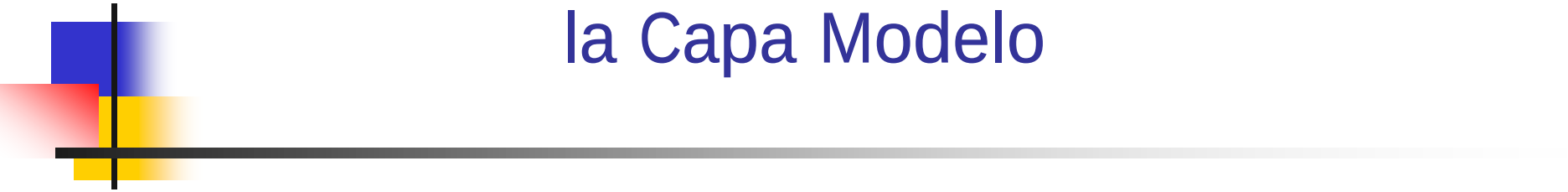
Tema 3: Diseño e Implementación de la Capa Modelo



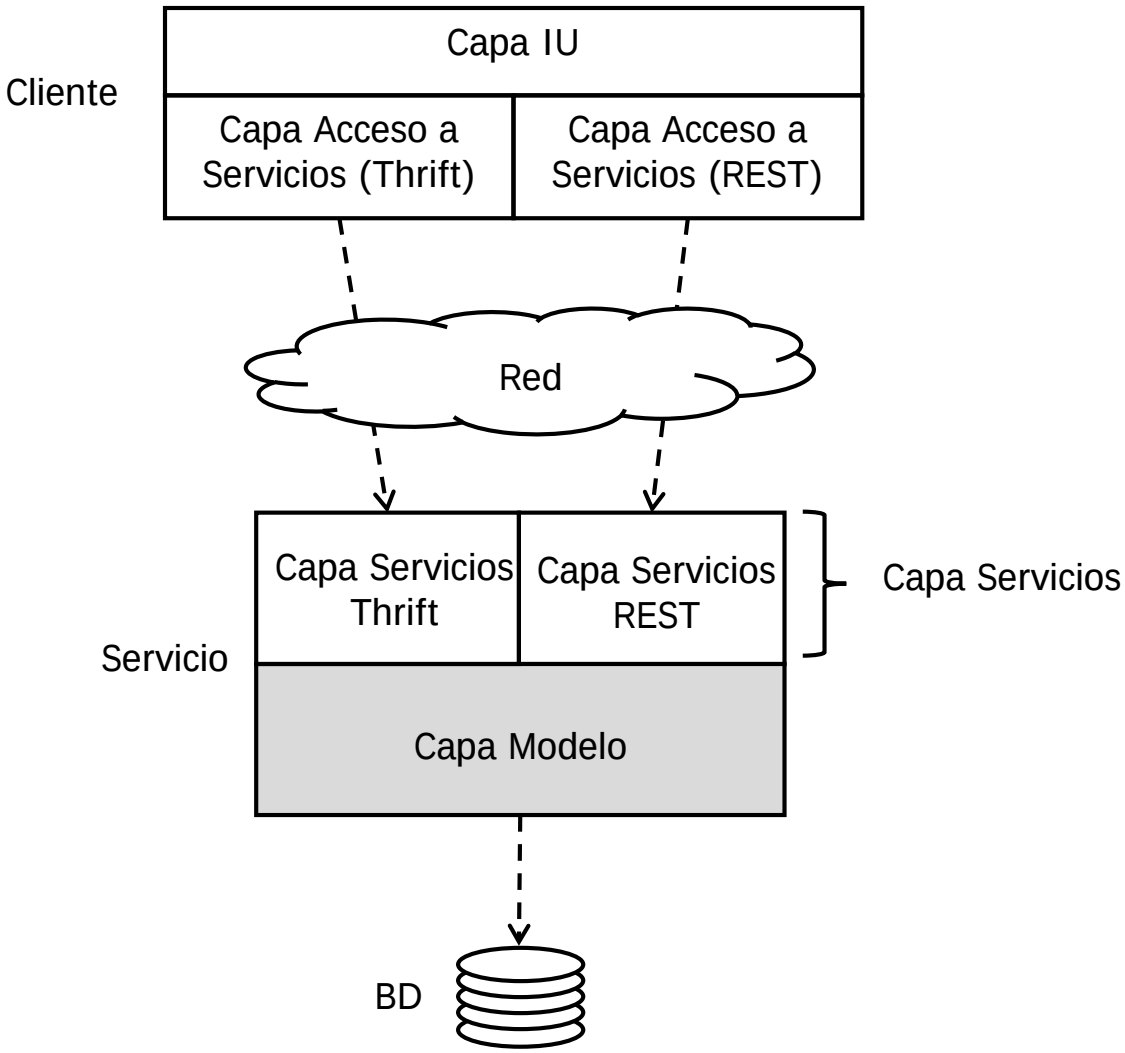
Objetivo

- Aprender un **método** para desarrollar sistemáticamente la capa Modelo de una aplicación
  - El método se apoya en técnicas de diseño consolidadas
  - Las ideas generales del método son independientes de la tecnología
    - Algunos aspectos concretos dependen de las tecnologías usadas en la implementación
    - En esta asignatura, para la implementación usaremos la tecnología de más bajo nivel: JDBC
- ¿Cómo?
  - Estudiando el diseño de la capa Modelo que utiliza el servicio Movies
  - El método es válido para el diseño de la capa Modelo de otro tipo de aplicaciones, como por ejemplo, una aplicación Web

¿Qué es Movies?

- Movies corresponde al módulo **`ws-movies`** de los ejemplos de la asignatura
- Proporciona
  - Una capa Modelo
    - Ofrece operaciones para gestionar información sobre películas (añadir, eliminar, etc.) y comprarlas
  - Dos capas de Servicios, una Thrift y otra RESTful, que delegan en la capa Modelo
    - Exponen gran parte de la lógica de negocio, mediante una interfaz remota y más adaptada a las necesidades de sus clientes
  - Un sencillo cliente de línea de comandos que puede invocar cualquiera de los servicios a través de una interfaz local común

Arquitectura general de Movies



¿Qué ofrece la capa Modelo del servicio Movies? (1)

