que utiliza herencia múltiple. En el esquema, los vehículos se especializan en vehículos terrestres *(LandVehicle)* o acuáticos *(WaterVehicle)*. Los vehículos terrestres se pueden especializar en coches *(Car)*, camiones *(Truck)* o vehículos anfibios *(Amphibious)*. Los vehículos anfibios se pueden desplazar por tierra y por medios acuáticos, y, por lo tanto, *'Anfibio'* también es una especialización del tipo de entidad *'Vehículo acuático'*, junto con los tipos de entidad *'barco' (Boat)* y *'submarino' (Submarine)*.

Como el tipo de entidad *'Anfibio'* tiene dos superclases *(LandVehicle* y *WaterVehicle)*, estamos ante un caso de herencia múltiple. Hay que señalar que si un atributo o relación es heredado más de una vez por caminos diferentes, solo se incluye una vez en la subclase. Es decir, los atributos y las relaciones de la clase *'Vehículo'* solo pueden aparecer una única vez en la subclase compartida.

Figura 41. Ejemplo de herencia múltiple



1.5. Clasificación múltiple

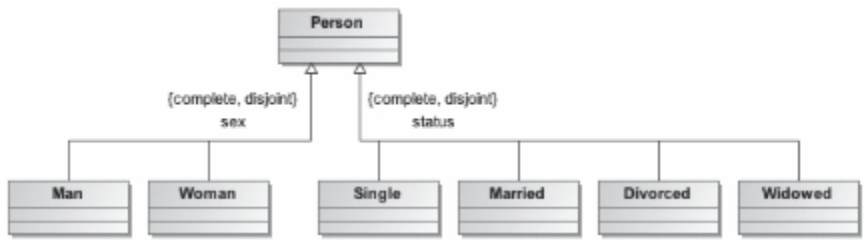
> Un esquema conceptual admite **clasificación múltiple** si una entidad puede ser instancia de más de un tipo de entidad sin que haya ninguna relación de generalización/especialización que conecte los distintos tipos de entidad.

Las relaciones de generalización/especialización solapadas permiten clasificación múltiple, puesto que una entidad puede pertenecer a diferentes subclases a la vez. La figura 37 muestra un ejemplo de generalización/especialización solapada que permite clasificación múltiple. Otro caso de clasificación múltiple lo encontramos cuando una superclase

tiene dos o más jerarquías de especialización según diferentes discriminantes. Cada entidad puede pertenecer a diferentes subclases a la vez.

Partiendo del tipo de entidad *'Persona' (Person)* podemos establecer un proceso de especialización para definir su estado civil, que indique si la persona está soltera *(Single)*, casada *(Married)*, divorciada *(Divorced)* o viuda *(Widowed)*. Por otro lado, una persona también se puede especializar, según el sexo, en *'Hombre' (Man)* o *'Mujer' (Woman)*. La figura 42 muestra el modelo conceptual descrito. Este esquema deberá utilizar clasificación múltiple para representar a las personas, ya que toda persona será instancia de *Hombre* y *Mujer* por un lado y de *Soltero*, *Casado*, *Divorciado* y *Viudo* por otro.

Figura 42. Ejemplo de generalización/especialización con clasificación múltiple



2. Agregación y composición

Los tipos de relación son elementos de modelado genéricos que permiten representar cualquier relación entre entidades

del dominio. Estas relaciones pueden representar cosas muy distintas: dependencia, pertenencia, identificación, ubicación geográfica, interés, etc. No obstante, hay un tipo de relaciones que es especialmente relevante en modelado conceptual y que por ese motivo diversos lenguajes proporcionan estructuras de modelado específicas para representarlas: las relaciones de meronimia/holonimia (o relaciones de parte-todo[43]).

En el modelado conceptual, las relaciones de meronimia permiten representar que una o más entidades forman parte de otra entidad, como por ejemplo que los dedos pertenecen a la mano o que una rueda pertenece a una bicicleta. UML dispone de dos construcciones para modelar este tipo de relaciones: las agregaciones y las composiciones. A continuación vamos a estudiar estas construcciones más a fondo.

2.1. Agregación

> La **agregación** es un tipo de relación binaria que permite representar que las instancias de un tipo de entidad (partes) son parte de las instancias de otro tipo de entidad (todo).

La parte que representa el «todo» recibe el nombre de compuesto y las diferentes «partes» que lo componen reciben el nombre de componentes. En UML la agregación se representa como una asociación pero con un diamante de color blanco en el tipo de entidad compuesto.

