Ejercicio con *framework* de persistencia – JPA – Hibernate. Facturas y líneas de factura.

Estructura de Tablas del Ejercicio

Dadas las dos siguientes tablas:

A) ***Facturas(nro(PK), cliente, fecha, total, direccion, cp, ciudad)***

donde:

- ***nro*** es clave primaria.
- ***cliente*** es un texto abreviado que describe al cliente.
- ***fecha*** es la fecha de emisión de la factura.
- ***total*** representa la suma de todos los importes de las líneas de factura asociadas a esa factura.
- ***direccion***, ***cp*** y ***ciudad*** conforman en conjunto la "dirección de facturación".

B) ***LineasFactura(linea, nro (FK -> Facturas), descripcion, unidades, importe)***

donde:

- ***(linea, nro)*** es clave primaria compuesta derivada.
- ***linea*** es el número de línea de la factura
- ***nro*** es la factura a la que pertenece la línea. Clave foránea relativa a la tabla ***Facturas***.
- ***descripcion*** es la descripción en texto del concepto de esa línea de factura.
- ***unidades*** el número de unidades facturadas por ese concepto.
- ***importe*** es el importe de esa línea.

Comentarios generales

- Se proporciona un **esqueleto del proyecto JPA**, con **código parcial** a completar, en UBUVirtual.
- Para la realización del ejercicio, **se seguirá el patrón de diseño DAO, visto en teoría, junto con el uso de JPA**.
- En el paquete `es.ubu.lsi.service.invoice` está la **interfaz de servicio** `Service` que debe ser implementada, conteniendo el método con la **transacción a implementar:**
  - `public void borrarLinea(int linea, int nro) throws PersistenceException;`

  donde dado un número de línea (`linea`), y factura a la que pertenece (`nro`), borra esa línea de factura y descuenta el importe de la línea de factura del total de la factura a la que pertenecía[1].
- Se asume que se utilizan las bibliotecas de usuario de sesiones anteriores con JPA (proyecto `user_lib_JPA` y bibliotecas de usuario asociadas). En concreto son necesarias **cuatro bibliotecas de usuario como mínimo: FileSystemContext, Oracle, SLF4J e Hibernate**.
- El script que crea las tablas, con el SQL detallado, se proporciona por los profesores en el proyecto ubicado en `./sql/script.sql` y **NO** debe ser modificado por los alumnos.

1Normalmente a este tipo de operaciones con todos sus efectos asociados se denomina la ***lógica de negocio***.

Modelo de clases

En este ejercicio se generarán **cuatro clases,** en correspondencia a las dos tablas dadas, en un paquete `es.ubu.lsi.model.invoice`:

- 2 entidades (con anotación `@Entity`)
- 2 tipos embebidos (`@Embeddable`).

Se describen a continuación:

- `Factura`: *entidad* con los datos de una factura. Contendrá un atributo embebido de tipo `DireccionFacturacion` que contiene los atributos correspondientes a los campos en la tabla con nombre `DIRECCION, CP` y `CIUDAD`.
- `DireccionFacturacion`: *tipo embebido* con los atributos `direccion`, `codigoPostal` y `ciudad`. Recordemos que cuando no existe coincidencia entre el nombre del atributo y el campo en la tabla, es necesario indicar con la anotación `@Column(name="")` el nombre de la columna correspondiente en la tabla (e.g., en este caso en la tabla `FACTURA` el nombre del campo es `CP` pero el del atributo debe ser `codigoPostal`).
- `LineaFactura`: *entidad* con los datos de una línea de factura.
- `LineaFacturaId`: *tipo embebido* con los atributos correspondientes a la clave primaria compuesta derivada de una línea de factura.

El diagrama de clases correspondiente, es el mostrado en la Ilustración 1. Se recuerda que en aquellas asociaciones en las que **NO se indican puntas de flechas en ninguno de ambos extremos** deben ser interpretadas como **bidireccionales**.