- Permite gestionar información sobre **películas**
  - Añadir
  - Actualizar
  - Eliminar una película si aún no tiene ninguna compra
  - Buscar una película por su clave
  - Buscar películas por palabras clave del título
- Permite comprar películas
  - El usuario compra la película pagando con tarjeta de crédito
  - Cada compra genera una **venta**, que entre otras cosas, contiene una URL de un servicio de Video Streaming que permite visionar la película las veces que desee durante dos días
  - Es posible recuperar la información de la venta a partir de su clave mientras la venta no expire

¿Qué ofrece la capa Modelo del servicio Movies? (2)

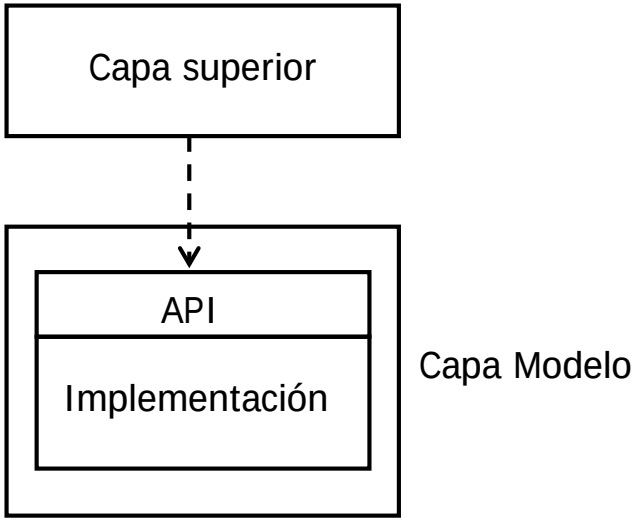
- Película
  - Identificador (clave numérica), título, duración, descripción, precio, fecha de alta en BD
  - El identificador de la película es numérico y se debe generar automáticamente
- Venta
  - Identificador de la venta (clave numérica), identificador de la película (clave foránea), identificador del usuario (clave foránea), fecha de expiración, número de tarjeta de crédito con la que se compró, precio al que se compró la película (el precio de la película puede variar posteriormente), URL de streaming, fecha de la venta
  - El identificador de la venta es numérico y se debe generar automáticamente
- Para simplificar el ejemplo, no se modelará la información específica del usuario (nombre, apellidos, etc.)