La figura 43 muestra un ejemplo de agregación entre un plato *(Dish)* y los ingredientes con los que se ha preparado

43. Relaciones de *partOf* en inglés.

(Ingredient). Como se puede ver en el modelo conceptual, un plato debe estar compuesto por uno o más ingredientes, mientras que un mismo ingrediente puede estar incluido en cualquier número de platos.

Figura 43. Ejemplo de agregación

En una agregación la pertenencia entre las partes y el todo no es exclusiva. Es decir, una parte puede pertenecer a más de un compuesto. Por ejemplo, el ingrediente *'Vino'* (entidad de *'Ingredient'*) podría formar parte de más de una receta.

La distinción entre tipos de relación y agregación es sutil, y a menudo subjetiva. En general, se considera que es necesario que haya una cierta relación de ensamblaje, sea física o lógica, entre las entidades para hablar de *agregación* en lugar de *tipo de relación*. La única restricción que añade la agregación respecto al tipo de relación es que las cadenas de agregaciones entre entidades no pueden formar ciclos.

2.2. Composición

La **composición** es un tipo concreto de agregación donde la existencia de los componentes está vinculada a la existencia del compuesto y la pertenencia de los componentes respecto al compuesto es exclusiva.

Por tanto, la composición impone dos restricciones respecto a la agregación:

- Cada componente puede estar presente en un único compuesto (la multiplicidad del tipo de entidad compuesta debe ser como máximo 1).
- Si se elimina el elemento compuesto, hay que eliminar todos los componentes vinculados a este elemento (los componentes presentan una dependencia de existencia hacia el elemento compuesto).

En UML la composición se representa de la misma forma que la agregación pero con un diamante de color negro.

Como ejemplo podemos pensar en el conjunto de dedos que forman una mano. La figura 44 muestra el modelo conceptual en el que se representa la composición de una mano (tipo de entidad *Hand*) como conjunto de dedos (tipos de entidad *Finger*) y donde cada dedo solo puede pertenecer a una sola mano.

Figura 44. Ejemplo de composición

3. Restricciones de integridad

Las **restricciones de integridad** definen condiciones que se deben cumplir para garantizar que la instanciación de un esquema conceptual es correcta.

A lo largo del libro se han ido viendo algunas restricciones de integridad, a continuación explicaremos en más detalle las

restricciones de integridad más relevantes en el proceso de diseño conceptual de bases de datos.

3.1. Restricciones en los tipos de entidad

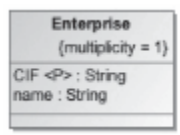
Al igual que en los atributos y los tipos de relación, también es posible indicar la multiplicidad de los tipos de entidad.

La multiplicidad de un tipo de entidad establece un rango de posibles cardinalidades para las entidades para un determinado tipo de entidad. Es decir, indica cuántas instancias de un determinado tipo de entidad puede haber en la base de datos.

Generalmente, y por defecto, la multiplicidad de un tipo de entidad es indefinida, pero algunas veces puede ser interesante poder limitar el número de instancias de un tipo de entidad.

Por ejemplo, supongamos que queremos modelar una base de datos para gestionar pequeñas o medianas empresas. Además de toda la información que sea necesaria (clientes, proveedores, productos, etc.) hay que almacenar los datos de la empresa (CIF, nombre, dirección, etc.). Una manera de modelar este requisito es crear un tipo de entidad que permita almacenar esta información. Como solo va a haber una empresa, se debe especificar una multiplicidad igual a 1 para el tipo de entidad. La figura 45 muestra el diagrama UML de este tipo de entidad.

Figura 45. Tipos de entidad con multiplicidad 1



3.2. Restricciones en los atributos

En la definición de los atributos hemos visto algunas restricciones básicas asociadas a los atributos. Estas restricciones son las siguientes:

- **Restricciones de dominio**: son las restricciones asociadas al conjunto de posibles valores legales que puede tomar un atributo. En este sentido, se puede especificar el tipo de datos que utilizará el atributo (cadena de texto, entero, real, booleano, etc.) y las restricciones adicionales específicas para cada tipo de datos (por ejemplo, la longitud en el caso de cadenas de texto o un intervalo válido en el caso de valores enteros).
- **Atributos derivados**: son atributos que calculan su valor a partir de otros atributos y que, por lo tanto, implícitamente son atributos de solo lectura desde el punto de vista del usuario o de la aplicación. No obstante, su valor puede cambiar si cambia el valor de los atributos de los que dependen.