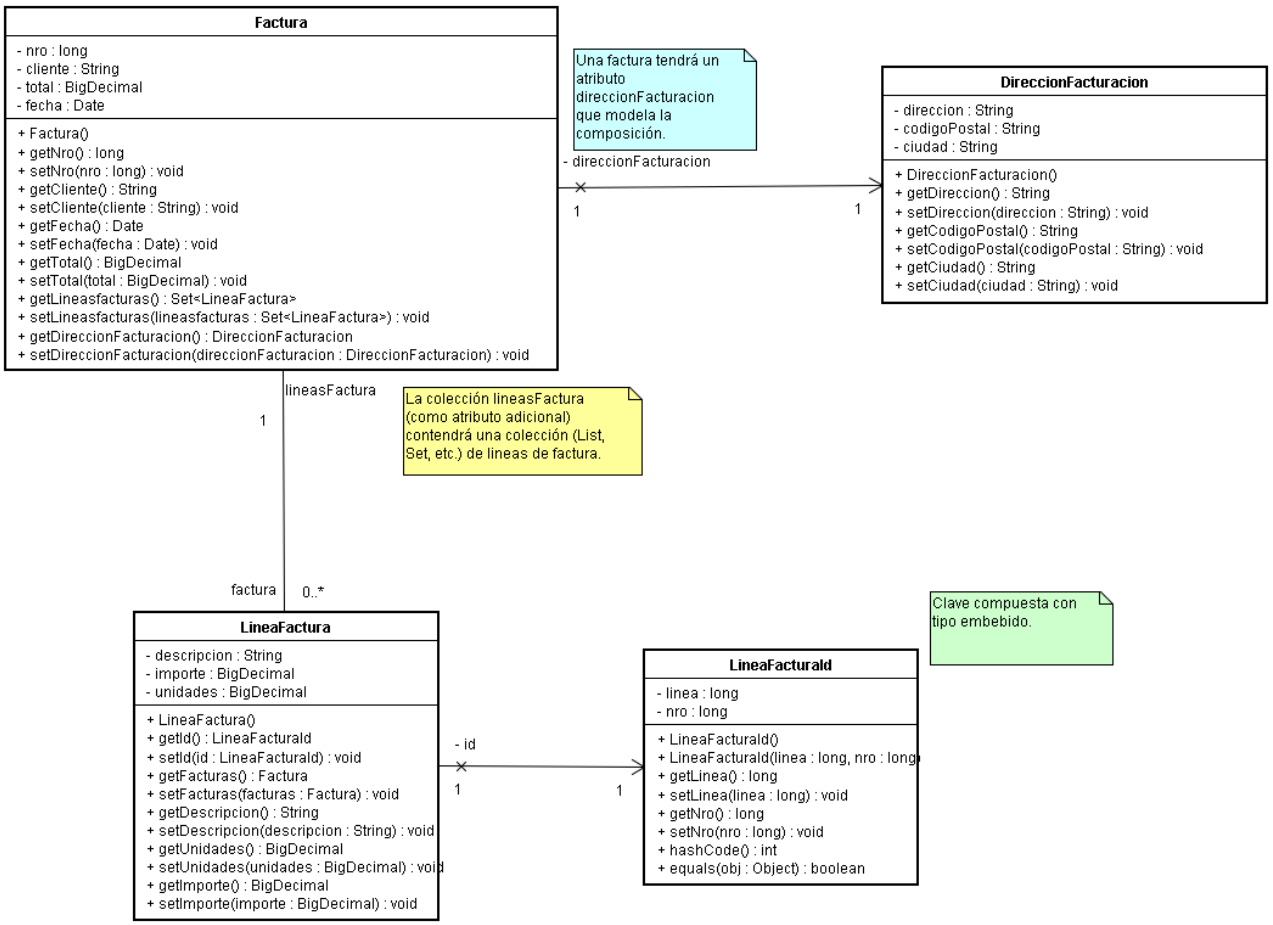

Ilustración 1: Diagrama de clases del modelo de clases persistentes del paquete `es.ubu.lsi.model.invoice` con JPA

Recordemos, que algunos **atributos se deducen de las asociaciones entre clases**. Por ejemplo de la asociación bidireccional de 1 a varios, entre un objeto `Factura` y `LineaFactura`, se deduce que en la factura tendremos un atributo adicional de tipo colección de lineas de factura (e.g., `Set<LineaFactura> lineasFactura`) y que en la linea de factura tendremos un atributo de tipo `Factura`.

Por otro lado se vuelve a hacer hincapié en el uso de un tipo embebido adicional, para guardar los datos de la dirección de facturación, en un objeto de tipo `DireccionFacturacion`. En este caso la `Factura` conoce su `DireccionFacturacion`, pero no al revés, marcando claramente los atributos a declarar (o no) en cada clase **según la navegación.**

Los diagramas de clases **no tienen que se exhaustivos**, sino mostrar parte de la vista del sistema. Se pueden añadir constructores y métodos adicionales, pero se sugiere seguir en la medida de lo posible, las indicaciones de dicho modelo, para homegeneizar las preguntas y dudas que puedan surgir al realizar el ejercicio.

