PROYECTO INTERMODULAR
ENTREGA 2. FASE DE DISEÑO

Tabla de contenido

1. Introducción y documentos a entregar

En la fase de diseño del Proyecto Intermodular el objetivo es presentar el diseño completo del proyecto. La documentación debe ser clara, estructurada y demostrar la planificación técnica y visual de la aplicación.

Los documentos a entregar son los siguientes:

1. **Diagramas de casos de uso.** Los diagramas de casos de uso describen las interacciones entre los usuarios (actores) y el sistema. En términos generales, se deberá entregar un diagrama por cada actor principal (cliente, administrador…), cubriendo todos los requisitos funcionales definidos en la propuesta inicial. Opcionalmente, si varios actores comparten casos de uso, se puede entregar un solo caso de uso general.

2. **Diagrama de clases.** Muestran la estructura estática del sistema, incluyendo las clases, sus atributos, métodos y las relaciones entre ellas. Se deberá entregar un solo diagrama de clases, que incluirá todas las clases principales y sus relaciones.

3. **Diagrama de secuencia.** Describen el orden de los mensajes intercambiados por objetos en un escenario particular. Se deberán entregar al menos 3 diagramas de las secuencias principales y representativas de la aplicación, evitando las secuencias más comunes, como login, registro en el sistema, cierre de sesión, etc.

4. **Diseño de la Base de Datos.** El objetivo es definir la estructura lógica de la base de datos, las relaciones entre sus componentes y proporcionar el código necesario para su creación e implementación. El diseño de la base de datos incluye 3 aspectos:

    4.1. **Diagrama Entidad-Relación (E/R).** Modelo conceptual que representa las entidades, sus atributos y las relaciones que existen entre ellas. Este diagrama es independiente del SGBD específico (MySQL, PostgreSQL…).

    4.2. **Diagrama Relacional (modelo lógico).** En el paso posterior al E/R, éste se transforma en una estructura más cercana a la implementación real en una base de datos relacional. Se especifican las tablas, las claves primarias y las claves foráneas que implementan las relaciones definidas en el E/R. Es obligatorio que el modelo lógico esté normalizado, al menos, hasta la tercera forma normal.

    4.3. **Script de Implementación (SQL).** El script de implementación es el código SQL necesario para crear la base de datos. Demuestra la capacidad técnica para trasladar el diseño lógico a un sistema funcional. El script estará compuesto por las sentencias SQL ejecutables en el gestor elegido.

5. **Mockups de la interfaz.** Un mockup es una representación visual estática de la interfaz de usuario de la aplicación. Son cruciales en la fase de diseño para visualizar el *look & feel* de la aplicación y validar la experiencia del usuario antes de escribir código. Se deberán entregar tantos mockups como pantallas o interfaces representativas de la aplicación. Para aplicaciones web se deberán entregar para cada mockup en versión de escritorio (pantallas grandes) una versión de la misma interfaz en versión móvil (pantallas pequeñas).

2. Diagramas de casos de uso

Los casos de uso forman parte de la **notación UML** (*Unified Modeling Language*), el estándar más utilizado para el modelado visual de sistemas software.

UML proporciona un conjunto de diagramas que permite representar de manera gráfica las interacciones entre los actores y el sistema, así como las relaciones entre los distintos casos de uso. UML facilita la comprensión, comunicación y documentación de los requisitos y el diseño del sistema entre todos los participantes del proyecto.

Los **casos de uso** son una técnica de modelado utilizada en la fase de análisis y diseño de software para describir cómo los usuarios (actores) interactúan con el sistema. Un caso de uso representa una secuencia de acciones que proporciona un resultado de valor para un actor específico.

Los casos de uso se emplean para:

- Identificar y documentar los requisitos funcionales del sistema.
- Servir de base para la elaboración de pruebas y validación del sistema.
- Guiar el diseño de la arquitectura y la implementación.

2.1. Elementos principales de los casos de uso

Existen dos elementos principales en un diagrama de casos de uso:

- **Actores.** Un actor es cualquier entidad externa que interactúa con el sistema, que puede ser una persona, otro sistema o un dispositivo. Un actor representa un rol, no una persona específica (una misma persona puede desempeñar varios roles y, por tanto, ser varios actores diferentes en el sistema). Los actores se representan como figuras de “stickman” en los diagramas.

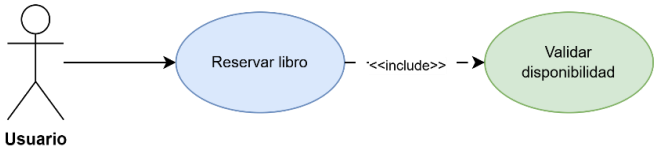
- **Casos de uso.** Un caso de uso representa una funcionalidad o servicio que el sistema ofrece a los actores. Algunos aspectos importantes a tener en cuenta a la hora de diseñar un caso de uso son los siguientes:
  - Se representan con una elipse con el nombre dentro.
  - Debe tener un nombre claro y descriptivo (verbo en infinitivo + objeto).
  - Debe representar una funcionalidad completa y con valor para el actor.
  - Debe ser independiente del orden de ejecución.
  - Debe tener un inicio y fin claros, evitando ambigüedades.

2.2. Relaciones entre casos de uso

2.2.1. «include»

La relación *«include»* indica que un caso de uso **siempre** incorpora el comportamiento de otro caso de uso. Se utiliza para reutilizar una funcionalidad común y se representa con una línea discontinua y una flecha dirigida desde el caso de uso principal al caso de uso incluido.

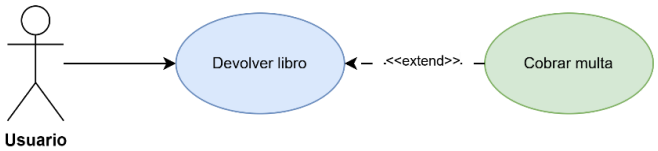
Por ejemplo, supongamos un sistema de biblioteca en el que el caso de uso "Reservar libro" siempre requiere verificar la disponibilidad del libro. En este caso, "Reservar libro" incluye el caso de uso "Validar disponibilidad":



2.2.2. «extend»

La relación *«extend»* indica que un caso de uso puede extender el comportamiento de otro bajo ciertas condiciones. Es útil para modelar variaciones o **funcionalidades opcionales**. Se representa con una línea discontinua y una flecha dirigida desde el caso de uso extendido al caso base.

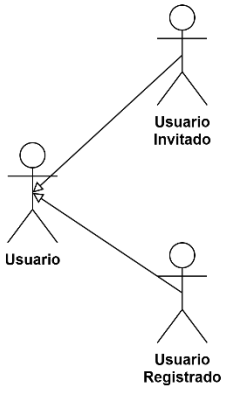
Siguiendo el ejemplo del mismo sistema de biblioteca, el caso de uso "Devolver libro" puede extenderse con "Cobrar multa" si la devolución es tardía:



2.2.3. Generalización

Los actores y casos de uso pueden tener relaciones de herencia (generalización), representadas por una línea continua con una flecha abierta.

Por ejemplo, supongamos que hay dos tipos de usuarios en el sistema: "Usuario registrado" y "Usuario invitado" y ambos heredan de un actor general "Usuario".



La generalización en UML tiene implicaciones importantes tanto en el resto del diseño del sistema como en la implementación final. En el **diagrama de clases**, la generalización en actores normalmente implica tener una clase base y varias subclases que heredan atributos y métodos. Pasa igual en el **diagrama E/R**, donde la generalización se modelaría como una entidad Usuario y dos subtipos: UsuarioRegistrado y UsuarioInvitado.

2.3. Ejemplo: Bookify

Para ejemplificar el diseño de casos de uso vamos a utilizar una plataforma ficticia llamada **Bookify**, destinada a la gestión de reservas de libros en una librería.

Bookify permitirá a los clientes buscar, reservar y gestionar pedidos de libros, mientras que los administradores pueden gestionar el inventario, las reservas y los usuarios.

2.3.1. Actores

Dado que el sistema permitirá tener usuarios visitantes y usuarios registrados, vamos a modelar esta relación mediante generalización.