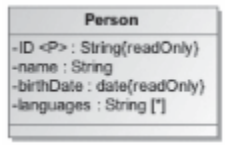
Otro tipo de restricción útil es la restricción de *'solo lectura'*, que permite indicar que los valores de un atributo no pueden cambiar una vez asignados. UML tiene distintas palabras reservadas que se pueden utilizar en la definición de un atributo para representar si un atributo puede cambiar o no. En particular hay cuatro valores permitidos:

- *'Cambiable'* (*changeable* o *unrestricted*): es el valor por defecto. Indica que se permite cualquier cambio sobre el atributo.

- *'Congelado'* (*frozen* o *readOnly*): indica que una vez que se ha asignado valor al atributo, no se podrá modificar ni eliminar.
- *'Solo-añadir' (addOnly)*: indica que una vez que se ha asignado valor al atributo, no se podrá modificar ni eliminar. Pero, en caso de ser un atributo multivalor, sí que se podrán asignar nuevos valores para este atributo.
- *'Solo-eliminar' (removeOnly)*: indica que la única operación permitida es eliminar el valor del atributo.

La figura 46 muestra el tipo de entidad *Person*, donde podemos ver dos atributos que no permiten modificación (*ID* y *birthDate*).

Figura 46. Ejemplo de restricciones de cambiabilidad en los atributos



3.3. Restricciones en los tipos de relación

A continuación presentamos otras restricciones de integridad que se pueden utilizar en las asociaciones para permitir una mayor expresividad del esquema conceptual.

3.3.1. Restricciones de cardinalidad *mín..máx*

Además de todas las restricciones referentes a la cardinalidad de las relaciones que hemos visto, es posible indicar un valor o rango concreto en la cardinalidad de una relación. Para indicar un rango en UML, se expresa el valor mínimo seguido de dos puntos y el valor máximo (*mín..máx*).

La figura 47 modela una situación en la que un empleado *(Employee)* solo puede pertenecer a un único departamento *(Department)* y cada departamento debe tener entre 10 y 50 empleados.

Figura 47. Ejemplo de restricción de cardinalidad *mín..máx*

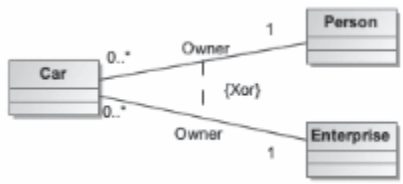
3.3.2. Restricciones *xor*

Esta restricción se utiliza para indicar que las entidades de un tipo de entidad solo pueden participar en uno de un conjunto de tipos de relación predefinido. Por ejemplo, supongamos que un coche puede pertenecer a una persona o a una empresa. En este caso, habría que crear dos tipos de relación para representar la propiedad del coche y se debería añadir la restricción de integridad para indicar que para cada instancia de coche, solo una de las dos relaciones es válida.

Para modelar la situación descrita en el párrafo anterior, la figura 48 muestra los tipos de relación que unen los tipos de entidad *'Coche' (Car)* con los tipos de entidad *'Persona' (Person)* y *'Empresa' (Enterprise)*. Los dos tipos de relación están restrin-

gidos con la restricción *xor*, indicando que un coche ha de ser de una persona o de una empresa, pero no de ambos a la vez.

Figura 48. Ejemplo de restricción *xor*



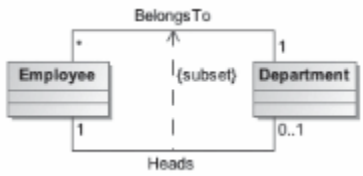
3.3.3. Restricciones *subset*

Esta restricción se utiliza para simular una relación de generalización/especialización entre tipos de relación. La restricción permite indicar que los participantes de las instancias de un tipo de relación son un subconjunto de los participantes de las instancias de otro tipo de relación.