¿Qué ofrece la capa Modelo del servicio Movies? (y 3)

- En el contexto de este tema, llamaremos **caso de uso** a cada una de las funcionalidades que ofrece la capa Modelo
- Muchas veces, aunque no siempre, un caso de uso a nivel de modelo corresponde a una funcionalidad concreta que el usuario de la aplicación puede ejecutar
  - Ejemplo: añadir una película, buscar una película, comprarla, etc.

Principios de diseño

- La capa Modelo debe ofrecer una API que
  - **Permita invocar cada caso de uso de manera sencilla** => facilidad de uso para el desarrollador de la capa superior
  - **Que oculte detalles de implementación** => es posible modificar la implementación sin que afecte a la capa superior



Método de desarrollo

- [Paso 1] Modelar las entidades (clases persistentes)
- [Paso 2] Para cada entidad, crear una abstracción que permita gestionar su persistencia
- [Paso 3] Definir una API sencilla para invocar los casos de uso
- [Paso 4] Implementar los casos de uso
- [Paso 5] Implementar las pruebas de integración de los casos de uso

Consideraciones previas (1)

- En **`ws-javaexamples`**: módulos **`ws-util`** y **`ws-movies-model`**
- **`ws-util`**
  - Clases reusables
  - Paquete principal: **`es.udc.ws.util`**
- **`ws-movies-model`**
  - Capa Modelo del ejemplo Movies
  - Paquete principal: **`es.udc.ws.movies.model`**

Consideraciones previas (2)

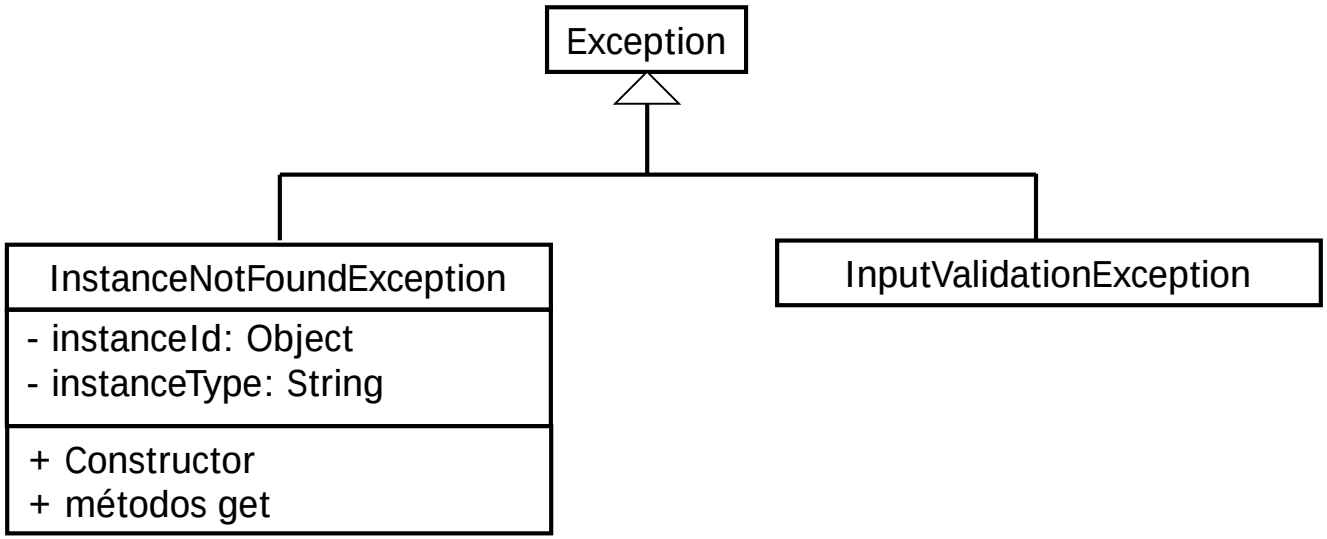
- Tratamiento de excepciones en **`ws-javaexamples`**
  - Errores “lógicos”
  - Errores “graves”
- Errores lógicos
  - Normalmente representan errores del usuario final
  - Ejemplos: actualizar una película que no existe, añadir una película con información de entrada en formato erróneo, etc.
  - En **`ws-javaexamples`**, la capa Modelo notifica este tipo de errores lanzando excepciones de tipo “checked” (hijas de **`Exception`**)
  - Normalmente, este tipo de excepciones se capturan en la interfaz de usuario y se muestran mensajes de error para que el usuario corrija los datos

Consideraciones previas (3)