- **Usuario.** Actor general que representa a cualquier persona que interactúa con la plataforma. Tiene acceso a funcionalidades básicas:
  - Navegar por el catálogo público.
  - Buscar libros sin restricciones.
  - Ver detalles de libros.
- **Usuario Invitado.** Usuario que accede a la plataforma sin registrarse ni iniciar sesión. Hereda de Usuario. Podrá hacer lo mismo que el usuario general.
- **Usuario Registrado.** Usuario que tiene una cuenta en el sistema y ha iniciado sesión. Hereda de Usuario. Tiene acceso a las funcionalidades básicas y, además:
  - Gestionar su cuenta personal.
  - Realizar y gestionar reservas.
  - Actualizar información personal.
  - Consultar historial de actividades.
- **Administrador.** Usuario con permisos administrativos completos. Tiene acceso total al sistema, incluyendo panel administrativo:
  - Gestionar inventario y catálogo.
  - Supervisar reservas y pedidos.
  - Administrar usuarios del sistema.
  - Generar reportes y estadísticas.

Los casos de uso para este ejemplo se muestran a continuación:

Las relaciones indicadas entre Casos de Uso son las siguientes:

- CU12 (Confirmar reserva) incluye a CU11 (Ver todas las reservas) porque el administrador debe visualizar las reservas para confirmar una específica.
- CU13 (Procesar entrega) incluye a CU11 (Ver todas las reservas) porque el administrador debe consultar las reservas para procesar una entrega.
- CU07 (Cancelar reserva) incluye a CU05 (Reservar libro) porque solo se puede cancelar una reserva existente.
- CU03 (Buscar libros) se extiende con CU16 (Buscar con filtros avanzados) porque no es obligatorio usarlos para usar una búsqueda básica.
- CU04 (Ver detalles de libro) se extiende con CU17 (Crear lista de deseos) porque añadir a la lista de deseos es opcional.

3. Diagramas de clases

Un **diagrama de clases** es una representación gráfica que muestra la estructura de un sistema orientado a objetos. Utiliza una notación estándar (UML, *Lenguaje Unificado de Modelado*) para describir las clases, sus atributos, métodos y las relaciones entre ellas. Es una herramienta fundamental en la fase de diseño de software, ya que permite visualizar cómo se organizarán y relacionarán los distintos componentes del sistema antes de comenzar la programación.

Durante la fase de diseño, el objetivo es planificar cómo se implementarán los requisitos del sistema. El diagrama de clases ayuda a:

- Visualizar la estructura del sistema: Permite ver de un vistazo las entidades principales y cómo interactúan.
- Detectar errores de diseño: Al representar las relaciones y dependencias, es más fácil identificar posibles problemas antes de programar.
- Facilitar la comunicación: Sirve como referencia para todo el equipo de desarrollo, asegurando que todos entienden el diseño de la misma manera.
- Documentar el sistema: Sirve como documentación técnica que puede ser consultada en el futuro para mantenimiento o evolución.

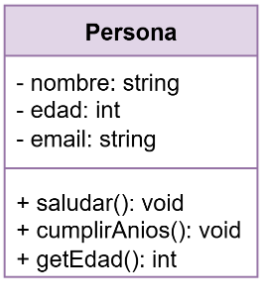
3.1. Elementos principales de un diagrama de clases

Un diagrama de clases está compuesto por varios elementos básicos:

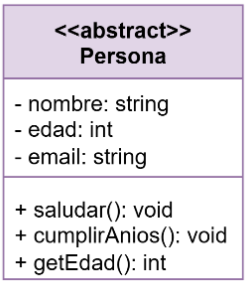
3.1.1. Clases

Las clases representan entidades o conceptos del sistema. Se dibujan como rectángulos divididos en tres secciones:

- **Nombre de la clase** (parte superior): Debe ser un sustantivo en singular y comenzar con mayúscula.
- **Atributos** (parte central): Propiedades o datos que caracterizan a la clase.
- **Métodos** (parte inferior): Acciones o comportamientos que puede realizar la clase.

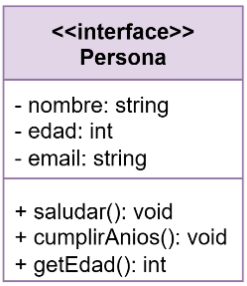


Las clases pueden ser abstractas e interfaces. Una **clase abstracta** es una clase que no puede ser instanciada directamente. Se utiliza como base para otras clases y puede contener métodos abstractos (sin implementación) y métodos concretos. En la representación, el nombre de la clase aparece en cursiva o se añade `<<abstract>>`.



Por ejemplo, podríamos definir la clase Animal como abstracta porque representa un concepto general que no se puede instanciar directamente. Sirve como base para otras clases más específicas, como Perro o Gato, que heredarían de Animal y podrían implementar el método `hacerSonido()` de forma específica para cada caso.

Por su parte, una **interfaz** define un conjunto de métodos que deben implementar las clases que usen la interfaz, pero sin definir cómo deben hacerlo. Se utiliza `<<interface>>` encima del nombre.



Por ejemplo, en Java es muy conocida la interfaz `Comparable`, que obliga a las clases que implementan la interfaz a definir la manera en que se comparan objetos de esas clases para su ordenamiento.

3.2. Relaciones entre clases

Las clases pueden estar relacionadas de varias formas, cada una representando un tipo de vínculo diferente en el sistema. A continuación, se describen las principales relaciones, su notación en UML y ejemplos prácticos:

3.2.1. Asociación

La **asociación** es la relación más básica y común. Indica que una clase está conectada con otra porque una necesita conocer o interactuar con la otra para cumplir su función. La asociación puede ser:

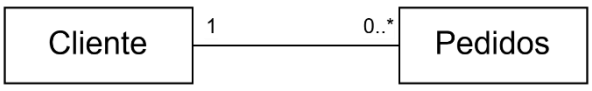
- **Unidireccional**: Solo una de las clases conoce a la otra. Por ejemplo, si Profesor conoce a Alumno, pero no al revés.
- **Bidireccional**: Ambas clases se conocen mutuamente. Por ejemplo, tanto Profesor como Alumno pueden interactuar entre sí.

Las asociaciones pueden tener **multiplicidad**, que indica cuántas instancias de una clase pueden estar relacionadas con una instancia de la otra. Las multiplicidades más comunes son:

- **1**: Exactamente uno.
- **0..1**: Cero o uno (opcional).
- **0..***: Cero o muchos.
- **1..***: Uno o muchos.
- **x..y**: Entre ‘x’ e ‘y’, siendo estos números enteros.
- **x**: Exactamente ‘x’, siendo ‘n’ un número entero.

Ejemplos de multiplicidad:

- Un `Cliente` puede tener **0 o muchos** `Pedidos`: 0..*.
- Un `Pedido` pertenece a **exactamente un** `Cliente`: 1.
- Un `Estudiante` puede estar inscrito en **1 a 5** `Asignaturas`: 1..5.
- Una `Empresa` tiene **al menos un** `Empleado`: 1..*.

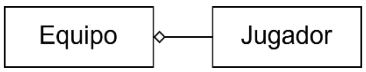


Esta notación se usa también en los diagramas E/R para establecer la cardinalidad de una relación entre dos entidades.

3.2.2. Agregación

Es un tipo especial de asociación que representa una relación "tiene un" o "es parte de", donde una clase (el todo) contiene a otras (las partes), pero las partes pueden existir independientemente del todo.

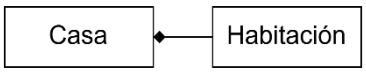
Ejemplo: Un `Equipo` está formado por varios `Jugador`, pero un `Jugador` puede no tener equipo. La **notación UML** es una línea con un rombo blanco en el extremo *todo*.



3.2.3. Composición

Es una forma más fuerte de agregación. Indica que la vida de las partes depende del todo; si el todo se destruye, las partes también.

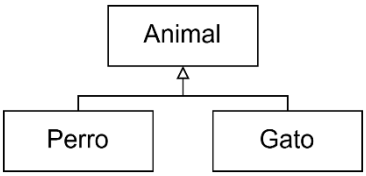
Ejemplo: Una `Casa` está compuesta por varias `Habitacion`. Si se destruye la casa, las habitaciones dejan de existir. La **notación UML** es una línea con un rombo negro en el extremo del *todo*.