Por ejemplo, supongamos una empresa con empleados y departamentos. Supongamos también que queremos representar los empleados asignados a cada departamento y los directores de departamento. Podemos representarlo tal y como se muestra en la figura 49, donde el tipo de relación *'Heads'* se utiliza para representar a los directores de departamento y el tipo de relación *'BelongsTo'* se emplea para representar a los trabajadores de un departamento. Como el director de un departamento es también un trabajador de este, las entidades que participan en todas las relaciones del tipo de relación *'Heads'* también estarán relacionadas por el tipo de relación *'Belongs To'*. Por tanto, si *'Pere Pi'* es el director del departamento de *'Finanzas'*, entonces *'Pere Pi'* también traba-

ja para el departamento de finanzas y, en consecuencia *'Pere Pi'* deberá estar relacionado con *'Finanzas'* por la relación *'Belongs To'*. De no ser así, querría decir que *'Pere Pi'* no trabaja en el departamento de finanzas y no puede ser su director.

Figura 49. Ejemplo de restricción *subset*



3.3.4. Restricciones *ordered*

Supongamos que queremos representar una receta y los distintos pasos para hacerla. Habrá una relación entre la receta y los pasos para indicar qué pasos tiene cada receta. Pero esta relación debe ser un poco especial porque los pasos de una receta tienen que estar definidos en el orden correcto, ya que sino, no sabremos llevarla a la práctica efectivamente. En este caso, sería interesante poder definir una restricción que permita decir que los pasos de las recetas deben guardarse de forma ordenada (primero el primer paso, luego el segundo, etc.).

Para resolver el problema anterior, se puede utilizar una restricción de integridad que permita especificar que las entidades de un tipo de entidad se relacionan con las de otro tipo de entidad siguiendo un orden determinado.

En UML esta restricción no se aplica a un tipo de relación, sino a sus extremos y solo es válida cuando la cardinalidad máxima del extremo donde se define es superior a 1.

En la figura 50 podemos ver un esquema UML que representa el ejemplo comentado. Para indicar la restricción de orden hemos puesto la palabra *{ordered}*, en el extremo del tipo de relación afectado.

Figura 50. Ejemplo de restricción *ordered*

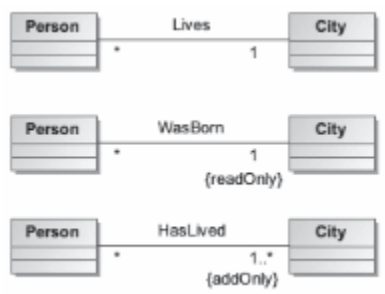
3.3.5. Restricciones de cambiabilidad

Al igual que en los atributos, se puede indicar si los valores del extremo de una relación pueden cambiar o no. En UML, esta restricción se representa mediante el uso de las palabras reservadas antes comentadas *(frozen, readOnly, addOnly, removeOnly)* en el extremo del tipo de relación que se quiera restringir.

Para poder comparar diferentes ejemplos relacionados con la cambiabilidad, la figura 51 muestra tres situaciones diferentes en las que intervienen los mismos tipos de entidad. En la primera, representamos la ciudad donde vive una persona, que evidentemente puede cambiar. Por lo tanto, el extremo de la asociación que conecta con el tipo de entidad *'ciudad' (City)* es cambiable o no restringida (opción por defecto). En la segunda situación, representamos el lugar de nacimiento de una persona, y en este caso etiquetamos como congelado el extremo de esta asociación ya que una persona no puede cambiar el lugar donde nació. Finalmente, si consideramos las ciudades donde ha vivido una persona, etiquetamos con el atributo *'addOnly'*, puesto que la lista de ciudades donde

ha vivido una persona puede aumentar, pero no modificar o eliminar valores existentes.

Figura 51. Ejemplos de restricciones de cambiabilidad en los tipos de relación

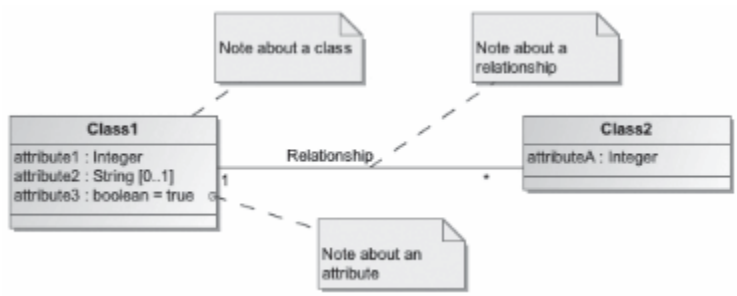


3.4. Otras restricciones

En el análisis de requisitos podemos encontrar restricciones de integridad muy difíciles o imposibles de representar mediante las estructuras predefinidas del lenguaje de modelado que usemos. Generalmente, para representar estas restricciones se utilizan notas ligadas a los tipos de entidad, atributos o tipos de relación a los que hacen referencia.