- Errores lógicos (cont)
  - **`ws-util`** incluye dos excepciones para dos tipos de errores "lógicos" muy frecuentes en casi cualquier aplicación
    - **`InstanceNotFoundException`**: indica que se intenta hacer algo sobre un objeto persistente que no existe (e.g. actualizar/eliminar una película no existente)
    - **`InputValidationException`**: indica que se ha proporcionado una información de entrada en formato erróneo (e.g. un campo numérico fuera de rango o con caracteres no numéricos, una fecha en formato incorrecto, un campo obligatorio en blanco, etc.)
    - Definidas en **`es.udc.ws.util.exceptions`**

Consideraciones previas (4)

- Errores lógicos (cont)

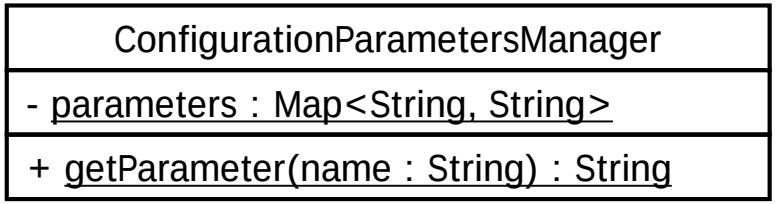


Consideraciones previas (5)

- Errores graves
  - Representan un error de infraestructura
  - Ejemplos: la BD está caída, la tabla a la que se intenta acceder no existe, etc.
  - Trataremos este tipo de situaciones lanzando **`RuntimeException`** (unchecked), con la excepción original encapsulada
  - Típicamente este tipo de excepciones se capturan de manera genérica en un único punto y se le indica al usuario que ha ocurrido un error interno

Consideraciones previas (y 6)

- Parámetros de configuración
  - Para poder leer parámetros de configuración (e.g. parámetros de conexión a la BD), **`ws-util`** proporciona la clase **`ConfigurationParametersManager`** (paquete **`es.udc.ws.util.configuration`**)



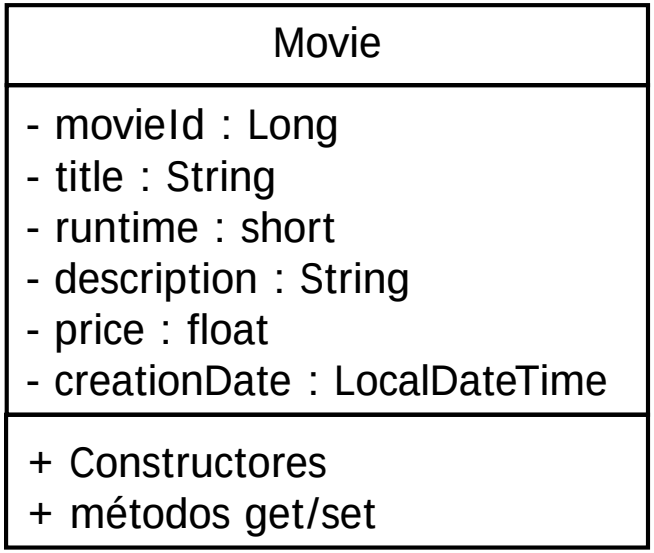
  - Los parámetros se leen de un fichero con nombre **`ConfigurationParameters.properties`**, que debe estar disponible en el classpath
  - Formato del fichero: ejemplo

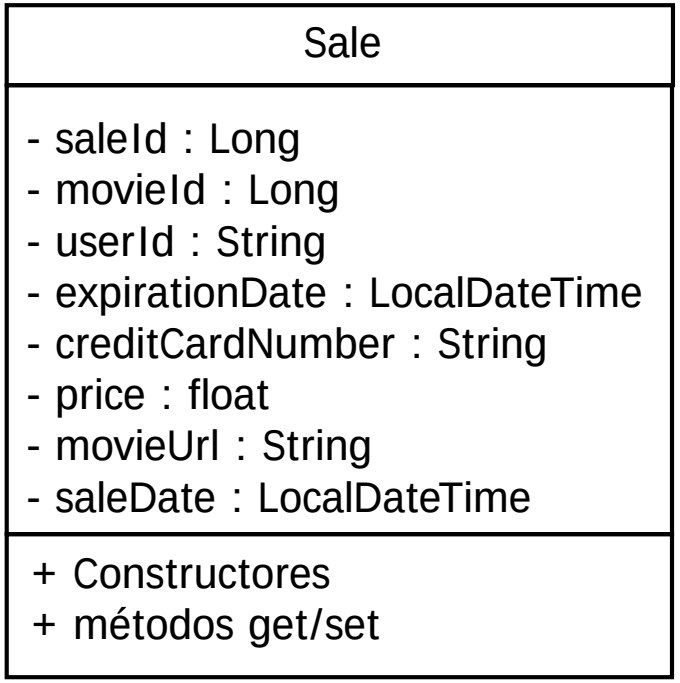
```
# MySQL.
url=jdbc:mysql://localhost/example?...
user=example
password=example
```