3.2.4. Herencia (Generalización)

Permite que una clase hija herede atributos y métodos de una clase padre, promoviendo la reutilización de código y estableciendo una relación “es un”.

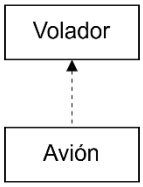
Ejemplo: `Perro` y `Gato` heredan de `Animal` porque ambos “son un” `Animal`. La **notación UML** es una línea con una flecha hueca apuntando a la superclase.



3.2.5. Implementación

Indica que una clase implementa una interfaz, obligándose a proporcionar la implementación de todos los métodos definidos en la interfaz.

Ejemplo: La clase `Avion` implementa la interfaz `Volador`. La **notación UML** es una línea discontinua con flecha hueca apuntando a la interfaz.

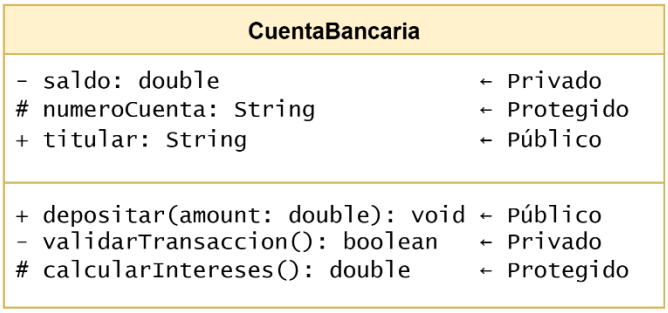


3.3. Visibilidad

Los atributos y métodos pueden tener distintos niveles de visibilidad:

- **Público (+)**: Accesible desde cualquier parte del programa.
- **Privado (-)**: Solo accesible desde la propia clase.
- **Protegido (#)**: Accesible desde la clase y sus subclases (herencia).
- **Paquete (~)**: Accesible desde clases del mismo paquete (menos común).

Por ejemplo:



3.4. Cómo elaborar un diagrama de clases

Algunos pasos clave en la elaboración de un diagrama de clases son estos:

- **Identificar las clases principales**: Analiza los requisitos y extrae los conceptos clave (sustantivos en la descripción del problema).
- **Definir atributos y métodos**: Para cada clase, determina qué datos almacenará y qué acciones realizará.
- **Establecer relaciones**: Dibuja cómo se relacionan las clases entre sí, determinando el tipo correcto de relación y las multiplicidades.
- **Definir visibilidades**: Establece qué elementos son públicos, privados o protegidos.
- **Revisar y refinar**: Asegúrate de que el diagrama sea claro, completo y sin redundancias.

3.5. Ejemplo: Bookify

Para ejemplificar el diseño de diagramas de clases vamos a continuar con el ejemplo de la plataforma **Bookify**, destinada a la gestión de reservas de libros.

3.5.1. Clases del sistema

- `Usuario`: Representa a los usuarios del sistema (clientes y administradores). Gestiona la información personal y las credenciales de acceso.
- `Libro`: Representa los libros disponibles en el catálogo de la librería. Almacena información bibliográfica y de stock.
- `Reserva`: Representa una reserva realizada por un cliente. Gestiona el estado y los detalles de las reservas.
- `Categoria`: Clasifica los libros por géneros o temáticas. Organiza el catálogo para facilitar la búsqueda.
- `ListaDeseos`: Representa una lista de libros de interés de un usuario registrado. Gestiona los libros que el usuario desea reservar en el futuro.

3.5.2. Relaciones entre clases

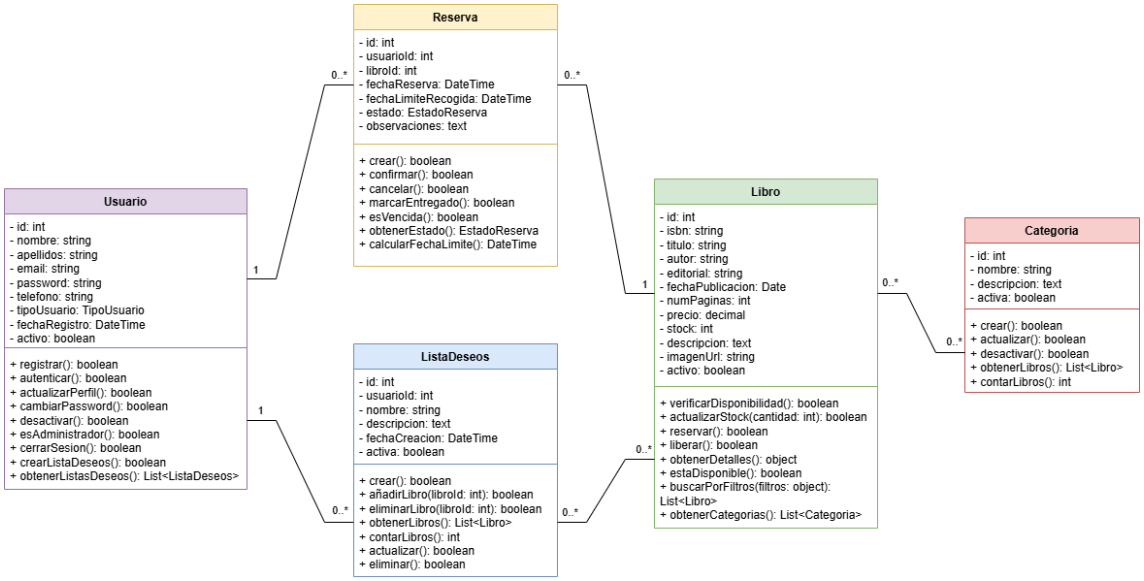
Las relaciones entre las clases son las siguientes:

- `Usuario` ↔ `Reserva`: Un usuario puede tener múltiples reservas.
- `Libro` ↔ `Reserva`: Un libro puede estar en múltiples reservas.
- `Usuario` ↔ `ListaDeseos`: Un usuario puede tener múltiples listas de deseos.
- `Libro` ↔ `ListaDeseos`: Un libro puede estar en múltiples listas de deseos.
- `Categoria` ↔ `Libro`: Un libro puede pertenecer a múltiples categorías y una categoría puede contener múltiples libros.

Además, describimos también las multiplicidades:

- `Usuario` (1) ↔ (0..*) `Reserva`: Un usuario puede tener cero o más reservas.
- `Libro` (1) ↔ (0..*) `Reserva`: Un libro puede estar en cero o más reservas.
- `Usuario` (1) ↔ (0..*) `ListaDeseos`: Un usuario puede tener cero o más listas de deseos.
- `Libro` (0..*) ↔ (0..*) `ListaDeseos`: Relación muchos a muchos entre libros y listas de deseos.
- `Categoria` (0..*) ↔ (0..*) `Libro`: Relación muchos a muchos entre categorías y libros.

Finalmente, el diagrama sería el siguiente:



4. Diagramas de secuencia

Los diagramas de secuencia permiten modelar la dimensión temporal de las interacciones entre los objetos del sistema, mostrando no solo **qué** interacciones ocurren, sino también **cuándo** y **en qué orden** se producen. Complementan los casos de uso y diagramas de clases, proporcionando una vista dinámica del comportamiento del sistema que facilita la comprensión de los flujos de ejecución complejos.

Más formalmente, un **diagrama de secuencia** es una representación gráfica que muestra cómo los objetos interactúan entre sí a lo largo del tiempo mediante el intercambio de mensajes. Cada diagrama de secuencia modela un escenario específico, normalmente derivado de un caso de uso, y muestra la secuencia cronológica de las interacciones necesarias para completar dicha funcionalidad.

Estos diagramas se enfocan en mostrar cómo los objetos colaboran en el tiempo para realizar una funcionalidad específica, siendo especialmente útiles para:

- Muestran el flujo temporal. A diferencia de otros diagramas UML, el tiempo es una dimensión explícita.
- Revelan dependencias. Permiten identificar qué objetos dependen de otros y en qué momento.
- Facilitan la detección de problemas. Ayudan a identificar cuellos de botella, dependencias circulares o flujos innecesariamente complejos.
- Guían la implementación. Proporcionan una hoja de ruta clara para los desarrolladores.