La figura 52 muestra tres notas que permiten notaciones de cualquier tipo para clarificar conceptos del modelo, asociadas a un tipo de entidad, un atributo o un tipo de relación.

Figura 52. Ejemplo de uso de notas para indicar requisitos o restricciones del modelo



4. Modelado de datos históricos

Generalmente las bases de datos representan el estado del dominio de interés en un momento determinado del tiempo. Cuando se produce un cambio de estado en el mundo real, la base de datos se actualiza y se pierde la información anterior. No obstante, para algunas aplicaciones es importante tener información histórica con el objetivo de ver y analizar cómo han evolucionado las cosas.

Cuando se quiera representar información histórica, se modelará el tiempo asociado a los conceptos de interés mediante instantes o intervalos de tiempo. En función de los requisitos del sistema, las necesidades de información temporal serán diferentes y escogeremos uno u otro.

En caso de querer relacionar una entidad o una relación con un instante de tiempo, se podrá hacer de dos formas:

- Añadiendo un atributo de tipo 'fecha y hora' en el tipo de entidad o tipo de relación que contenga el elemento de interés.
- Añadiendo una relación con el tipo de entidad 'fecha' *(Date)*, que contiene un atributo de tipo 'fecha y hora'.

Un ejemplo en el que es relevante modelar/representar información temporal mediante instantes de tiempo es el siguiente. Supongamos un sistema de monitorización de una fábrica que va tomando lecturas continuamente. Nos interesa guardar el valor de la lectura y el momento (tiempo) en el que se ha producido esta lectura. En este caso, lo haríamos mediante un instante de tiempo porque necesitamos almacenar las lecturas y cuándo se han hecho, no intervalos de tiempo.

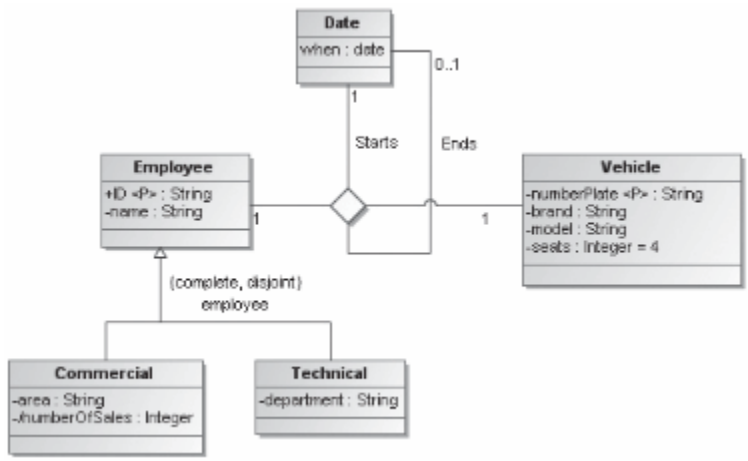
En caso de querer relacionar una entidad o una relación con un intervalo de tiempo, se podrá hacer de dos formas:

- En caso de tratarse de un tipo de entidad o un tipo de entidad asociativo, se pueden añadir dos atributos de tipo fecha y hora en el tipo de entidad para indicar el inicio y el final del intervalo.
- En caso de tratarse de un tipo de relación, hay que añadir un nuevo participante en el tipo de relación. El nuevo participante será una entidad *'Fecha' (Date)* que determinará los tiempos inicial y final del intervalo.

Por ejemplo, supongamos que una empresa tiene un conjunto de vehículos y un conjunto de comerciales y personal técnico que los utiliza. Según las tareas y las visitas de cada día, cada uno de los comerciales y los técnicos puede nece-

sitar un vehículo diferente. Por lo tanto, a esta empresa le interesará saber qué coche ha estado utilizando cada uno de los comerciales y técnicos en el pasado (por ejemplo, en caso de recibir una sanción de tráfico). Como una persona no usa el vehículo en un momento concreto del tiempo, sino que lo utiliza en un intervalo, deberemos utilizar intervalos de tiempo para modelar la información temporal. Para ello, crearemos un tipo de relación cuaternaria en la que participan los tipos de entidad *'Empleado', 'Vehículo', 'Fecha'* y *'Fecha'*. Dos de los participantes de la relación son 'Fecha' para poder representar el inicio y el fin del intervalo (desde que empieza a usar el vehículo hasta que lo devuelve). La figura 53 muestra un modelo que implementa la situación descrita.