Se recuerda que los métodos `get`, `set`, `equals`, `hashCode` y `toString` se pueden generar **automáticamente** a partir de los atributos incluidos en la clase, **utilizando Eclipse** (botón derecho sobre el código, opción *Source* en el menú contextual y opciones *Generate Getter and Setter...* , *Generate hashCode() and equals()...* o *Generate toString()...*). Nota: tened cuidado a la hora de generar un `toString()` no realizar llamadas infinitas desde un método `toString()` al de otra clase, que a su vez vuelve a llamar al método `toString()` inicial. Esto provoca un conjunto infinito de llamadas y un desbordamiento de la pila de llamadas (`StackOverflowError`).

El modelo **se puede generar automáticamente** desde *Eclipse* con los *plugins* instalados, como se ha visto en sesiones previas, pero posteriormente **habrá que ajustar e incluso crear y corregir cuestiones adicionales, como el uso de claves compuestas (y derivadas), relaciones entre entidades/tipos embebidos o el uso de un embebido para almacenar en un objeto diferente la dirección de facturación**.

En aquellos casos en los que **no hay coincidencia de nombre** entre el modelo de clases y las tablas, recordad que es necesario **introducir anotaciones adicionales.**

Clases DAO

En este ejercicio se solicita implementar dos clases de acceso a datos (DAO):

- `es.ubu.lsi.dao.invoice.FacturaDAO`: acceso a datos de `Factura`.
- `es.ubu.lsi.dao.invoice.LineaFacturaDAO`: acceso a datos de `LineaFactura`.

A la hora de su implementación, hay que tener especial cuidado en la cláusula de herencia con los parámetros actuales que se dan, relativos al tipo de entidad que maneja el DAO y el tipo de la clave primaria de dicha entidad.

Por ejemplo: para una entidad `Alumno` con clave tipo embebido `AlumnoId`, la cláusula de herencia sería: `public class AlumnoDAO extends JpaDAO<Alumno, AlumnoId> {...`, donde el primer tipo es el tipo de entidad a manejar por el DAO y el segundo el tipo de la clave primaria de dicha entidad.

Debido al uso de herencia, las clases DAO de este ejercicio se resuelven de forma casi inmediata al heredar de `es.ubu.lsi.dao.JpaDAO` toda la funcionalidad necesaria para este ejercicio en particular (en próximos ejercicios habrá que completar con más código los DAO). Se aconseja revisar el código de dicha clase ancestro.

Algoritmo Propuesto

Para la **resolución de la transacción**, (suponiendo que tenemos ya el modelo y los DAO), se indican los pasos equivalentes para su implementación con JDBC (pero se puede hacer de otras formas, en particular al abordar la transacción desde el punto de vista de JPA):

1. Si la línea de factura existe:
    1. Consultar el importe de la línea de factura que se va a borrar y guardarlo en una variable.
    2. Borrar la línea de factura que indican los parámetros del método.
    3. Actualizar la factura en la tabla de facturas, descontando del campo total el importe de la factura borrada.
2. Si no indicar el error en la transacción.

Para implementar dichos pasos se deben tener en cuenta las siguientes indicaciones:

- El método concreto `borrarLinea`, con la transacción, se i**mplementa** en la **clase concreta** `es.ubu.lsi.service.invoice.ServiceImpl`.
- Dicha clase debe heredar de `es.ubu.lsi.service.PersistenceService` e implementar la interfaz `es.ubu.lsi.service.invoice.Service`.
- Para implementar la transacción, se tomará una conexión del *pool* de conexiones, a través de la obtención de una instancia de tipo `EntityManager` (revisar el código del método `createSession` en `es.ubu.lsi.service.PersistenceService`, similar a adquirir una conexión de un *pool* de conexiones con JDBC).