4.1. Elementos principales

4.1.1. Actores y objetos

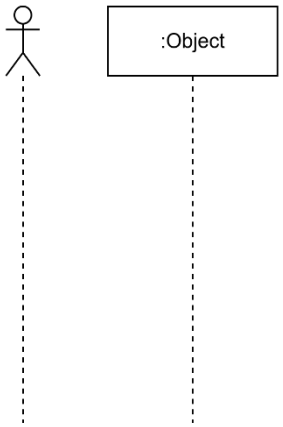
- **Actores.** Representan entidades externas que interactúan con el sistema (usuarios, sistemas externos…). Se dibujan como figuras humanas (*stickman*) o rectángulos con `<<actor>>`.
- **Objetos.** Representan instancias de clases del sistema. Se dibujan como rectángulos con el formato:
  - `nombreObjeto : NombreClase` (objeto específico).
  - `: NombreClase` (objeto anónimo).
  - `NombreClase` (representa la clase en general).

Además, hay una serie de objetos especiales:

- **Boundary objects**: Interfaces del sistema, como pantallas o formularios (`<<boundary>>`).
- **Control objects**: Coordinan la lógica de negocio (`<<control>>`).
- **Entity objects**: Representan datos persistentes (`<<entity>>`).

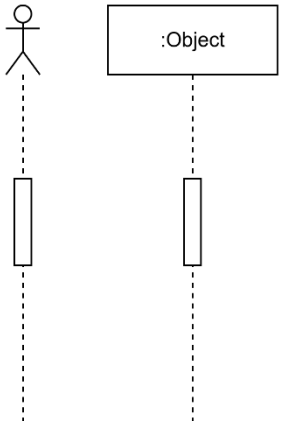
4.1.2. Líneas de vida

Las **líneas de vida** son líneas verticales discontinuas que se extienden hacia abajo desde cada participante, representando la existencia del objeto a lo largo del tiempo. El tiempo fluye de arriba hacia abajo.



4.1.3. Barras de activación

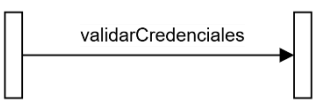
Las barras de activación son rectángulos delgados superpuestos a las líneas de vida que indican cuándo un objeto está "activo" o ejecutando alguna operación.



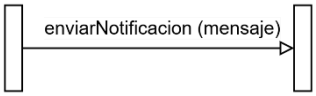
4.1.4. Mensajes

Los mensajes representan comunicaciones entre participantes y se dibujan como flechas horizontales entre las líneas de vida. Existen varios tipos:

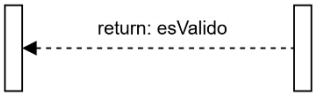
- **Mensajes síncronos.** Representan llamadas a métodos donde el emisor espera la respuesta antes de continuar. Se representan con una flecha sólida con cabeza llena.



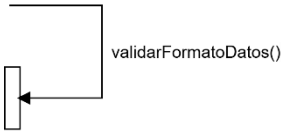
- **Mensajes asíncronos.** Representan comunicaciones donde el emisor no espera respuesta inmediata. Se representan con una flecha sólida con cabeza abierta.



- **Mensajes de retorno.** Muestran valores o confirmaciones que regresan al emisor original. Se representan con una flecha discontinua.



- **Auto-mensajes.** Representan llamadas que un objeto hace a sí mismo. Se representan con una flecha que sale y regresa a la misma línea de vida.



- **Mensajes de creación.** Indican la instanciación de un nuevo objeto. El objeto aparece en el momento de la creación, no al inicio del diagrama. Se representan con una flecha apuntando al objeto recién creado con `<<create>>`.

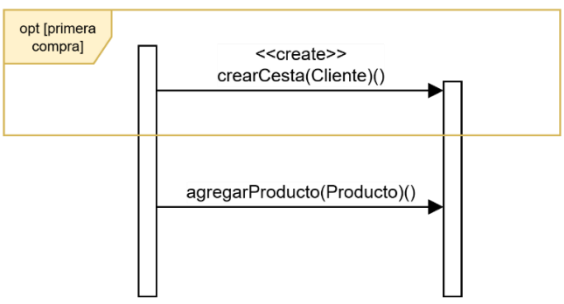
- **Mensajes de destrucción.** Indican la eliminación de un objeto de la memoria. Se representan con una flecha apuntando a una X al final de la línea de vida con `<<destroy>>`.

4.1.5. Fragmentos de interacción

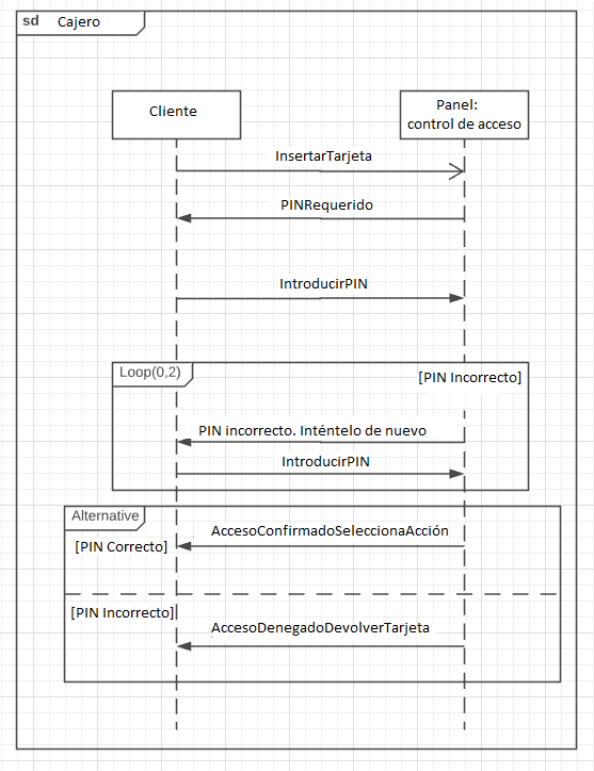
Los fragmentos de interacción (también llamados fragmentos combinados) permiten modelar lógica condicional, bucles y otras estructuras de control:

- ***alt* (alternativa).** Sirve para modelar estructuras *if-then-else*.

- ***opt* (opcional).** Modela una ejecución condicional simple (*if sin else*). Por ejemplo, un cliente podría añadir un producto a la cesta, para lo que el sistema tendría que crearla si es el primer producto en ser añadido:

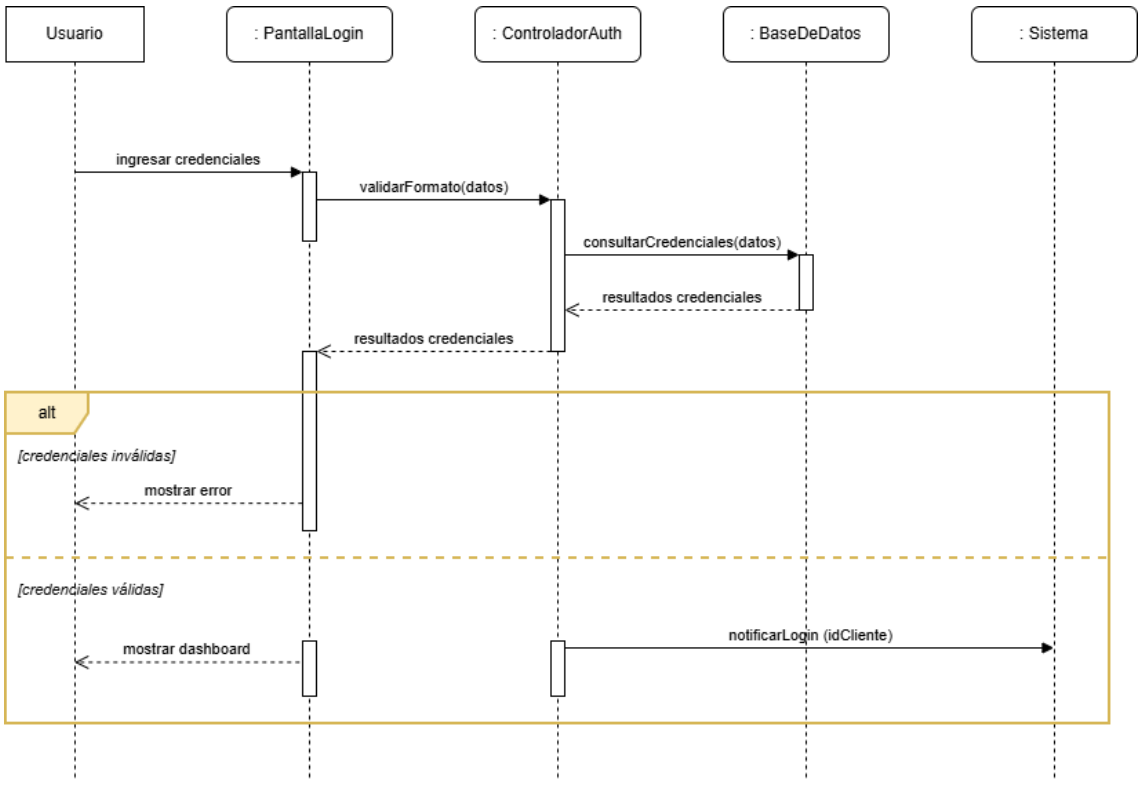


- **loop.** Sirve para modelar bucles.



4.2. Ejemplo completo. Sistema de login

Un posible diagrama de secuencia de un sistema de login podría ser el siguiente:



4.3. Consejos para crear diagramas de secuencia

- Un diagrama, un escenario: Cada diagrama debe mostrar un único flujo de ejecución principal.
- Limita participantes: Cuantos menos participantes involucres, más claro quedará el diagrama. Si necesitas muchos con el paso de muchos mensajes, probablemente podrías dividir el diagrama en dos o más partes más simples.
- Orden de aparición: Coloca los participantes de izquierda a derecha según su flujo lógico.
- Agrupación temporal: Los mensajes relacionados deben estar cerca visualmente.

4.3.1. Reglas de nomenclatura

Para participantes:

- Actores: Usa roles específicos (Cliente, Administrador, etc.).
- Objetos: Usa nombres descriptivos que reflejen su responsabilidad (`ControladorPedidos`, `ValidadorDatos`, etc.).
- Formato consistente: Mantener el mismo estilo de nomenclatura en todos los diagramas y sé consecuente en el proyecto.

Para mensajes:

- Usa verbos en infinitivo: `aniadirProducto()`, `crearPedido()`…
- Indica parámetros relevantes: Incluye parámetros importantes en los mensajes: `buscarProducto(codigo)`, `transferir(origen, destino)`.
- Especifica valores de retorno: Indica los valores de retorno significativos: `return: productoEncontrado`, `return: true`.

4.3.2. Errores comunes a evitar

- **Mezclar niveles de abstracción**: No combines detalles de implementación con lógica de negocio. Por ejemplo, mostrar llamadas a métodos internos de la base de datos (`abrirConexion()`, `commit()`) junto con operaciones de alto nivel como `realizarCompra()`.
- **Diagramas excesivamente complejos**: No trates de mostrar todos los casos posibles en un solo diagrama. Por ejemplo, evita incluir todas las alternativas, excepciones y validaciones en un único diagrama de secuencia de una compra. Dificulta su lectura.

- **Mensajes ambiguos**: No uses nombres genéricos, como `procesar()` o `notificar()`.
- **Ignorar el tiempo**: No respetar el orden cronológico de las interacciones.
- **Objetos innecesarios**: Incluir participantes que no añaden valor al escenario.

4.3.3. Relación con otros diagramas UML

Los diagramas de secuencia se complementan con otros diagramas UML:

Con los casos de uso:

- Cada caso de uso puede generar múltiples diagramas de secuencia (flujo principal, alternativos, excepciones).
- Los actores del caso de uso aparecen como participantes en los diagramas de secuencia.

Con diagramas de clases

- Los objetos del diagrama de secuencia son instancias de las clases del diagrama de clases.
- Los mensajes corresponden a métodos definidos en las clases.

Ayudan a validar el diseño de clases, identificando métodos faltantes o responsabilidades mal asignadas.

5. Diagramas Entidad-Relación

Los diagramas Entidad-Relación (E-R) se sitúan en la fase de diseño de la base de datos del ciclo de vida del software, específicamente después de haber definido los casos de uso y el diagrama de clases, y antes de la implementación física de la base de datos.

El modelo Entidad-Relación fue desarrollado por Peter Chen en 1976 y se ha convertido en el estándar de facto para el modelado conceptual de bases de datos. Este modelo permite representar la estructura lógica de una base de datos de manera independiente del sistema de gestión de bases de datos (SGBD) que se vaya a utilizar posteriormente.

Los diagramas E-R forman parte del **proceso de diseño de bases de datos**, que consta de tres niveles:

1. Diseño conceptual: Se utilizan los diagramas E-R para capturar los requisitos de datos sin considerar aspectos técnicos.
2. Diseño lógico: Se transforma del modelo E-R al modelo relacional (tablas, claves, etc.).
3. Diseño físico: Implementación específica en un SGBD particular.

Un diagrama Entidad-Relación es una representación gráfica que muestra las entidades (objetos o conceptos) de un sistema de información, sus atributos (propiedades) y las relaciones entre dichas entidades.

5.1. Elementos principales de los diagramas E-R

5.1.1. Entidades

Una entidad representa un objeto, concepto o "cosa" del mundo real sobre la cual se desea almacenar información. Las entidades pueden ser fuertes o débiles:

- **Entidades fuertes.** Existen por sí mismas, no dependen de otras entidades para existir. Se representan mediante un rectángulo simple. Por ejemplo: Cliente, Producto, Empleado.
- **Entidades débiles.** Son entidades cuya existencia depende de otra entidad (entidad propietaria) y no pueden ser identificadas únicamente por sus propios atributos. Se representan mediante un rectángulo con doble línea. Por ejemplo: una entidad Habitación depende de la existencia del hotel donde está situada.

5.1.2. Atributos

Los atributos son las propiedades o características que describen a una entidad. Los atributos deben contener un solo valor indivisible y se representan mediante un óvalo simple conectado a la entidad (en ocasiones también los encontrarás sin forma geométrica). Por ejemplo: `nombre`, `fecha_nacimiento`, `salario`, etc.

5.1.3. Claves

Las claves son atributos o conjuntos de atributos que identifican únicamente a las instancias de una entidad.

- **Clave candidata**. Una clave candidata es un atributo o conjunto de atributos que identifican de manera única una fila de una entidad. Por ejemplo, en la entidad `Persona` tanto `dni` como `email` serían buenas claves candidatas.
- **Clave primaria**. De entre el conjunto de claves candidatas, una de ellas debe ser clave primaria. Se representan subrayando el atributo o atributos.
- **Clave foránea**. Es un atributo que referencia la clave primaria de otra entidad. Se define en las relaciones.

5.1.4. Relaciones

Las relaciones representan asociaciones entre dos o más entidades. Describen cómo las entidades están conectadas o interactúan entre sí en el mundo real.

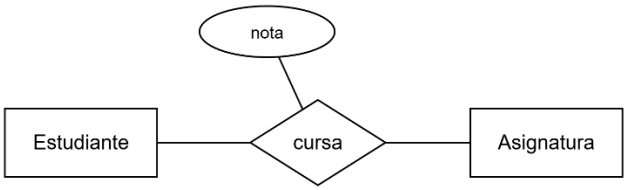
La representación básica es un rombo conectado a las entidades participantes. Dentro del rombo se indica un verbo en tercera persona del singular que describe la acción o asociación.

La **cardinalidad** especifica cuántas instancias de una entidad pueden estar asociadas con instancias de otra entidad.