Figura 53. Ejemplos de situación con modelado de datos históricos



La tabla 2 muestra, a modo de ejemplo, una posible configuración de entidades del ejemplo anterior en un instante de tiempo concreto.

ID	name	numberPlate	starts	ends
33941857B	John	8754-GFD	2013-05-12 09:25	2013-05-12 17:36
33941857B	John	1258-CGC	2013-05-17 11:13	--
15488574Q	Mary	8754-GFD	2013-05-10 15:11	2013-05-10 19:47
15488574Q	Mary	1258-CGC	2013-05-13 07:02	2013-05-14 14:41
15488574Q	Mary	1126-BMR	2013-05-16 20:14	--
25486257F	Fred	1126-BMR	2013-05-11 09:13	2013-05-11 18:56

5. Ejemplo completo

A continuación planteamos un ejemplo completo para poder ver un esquema conceptual en el que aparezcan algunos de los elementos vistos en este apartado. Continuaremos con el ejemplo del apartado anterior, que se basa en el diseño de una base de datos para la gestión universitaria.

A continuación enumeramos los diferentes aspectos de los requisitos de los usuarios que hay que tener en cuenta al realizar el diseño conceptual de la base de datos:

1. La universidad está formada por diferentes departamentos. Cada departamento tiene asignados un conjunto de profesores y está dirigido por un único profesor. Nos interesa conocer el nombre de los departamentos y el número de profesores que trabajan en cada uno.
2. Para cada profesor, nos interesa poder almacenar sus datos personales, como, por ejemplo, su nombre, DNI, fecha de nacimiento, sexo, dirección y números

de teléfono. Para los profesores que son directores de departamento, nos interesa poder identificar la fecha en la que empezaron a ejercer este cargo.

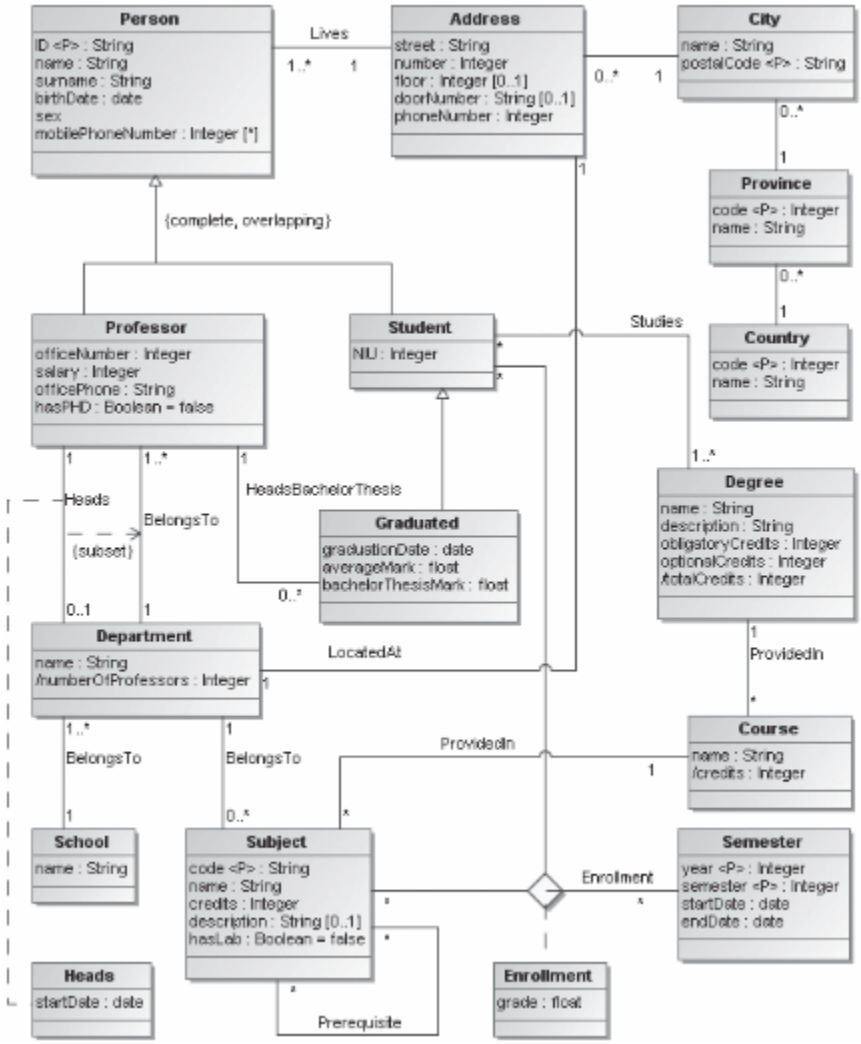
3. En la universidad se imparte un conjunto de asignaturas. Interesa saber, para cada una de estas asignaturas, el código, el nombre, el número de créditos, la descripción, si tiene laboratorio asociado y sus prerrequisitos. Además, hay que tener en cuenta que una asignatura pertenece a un único departamento.
4. Los departamentos están agrupados en escuelas o facultades. Por ejemplo, el departamento de matemáticas pertenece a la facultad de ciencias. Un departamento pertenece a una única escuela o facultad.
5. Los estudiantes son una parte muy importante de la base de datos. Se quiere almacenar información sobre los datos personales de estos estudiantes (nombre, DNI, fecha de nacimiento, sexo, dirección y números de teléfono) y su número de identificación universitaria (conocido como NIU).
6. Cada estudiante puede estar matriculado de varias asignaturas en cada semestre. Y para cada una de las asignaturas en las que está matriculado cada semestre, debemos almacenar una nota final.
7. También se quiere dejar constancia de las fechas de inicio y final de cada semestre.
8. Las asignaturas se agrupan en los diferentes cursos que forman cada uno de los grados que ofrece la universidad. Por ejemplo, la asignatura de Álgebra pertenece al primer curso del grado de Matemáticas. Sobre cada uno de los cursos solo nos interesa almacenar el conjunto de asignaturas que lo forman. Sobre cada uno