- Una vez obtenida la "conexión" (realmente una instancia del `EntityManager`), se pueden **instanciar** los DAO necesarios (e.g., `es.ubu.lsi.dao.invoice.FacturaDAO`, `es.ubu.lsi.dao.LineaFacturaDAO`) que utilizan dicha conexión.
- Con los DAO, podemos utilizar por herencia de `es.ubu.lsi.dao.JpaDAO`, los métodos de búsqueda por clave primaria (`findById`) y los métodos de borrado de entidades (`remove`).
- Una vez recuperado un objeto, se puede implementar la lógica de negocio como si se tratase ya con **objetos Java normales**, recuperando/consultando objetos con sus métodos `get` y modificando su estado con métodos `set`. Trabajando como se hace habitualmente en Java.
- Tened en cuenta que si el objeto no se encuentra con `findById`, se devuelve un valor **`null`**.
- El borrado de un objeto, que es entidad persistente, se realizará a través del método heredado `remove`, del DAO correspondiente.
- La actualización de los valores de los objetos persistentes no requiere de la ejecución de ningún método especial del DAO, sino simplemente modificar el estado del objeto (con sus métodos `set`).
- Al contrario que con JDBC, NO es necesario trabajar con cursores (*result sets*) **ni SQL embebido. Todo el código SQL** se generará de forma transparente por el proveedor concreto, en este ejemplo Hibernate.
- El método debe iniciar la transacción explícitamente (`beginTransaction`) y cometer (`commit`) la transacción si todo va bien. En caso contrario (si saltan excepciones) se deshace (`rollback`). Siguiendo la estructura habitual de una transaccción vista a lo largo de la asignatura.
- Al finalizar la transacción, con indepencia del éxito o fracaso, es **MUY importante cerrar TODOS los recursos abiertos** (en particular la conexión – encapsulada en la instancia del gestor de entidades o `EntityManager` – obtenida previamente con el método `createSession`).

Para facilitar el desarrollo, **se deben utilizar los métodos ya existentes y heredados de las clases proporcionadas**. Poned atención al **código proporcionado** por los profesores, puesto que l**ibera de mucho trabajo y reduce mucho el número de líneas a escribir** en la implementación final de la transacción.

Tratamiento de Excepciones

El método con la transacción lanza excepciones generales del tipo `PersistenceException`. La única excepción registrada en el problema es que **no exista la línea de factura que se pretende borrar**.

En tal caso se recoge como una excepcion `InvoiceException`, descendiente de `PersistenceException`, con su correspondinte código de error definido en `InvoiceError` y mensaje asociado `"No existe esa línea de factura."`. Estas clase y enumeración se proporcionan por los profesores y no deben ser modificadas.

En estos casos, cuando salte la excepción, se retrocede la transacción y se propaga dicha excepción generada por el código.

Todo el resto de excepciones, sean del tipo que sea, también retrocederán la transacción y se propagan al método que la invocó, encapsulada como `PersistenceException`.

Código a implementar y orden sugerido de implementación

1. Paquete `es.ubu.lsi.model.invoice` con las clases/entidades persistentes de JPA necesarias para el modelo de datos (4 clases).
2. Paquete `es.ubu.lsi.dao.invoice`: con los DAO necesarios (`FacturaDAO` y `LineaFacturaDAO` que heredan de `JpaDAO`) (2 clases).
3. Paquete `es.ubu.lsi.service.invoice`: implementación del servicio concreto `ServiceImpl` (1 clase).
   - Ojo: **`extends`** `PersistenceService` **`implements`** `Service` dando código concreto al cuerpo del método `borrarLinea`.

Pruebas "Automáticas"

Las pruebas "automáticas"[2] se proporcionan en el propio proyecto para que los alumnos comprueben que su solución funciona (el mero hecho de "pasar" el test garantiza un mínimo funcionamiento, pero **no implica que la solución sea la ideal u óptima**).

Dichas pruebas se invocan ejecutando el `main` de la clase `es.ubu.lsi.test.TestClient`.

Esas pruebas están implementadas a su vez con JDBC (utilizando un pool de conexiones con JNDI), tal y como se ha venido trabajando en la asignatura previamente en la parte de JDBC. Es una solución híbrida combinando JPA para ejecutar la transacción y JDBC para comprobar el correcto estado en la base de datos.

La creación del recurso JNDI se realiza a través del método `main` de la clase `es.ubu.lsi.test.util.RegisterUCPPool` (generando el correspondiente fichero `.bindings` en `./res` o `./res/jdbc` según usemos Windows o Linux). Es **muy importante ejecutar dicha clase antes de iniciar los tests** para asegurarnos que efectivamente hemos generado el fichero `.bindings`.