- **Relación uno a uno (1:1).** Cada instancia de la primera entidad se relaciona con exactamente una instancia de la segunda entidad, y viceversa. Por ejemplo, en la relación `Persona` - `Pasaporte` cada persona tiene un pasaporte único.
- **Relación uno a muchos (1:N).** Cada instancia de la primera entidad puede relacionarse con múltiples instancias de la segunda entidad, pero cada instancia de la segunda se relaciona con exactamente una de la primera. Por ejemplo, en la relación `Departamento` - `Empleado` un departamento tiene muchos empleados, pero cada empleado pertenece a un departamento.
- **Relación muchos a muchos (N:M).** Cada instancia de cualquiera de las entidades puede relacionarse con múltiples instancias de la otra. Por ejemplo, en la relación `Estudiante` - `Asignatura` un estudiante puede cursar varias asignaturas, y una asignatura puede tener varios estudiantes.

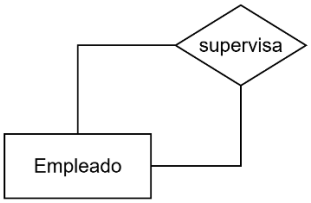
5.1.5. Atributos de las relaciones

Las relaciones también pueden tener atributos propios, especialmente en relaciones muchos a muchos. Por ejemplo, en la relación `Estudiante` - `cursa` - `Asignatura` podríamos tener el atributo `nota`.



5.1.6. Relaciones recursivas

Una entidad puede relacionarse consigo misma. Por ejemplo, en la relación `Empleado` - `supervisa` - `Empleado` (un empleado supervisa a otros empleados).



5.2. Proceso de creación de un diagrama E-R

El desarrollo de un diagrama Entidad-Relación comienza con un **análisis detallado de los requisitos del sistema**. En esta etapa, es fundamental entender el problema que vamos a modelar y prestar atención a los **sustantivos clave**, ya que suelen señalar las **entidades principales** sobre las que se debe almacenar información. Por ejemplo, en un sistema de biblioteca, entidades típicas serían `Libro`, `Usuario`, `Autor` o `Prestamo`. Una vez identificadas las entidades, se procede a determinar los atributos relevantes para cada una, considerando tanto los datos básicos como aquellos calculados o de control. Así, para la entidad `Libro`, los atributos podrían incluir `ISBN`, `titulo` y `fecha_publicación`.

El siguiente paso consiste en **identificar las relaciones entre las entidades**, lo cual suele hacerse prestando atención a los **verbos** presentes en la descripción de los requisitos. Es importante preguntarse cómo interactúan las entidades entre sí; por ejemplo, un usuario puede prestar un libro, o un autor puede escribir varios libros.

Con esta información, se inicia el **modelado conceptual**. Primero se dibujan las entidades principales en el diagrama, añadiendo los atributos más relevantes a cada una. Luego se conectan las entidades mediante las relaciones identificadas, especificando la cardinalidad correspondiente en cada caso. Este primer boceto debe ser revisado y refinado: es necesario comprobar si todas las entidades son realmente necesarias o si falta alguna, validar que los atributos sean completos y no redundantes, y asegurarse de que las relaciones y sus cardinalidades reflejan correctamente las reglas del negocio. Además, es esencial identificar una clave primaria clara para cada entidad.

La **fase de validación y optimización** implica aplicar principios básicos de normalización para eliminar redundancias y asegurar la consistencia de los datos. Aunque la normalización estrictamente hablando se aplica cuando el modelo conceptual se convierte en el modelo lógico, es conveniente tenerla presente en el diseño del diagrama E-R.

5.3. Consejos para crear diagramas E-R

5.3.1. Principios básicos

- **Usa nombres descriptivos**: Los nombres de entidades, atributos y relaciones deben ser claros y auto explicativos.
- **Evita la complejidad excesiva**: Un diagrama muy complejo es difícil de entender y mantener.
- **Mantén la consistencia**: Usa las mismas convenciones de nomenclatura en todo el diagrama.
- **Captura todos los requisitos**: Asegurarse de que el diagrama refleja completamente las necesidades del sistema.
- **Incluye restricciones importantes**: Documenta las reglas de negocio relevantes.

5.3.2. Reglas de nomenclatura

Para entidades

- **Usa sustantivos en singular**: `Cliente` (no `Clientes`).
- **Sé específico**: `Producto` es mejor que `Item`.

Para atributos

- **Usa nombres descriptivos**: `fecha_nacimiento` es mejor que `fecha`.
- **Sé consistente con los tipos**: `codigo_cliente`, `codigo_producto` (no `codigo_cliente`, `id_producto`).
- **Evita abreviaciones confusas**: `telefono` es mejor que `tel` o `tlf`.

Para relaciones

- **Usa verbos en tercera persona**: `realiza`, `contiene`, `pertenece`.
- **Sé específico**: `supervisa` es mejor que `relaciona`.
- **Mantén la direccionalidad**: El verbo debe tener sentido desde la entidad de origen.

5.3.3. Errores comunes a evitar

- **Confundir entidades con atributos**: Convertir en entidad algo que debería ser un atributo. *Ejemplo incorrecto*: Crear entidad `Direccion` cuando podría ser atributo compuesto.
- **Modelar procesos como entidades**: Las entidades representan "cosas", no acciones. *Ejemplo incorrecto*: Crear entidad `Compra` cuando debería ser relación entre `Cliente` y `Producto`.
- **Redundancia de información**: Almacenar la misma información en múltiples lugares. *Ejemplo incorrecto*: Guardar el nombre del cliente tanto en `Cliente` como en `Pedido`.
- **Cardinalidades incorrectas**: No reflejar correctamente las reglas del negocio. *Ejemplo incorrecto*: Modelar `Empleado-Departamento` como N:M cuando debería ser N:1.
- **Claves primarias inadecuadas**: Usar claves que no son únicas. *Ejemplo incorrecto*: Usar `nombre` como clave primaria en lugar de un código único.

5.4. Integración con otros diagramas del proyecto

Los diagramas E-R se relacionan estrechamente con otros diagramas del diseño:

Con diagramas de clases:

- Las entidades del E-R pueden corresponder a clases del dominio en el diagrama de clases.
- Los atributos del E-R se mapean a atributos de las clases.
- Las relaciones E-R pueden convertirse en asociaciones entre clases.
- Diferencia clave: El E-R se enfoca en persistencia, el diagrama de clases en comportamiento.

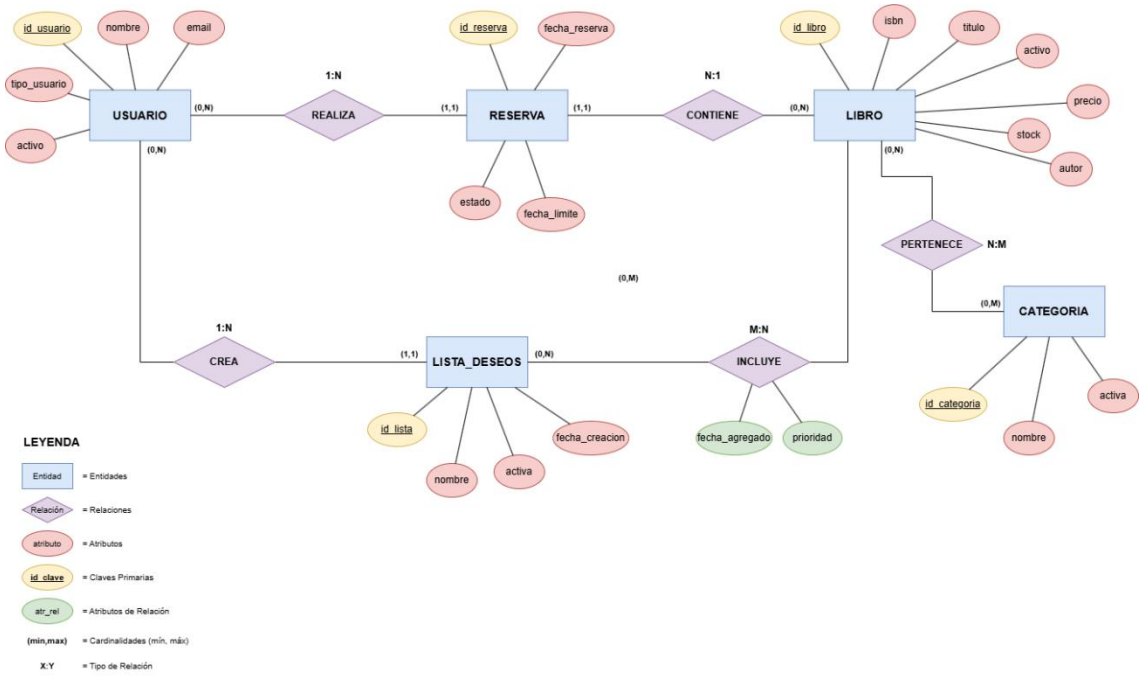
Con diagrama de casos de uso:

- Los casos de uso ayudan a identificar qué datos necesita el sistema.
- Las entidades del E-R almacenan la información manipulada en los casos de uso.
- Los flujos de casos de uso pueden revelar relaciones entre entidades.