de los grados, nos interesa almacenar información referente a su número de créditos (obligatorios, opcionales y totales) y una descripción.

9. Los estudiantes estudian uno o más grados.
10. Cuando un estudiante se gradúa, elabora un proyecto de final de carrera sobre el que nos interesa almacenar información acerca de la fecha en la que se ha presentado, la nota que ha obtenido y el profesor que lo ha dirigido.

A partir de los requisitos expresados, la figura 54 muestra un diagrama del modelo conceptual. Como hemos comentado, este modelo no es único, sino que puede haber diversas aproximaciones e interpretaciones válidas para una misma representación del mundo real.

Figura 54. Diagrama del modelo conceptual para la segunda aproximación de la base de datos de gestión universitaria



Resumen

En el primer capítulo hemos visto, de una manera muy general, el proceso de diseño de una base de datos. Hemos introducido, aunque muy brevemente, las distintas etapas que forman el proceso de diseño de una base de datos. El proceso de diseño de una base de datos se inicia con la fase de recogida y análisis de los requisitos, lo cual permite recoger y centralizar las necesidades de los diferentes grupos de usuarios y aplicaciones. A partir de este análisis, en la segunda fase, se modela un esquema conceptual que permite describir el modelo de datos de una manera independiente de la tecnología. La etapa siguiente en este proceso es el diseño lógico, que requiere haber elegido previamente el tipo de base de datos que se quiere utilizar en la implementación del sistema de información. El tipo de base de datos determina el modelo lógico que va a desarrollarse. Por ejemplo, en el caso de utilizar un tipo de base de datos relacional, se transforma el modelo conceptual en un modelo lógico específico para bases de datos relacionales denominado modelo relacional. El diseño físico permitirá adaptar el modelo lógico a un sistema gestor de bases de datos (SGBD) concreto. Por lo tanto, previamente a este paso se deberá escoger el SGBD específico que se quiere utilizar para implementar el sistema de información. En esta etapa se crea la estructura física que almacenará los datos de la base de datos. Finalmente, la última etapa

permite la optimización de la base de datos y la gestión de la seguridad relacionada con los usuarios y las aplicaciones de la base de datos. Lógicamente, habrá que realizar la carga de los datos previamente, puesto que el tamaño de las tablas, los tipos de consultas y las frecuencias de estas son elementos importantes para optimizar el acceso a los datos por parte del sistema gestor.