El test **reinicia** el estado de la base de datos, ejecutando el script SQL proporcionado. Esto se ve en la traza en pantalla al ejecutar el test, si activamos el log, garantizando que **cada ejecución es independiente de la anterior (se puede repetir cuantas veces se quiera).**

Para probar la correcta implementación del servicio y transacción con JPA, se realizan dos tests:

1. Borra la linea 2 de la factura 1, que sí existe. Para ello, hace el JOIN de facturas con lineas de facturas, y comprueba que el resultado no contiene la fila borrada y que el importe de la factura 1 ahora es 10 (inicialmente era 15).
2. Vuelve a intentar borrar al misma línea 2 de la factura 1 para comprobar que salta la excepción `InvoiceException` indicando que esa línea de factura no existe.

El mesaje final, cuando todo funciona debe ser similar al mostrado a continuación (según las opciones de *log* activadas, en el ejemplo con `rootLogger.level = OFF`):

2 Por motivos de simplicidad no se utiliza un framework de pruebas automáticas como JUnit.

```
INICIANDO TEST...
Probando el servicio...
OK Framework y servicio iniciado
OK Transacción de borrado realizada
OK Borrado
OK Se da cuenta de que no existe la linea de factura
FIN TEST.............
```

Queda a discreción del alumno, el añadir algún test más modificando dicha clase, pero **NO** deberían modificarse los tests ya proporcionados.

Comentarios adicionales

Se comentan algunos detalles **MUY IMPORTANTES** a tener en cuenta y revisar para la correcta ejecución y depuración/corrección de errores que pudieran darse en el desarrollo:

- Las bibliotecas de usuario, enlazadas al proyecto deben ser como mínimo:
  - `FileSystemContext`
  - `Hibernate`
  - `Oracle`
  - `SLF4J`
- La unidad de persistencia JPA debe llamarse "**`Invoices`**".
  - Ese nombre se utiliza en el fichero `persistence.xml` (en la línea `<persistence-unit name="Invoices" transaction-type="RESOURCE_LOCAL">`).
  - Y se define en la clase `es.ubu.lsi.service.PersistenceFactorySingleton`, como constante `PERSISTENCE_CONTEXT_NAME`.
  - Es **vital** quea **ambos valores coincidan** para la ejecución correcta. En caso contrario la unidad de persistencia no será encontrada, ni creada.
- El sistema de log utilizado es SLF4J con Log4J en su versión 2.
  - En el proyecto de ejemplo se proporciona un fichero `log4j2.properties` con dos "*logger*" configurados: uno a consola y otro a un fichero de nombre `log4j.log` que se almacena en el subdirectorio `res`.
  - Para mostrar más o menos información de depuración, se puede cambiar el correspondiente nivel, en dicho fichero, para cada *logger* específico. Actualmente está configurado para ocultar todo por consola (no muestra nada)) y enviar al fichero los mensajes por debajo del nivel `WARN`, pero se pueden cambiar discrecionalmente estos dos valores:
    - `appender.console.filter.threshold.level = OFF`
    - `appender.file.filter.threshold.level = WARN`
  - Tened en cuenta que *Hibernate* también utiliza este sistema de *log*, y por lo tanto se pueden mezclar sus mensajes de *log* con los que hayamos insertado en nuestro código, aunque se puede filtrar la información.
- Si se quieren visualizar además las sentencias SQL que se generan y ejecutan automáticamente por parte de *Hibernate* **en consola** (independientemente de la

configuración del *log*), en el fichero `persistence.xml` se debe cambiar a **`true`** las siguientes propiedades:

- `<property name="hibernate.show_sql" value="false"/>`
- `<property name="hibernate.format_sql" value="false"/>`

- En este ejercicio los DAO a implementar por los alumnos, **no tienen apenas código.**
  - Esto es debido a que **gracias a la herencia**, obtienen la implementación de las superclases.
  - **Ojo, en ejercicios y proyectos posteriores, será neceario incluir más código en dichas clases, en particular para incluir más consultas avanzadas, no solo búsquedas por clave primaria.**
- **En UBUVirtual se proporciona el proyecto con la solución parcial a completar por los alumnos.**
  - **Se recuerda que debe ajustarse correctamente el `classpath` con las bibliotecas de usuario necesarias.**
- **Se proporciona** en UBUVirtual el documento **FAQ - Errores frecuentes con JPA**. Es recomendable revisar dicho documento para corregir errores y antes de consultar dudas con los profesores.