Con diagramas de secuencia:

- Los objetos de los diagramas de secuencia pueden requerir persistencia en entidades E-R.
- Las interacciones temporales revelan qué datos se necesitan cuándo.

5.5. Ejemplo: Bookify



6. Modelo lógico o relacional

El modelo lógico se sitúa en la segunda fase del diseño de bases de datos, específicamente después de haber completado el modelo conceptual (diagrama E-R) y antes del diseño físico. Su importancia radica en que transforma el modelo conceptual independiente de tecnología en un esquema que puede implementarse en un sistema de gestión de bases de datos (SGBD) específico.

El modelo lógico es una representación estructurada de los datos que define exactamente cómo se organizarán en tablas dentro de una base de datos relacional. Traduce las entidades, atributos y relaciones del diagrama E-R en tablas, columnas, claves primarias, claves foráneas y restricciones de integridad, siguiendo las reglas del modelo relacional.

6.1. Elementos principales del modelo lógico

6.1.1. Tablas (relaciones)

Las tablas son la estructura fundamental del modelo relacional. Cada tabla representa una entidad del modelo conceptual y está compuesta por filas (tuplas/registros) y columnas (atributos/campos).

Las **características** de las tablas son:

- Nombre único: Cada tabla debe tener un identificador único dentro de la base de datos.
- Estructura fija: Todas las filas tienen la misma estructura de columnas.
- Datos homogéneos: Cada columna contiene valores del mismo tipo de datos.
- Sin orden específico: Las filas no tienen un orden inherente.

Por ejemplo, una definición de una tabla podría ser:

```
CLIENTE (
  dni             VARCHAR(9)      PRIMARY KEY,
  nombre          VARCHAR(100)    NOT NULL,
  apellidos       VARCHAR(150)    NOT NULL,
  email           VARCHAR(200)    UNIQUE,
  telefono        VARCHAR(15),
  fecha_registro  DATE            DEFAULT CURRENT_DATE,
  activo          BOOLEAN         DEFAULT TRUE
);
```

6.1.2. Atributos (columnas)

Las columnas definen las propiedades específicas de cada tabla, especificando qué tipo de información se almacenará y bajo qué restricciones.

Los **tipos de datos** de los atributos define qué clase de valores puede almacenar la columna:

- Numéricos: `INTEGER`, `DECIMAL(10,2)`, `FLOAT`, `BIGINT`.
- Texto: `VARCHAR(n)`, `CHAR(n)`, `TEXT`, `NVARCHAR(n)`.
- Fechas: `DATE`, `TIME`, `DATETIME`, `TIMESTAMP`.
- Booleanos: `BOOLEAN`, `BIT`.
- Binarios: `BLOB`, `VARBINARY(n)`, `IMAGE`.

Además, tienen una serie de **restricciones**:

- `NOT NULL`: La columna debe tener un valor en cada fila.
- `UNIQUE`: Todos los valores de la columna deben ser únicos.
- `DEFAULT`: Valor que se asigna automáticamente si no se especifica.
- `CHECK`: Condición que debe cumplir el valor de la columna.

Por ejemplo:

```
PRODUCTO (
  id_producto     INTEGER         PRIMARY KEY AUTO_INCREMENT,
  codigo_barras   VARCHAR(13)     UNIQUE NOT NULL,
  nombre          VARCHAR(200)    NOT NULL,
  descripcion     TEXT,
  precio          DECIMAL(10,2)   CHECK (precio > 0),
  stock           INTEGER         DEFAULT 0 CHECK (stock >= 0),
  categoria_id    INTEGER         NOT NULL,
  fecha_creacion  TIMESTAMP       DEFAULT CURRENT_TIMESTAMP,
  activo          BOOLEAN         DEFAULT TRUE
);
```

6.1.3. Claves primarias y foráneas

La **clave primaria** identifica inequívocamente cada fila de la tabla. Una clave primaria puede ser simple, con un solo atributo, o compuesta, con varios.

Las características son:

- No puede haber dos filas con el mismo valor de clave primaria.
- Ninguna columna de la clave primaria puede contener valores nulos.
- El valor de la clave primaria no debería cambiar una vez asignado.
- Debe incluir solo las columnas necesarias para garantizar unicidad.

Las **claves foráneas** establecen y mantienen la integridad referencial entre tablas, asegurando que los valores en una tabla correspondan a valores existentes en otra tabla. Sus características son:

- Referencia: Debe apuntar a la clave primaria de otra tabla (o a una clave candidata).
- Integridad referencial: No puede contener valores que no existan en la tabla referenciada.
- Puede ser nula: A menos que se especifique NOT NULL, puede contener valores nulos.
- Acciones en cascada: Puede definir qué hacer cuando se modifica o elimina la fila referenciada.

Un ejemplo de clave foránea podría ser:

```
PEDIDO (
  id_pedido       INTEGER PRIMARY KEY AUTO_INCREMENT,
  dni_cliente     VARCHAR(9) NOT NULL,
  fecha_pedido    DATE DEFAULT CURRENT_DATE,
  total           DECIMAL(10,2),
  FOREIGN KEY (dni_cliente) REFERENCES CLIENTE(dni)
      ON DELETE RESTRICT
      ON UPDATE CASCADE
);
```

De entre las acciones referenciales, tenemos:

- **RESTRICT/NO ACTION**: Impide la eliminación o modificación si existen referencias.
- **CASCADE**: Elimina o modifica automáticamente las filas relacionadas.
- **SET NULL**: Establece la clave foránea a NULL cuando se elimina la referencia.
- **SET DEFAULT**: Establece la clave foránea a su valor por defecto.

Las **claves candidatas** son claves alternativas que podrían servir como clave primaria. Se implementan usando la restricción **UNIQUE**.

```
USUARIO (
  id_usuario  INTEGER PRIMARY KEY AUTO_INCREMENT,
  email       VARCHAR(200) UNIQUE NOT NULL,    -- Clave candidata
  dni         VARCHAR(9) UNIQUE,               -- Clave candidata
  username    VARCHAR(50) UNIQUE NOT NULL      -- Clave candidata
);
```

6.1.4. Restricciones de integridad

Las restricciones de integridad son fundamentales en el modelo lógico, ya que aseguran que los datos almacenados en la base de datos sean coherentes, válidos y reflejen fielmente las reglas del negocio. Existen varios **tipos de integridad** que deben ser considerados durante el diseño.

En primer lugar, la **integridad de entidad** exige que cada tabla cuente con una clave primaria, la cual identifica de manera única cada fila y no permite valores nulos. Esta característica se implementa mediante la declaración de la restricción `PRIMARY KEY`, garantizando así que no existan duplicados ni registros huérfanos en la tabla.

Por otro lado, la **integridad referencial** se encarga de mantener la coherencia entre tablas relacionadas. Esto significa que los valores de las claves foráneas deben corresponder necesariamente a valores existentes en la tabla referenciada. Para lograrlo, se utilizan las restricciones `FOREIGN KEY`, que pueden ir acompañadas de acciones específicas ante la actualización o eliminación de registros, como `CASCADE`, `RESTRICT` o `SET NULL`, entre otras.

La **integridad de dominio** se refiere a la obligación de que los valores almacenados en cada columna pertenezcan a un conjunto permitido o dominio. Esta integridad se implementa definiendo tipos de datos apropiados para cada columna, así como restricciones adicionales como `CHECK`, `NOT NULL` y valores por defecto mediante `DEFAULT`. De este modo, se evita la introducción de datos inválidos o fuera de rango.