En el segundo capítulo hemos visto el proceso de diseño conceptual. Este proceso es una de las etapas del diseño de bases de datos, concretamente, es la segunda etapa, y se realiza después del análisis de requisitos. El diseño conceptual permite crear un esquema conceptual de alto nivel e independiente de la tecnología de implementación a partir de las especificaciones y los requisitos de un problema del mundo real. El enfoque de este material nos ha permitido ver las bases del diseño conceptual empleando los diagramas UML como sistema de notación. En la introducción, correspondiente a la primera parte de este material, hemos detallado las bases, los objetivos y los requisitos del diseño conceptual y de los diagramas en lenguaje UML. En la segunda parte, hemos descrito los elementos básicos de modelado en el diseño conceptual. Entre los elementos principales hay que destacar los tipos de entidad, los atributos y los tipos de relación. Además de estos tres elementos, que forman la base principal del modelo conceptual, también hemos visto los tipos de entidades asociativas y los tipos de entidades débiles, que permiten una mayor riqueza en la representación del modelo conceptual. Finalmente, en la tercera parte de este material, hemos visto algunos de los elementos avanzados en el diseño conceptual. Los elementos descritos en la parte anterior no permiten representar fácilmente situaciones que encontra-

mos en el mundo real y que queremos poder representar en nuestro modelo. Fruto de esta necesidad se extiende el modelo para incluir conceptos como la generalización/especialización, la agregación o la composición, que permiten una mayor riqueza en la representación del modelo conceptual. Para finalizar este texto, nos hemos referido brevemente a las restricciones de integridad básicas y al modelado de datos históricos.

Glosario

atributo m. Propiedad que interesa representar de un tipo de entidad.

clase f. Nombre que reciben los tipos de entidad en el modelo UML.

tipo de entidad asociativa m. Tipo de entidad resultante de considerar una relación entre entidades como una nueva entidad.

tipo de entidad débil m. Tipo de entidad cuyos atributos no la identifican completamente, sino que solo la identifican de manera parcial.

conectividad f. Expresión del tipo de correspondencia entre las entidades de una relación.

diseño conceptual m. Etapa del diseño de una base de datos que obtiene una estructura de la información de la futura base de datos independiente de la tecnología que se quiere emplear.

diseño lógico m. Etapa del diseño de una base de datos que parte del resultado del diseño conceptual y lo transforma de manera que se adapte al modelo de sistema gestor de bases de datos con el cual se desea implementar la base de datos.

generalización/especialización f. Construcción que permite reflejar que existe un tipo de entidad general, llamada superclase, que se puede especializar en diferentes tipos de entidad más específicas, llamadas subclases.

grado de una relación m. Número de entidades que asocia la relación.

entidad f. Objeto del mundo real que podemos distinguir del resto de objetos y del cual nos interesan algunas propiedades.

tipo de relación m. Asociación entre entidades.

tipo de relación recursiva m. Asociación a la cual alguna entidad está asociada más de una vez.

modelo entidad-interrelación m. Modelo de datos de alto nivel ampliamente utilizado para el diseño conceptual de las aplicaciones de bases de datos. El objetivo principal del modelo ER es permitir a los diseñadores reflejar en un modelo conceptual los requisitos del mundo real. En inglés *entity-relationship model*

sistema gestor de bases de datos m. Tipo de software específico dedicado a servir de interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan. Sigla SGBD. En inglés *database management system* (DBMS)

lenguaje unificado de modelado m. Lenguaje gráfico para modelar, visualizar, especificar, construir y documentar sistemas de software o de bases de datos. Sigla UML. En inglés *unified modeling language* (UML)

Bibliografía

Costal Costa, D. *Diseño de bases de datos*. Barcelona: Ediciones UOC, 2005.

Elmasri, R., Navathe, S. B. *Fundamentos de sistemas de bases de datos*. Madrid: Pearson Educación, 2007.

Date, C. J. *Introducción a los sistemas de bases de datos*. Madrid: Pearson Educación, 2001.

Larman, C. *UML y patrones. Introducción al análisis y diseño orientado a objetos*. México: Prentice Hall, 1999.

Olivé, A. *Conceptual Modeling of Information Systems*. Springer, 2007.

Ramakrishnan, R, Gehrke, J. *Database management systems*. Boston: Mc-Graw-Hill Higher Education, 2002.

Rumbaugh, J., Jacobson, I., Booch, G. *El lenguaje unificado de modelado. Manual de referencia*. Madrid: Pearson Educación, 2007.

Teorey, T. J. *Database design: Know it all*. Burlington: Morgan Kaufmann, 2008.