Finalmente, la **integridad semántica** va un paso más allá y se ocupa de que los datos respeten reglas de negocio más complejas y específicas, que no siempre pueden ser expresadas únicamente con restricciones básicas. Para ello, se recurre a restricciones `CHECK` más elaboradas, así como a la utilización de *triggers* o procedimientos almacenados que validan condiciones particulares antes de permitir la inserción o modificación de datos.

En conjunto, estas restricciones forman la base de la calidad y fiabilidad de la información en cualquier sistema de bases de datos relacional, y su correcta definición es esencial para evitar errores, inconsistencias y problemas futuros en la gestión de los datos.

Un ejemplo de restricciones semánticas podría ser:

```sql
-- Restricción de rango
ALTER TABLE EMPLEADO
ADD CONSTRAINT ck_salario_positivo
CHECK (salario > 0 AND salario <= 1000000);

-- Restricción de dependencia entre columnas
ALTER TABLE PEDIDO
ADD CONSTRAINT ck_fechas_coherentes
CHECK (fecha_entrega >= fecha_pedido);

-- Restricción de valores específicos
ALTER TABLE PRODUCTO
ADD CONSTRAINT ck_categoria_valida
CHECK (categoria IN ('Electrónicos', 'Ropa', 'Hogar', 'Deportes'));
```

6.2. Transformación del modelo E-R al modelo relacional

Una vez completado el diagrama E-R, es necesario transformarlo al modelo relacional para implementar la base de datos.

6.2.1. Transformación de entidades

Las **entidades fuertes** se convierten directamente en tablas. Los atributos simples se convierten en columnas y la clave primaria del diagrama E-R se mantiene como clave primaria.

Las **entidades débiles** se convierten en tablas que incluyen la clave de la entidad propietaria. La clave primaria se forma combinando la clave parcial con la clave foránea.

6.2.2. Transformación de relaciones

- **Relaciones 1:1**. Se fusionan ambas entidades en una sola tabla con ambas claves o se añade la clave primaria de una entidad como clave foránea en la otra.
- **Relaciones 1:N**. La entidad del lado N recibe la clave primaria del lado 1 como clave foránea.
- **Relaciones N:M**. Se crea una nueva tabla para la relación que contiene las claves primarias de ambas entidades como claves foráneas. La combinación de estas claves forma la clave primaria de la nueva tabla.

6.3. Normalización en el modelo lógico

La normalización es el proceso de organizar los datos de una base de datos para reducir la redundancia y mejorar la integridad de los datos. Se basa en el concepto de **forma normal**.

6.3.1. Primera Forma Normal (1FN)

Una tabla está en 1FN si:

- Cada celda contiene un valor atómico (indivisible).
- No hay grupos repetitivos de columnas.
- Cada fila es única.

Por ejemplo:

```
-- MAL: No está en 1FN
CLIENTE (
  dni     VARCHAR(9),
  nombre  VARCHAR(100),
  telefonos VARCHAR(500)  -- "123456789, 987654321, 555123456"
);
```

El atributo teléfono es multivaluado, hay que corregirlo.

```
-- BIEN: Está en 1FN
CLIENTE (
  dni     VARCHAR(9) PRIMARY KEY,
  nombre  VARCHAR(100)
);

CLIENTE_TELEFONO (
  dni         VARCHAR(9),
  telefono    VARCHAR(15),
  PRIMARY KEY (dni, telefono),
  FOREIGN KEY (dni) REFERENCES CLIENTE(dni)
);
```

6.3.2. Segunda Forma Normal (2FN)

Una tabla está en 2FN si está en 1FN y, además, todos los atributos no clave dependen funcionalmente de toda la clave primaria (no de parte de ella). Por ejemplo:

```
-- MAL: No está en 2FN
MATRICULA (
    codigo_estudiante  VARCHAR(10),
    codigo_asignatura  VARCHAR(10),
    nombre_estudiante  VARCHAR(100), -- Depende solo de codigo_estudiante
    nombre_asignatura  VARCHAR(100), -- Depende solo de codigo_asignatura
    calificacion       DECIMAL(4,2),
    PRIMARY KEY (codigo_estudiante, codigo_asignatura)
);
```

El nombre del estudiante depende solo del código, si sabemos el código de un estudiante, sabremos también su nombre. Pasa lo mismo con el nombre de la asignatura.

```
-- BIEN: Está en 2FN
ESTUDIANTE (
    codigo_estudiante VARCHAR(10) PRIMARY KEY,
    nombre_estudiante VARCHAR(100)
);

ASIGNATURA (
    codigo_asignatura VARCHAR(10) PRIMARY KEY,
    nombre_asignatura VARCHAR(100)
);

MATRICULA (
    codigo_estudiante VARCHAR(10),
    codigo_asignatura VARCHAR(10),
    calificacion      DECIMAL(4,2),
    PRIMARY KEY (codigo_estudiante, codigo_asignatura),
    FOREIGN KEY (codigo_estudiante) REFERENCES ESTUDIANTE(codigo_estudiante),
    FOREIGN KEY (codigo_asignatura) REFERENCES ASIGNATURA(codigo_asignatura)
);
```

6.3.3. Tercera Forma Normal (3FN)

Una tabla está en 3FN si está en 2FN y, además, no existen dependencias transitivas (atributos no clave que dependen de otros atributos no clave).

```
-- MAL: No está en 3FN
EMPLEADO (
  dni                 VARCHAR(9) PRIMARY KEY,
  nombre              VARCHAR(100),
  id_departamento     INTEGER,
  nombre_departamento VARCHAR(100), -- Depende de id_departamento, no de dni
  salario             DECIMAL(10,2)
);
```

El atributo `nombre_departamento` depende de `id_departamento`, que no es clave primaria en esta tabla (es clave foránea).

```
-- BIEN: Está en 3FN
DEPARTAMENTO (
  id_departamento     INTEGER PRIMARY KEY,
  nombre_departamento VARCHAR(100)
);

EMPLEADO (
  dni             VARCHAR(9) PRIMARY KEY,
  nombre          VARCHAR(100),
  id_departamento INTEGER,
  salario         DECIMAL(10,2),
  FOREIGN KEY (id_departamento) REFERENCES DEPARTAMENTO(id_departamento)
);
```

6.3.4. Forma Normal de Boyce-Codd (FNBC)

La Forma Normal de Boyce-Codd es una versión más estricta de la 3FN. Una tabla está en FNBC si, para toda dependencia funcional de la forma X → Y, X es una clave candidata de la tabla. En otras palabras:

- Si un conjunto de atributos determina a otros, ese conjunto debe ser una clave candidata (es decir, debe identificar de forma única cada fila de la tabla).
- Así, la FNBC evita dependencias funcionales donde el determinante no sea clave candidata.

En la mayoría de proyectos es obligatorio normalizar al menos hasta la **Tercera Forma Norma**. La FNBC es una versión más estricta de la 3FN y solo es necesaria en casos donde existen dependencias funcionales complejas o claves candidatas superpuestas. En la práctica, con llegar a 3FN es suficiente para la mayoría de aplicaciones.

6.4. Integración con otros elementos del proyecto

Con los **casos de uso**. El modelo lógico debe soportar todos los escenarios definidos en los casos de uso:

- Cada caso de uso debe poder ejecutarse con los datos disponibles.
- Las consultas necesarias deben ser eficientes con el esquema propuesto.
- Los datos modificados en cada caso de uso deben tener las tablas correspondientes.

Con el **diagrama de clases**. El modelo lógico y el diagrama de clases deben mantener consistencia:

- Entidades del modelo lógico pueden corresponder a clases de dominio.
- Atributos de las tablas se mapean a propiedades de las clases.
- Relaciones entre tablas se reflejan en asociaciones entre clases.
- Restricciones de la base de datos se implementan como validaciones en las clases.

Con los **diagramas de secuencia**. El modelo lógico influye en los diagramas de secuencia relacionados con persistencia:

- Operaciones CRUD (*Create*, *Read*, *Update*, *Delete*) se basan en el esquema.
- Consultas complejas pueden requerir objetos específicos en los diagramas de secuencia.

6.5. Ejemplo: Bookify