[Paso 1] Modelado de entidades (1)

- De la descripción de los casos de uso se deduce que hay que guardar en BD
  - La información sobre las películas
  - Las información sobre las ventas
- Por tanto, definimos dos entidades
  - **`es.udc.ws.movies.model.movie.Movie`**
  - **`es.udc.ws.movies.model.sale.Sale`**

[Paso 1] Modelado de entidades (2)





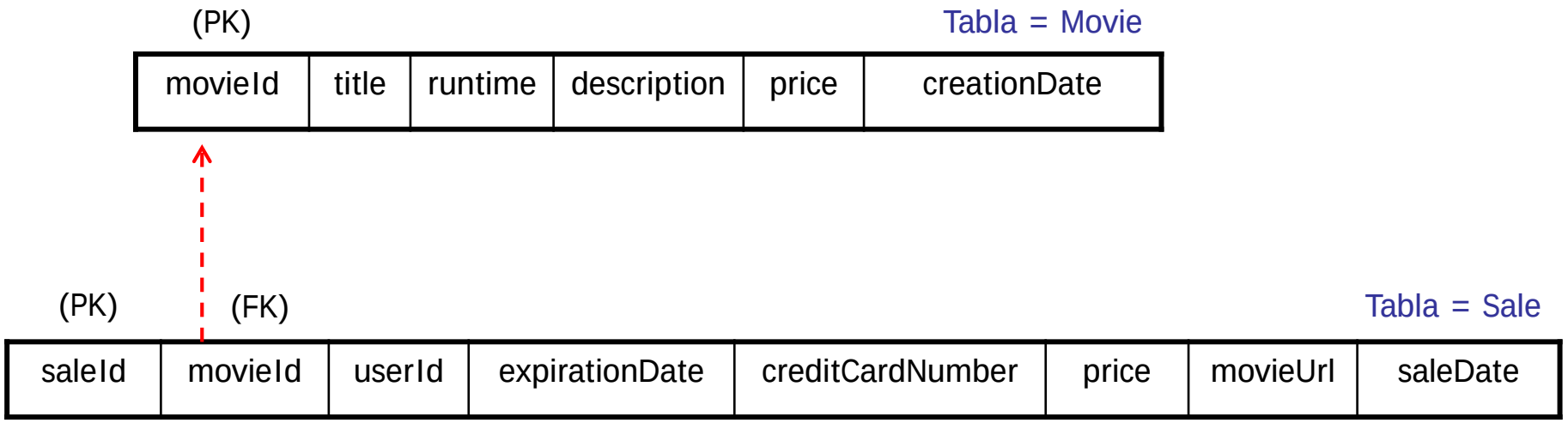
- **NOTA**: en una aplicación real las cantidades monetarias se modelan como **`java.math.BigDecimal`**

[Paso 1] Modelado de entidades (y 3)

- Cada entidad dispone de
  - Atributos privados para modelar el estado
  - Métodos públicos **`get`** para los atributos que se puedan leer
  - Métodos públicos **`set`** para los atributos que se puedan modificar
- La relación entre **`Movie`** y **`Sale`** es 1:N
  - Una película (lado 1) puede tener asociada múltiples ventas (lado N)
  - El atributo **`movieId`** (clave foránea) en **`Sale`** es el que mantiene la relación

[Paso 2] Gestión de la persistencia (1)

- Para guardar las películas y ventas se usan las siguientes tablas
  - Cada instancia de una entidad se guarda en una fila de la tabla correspondiente



[Paso 2] Gestión de la persistencia (2)

- En general, para implementar los casos de uso se puede necesitar para cada entidad
  - Operaciones básicas: insertar/leer/actualizar/eliminar una instancia
    - **CRUD** (Create-Read-Update-Delete)
  - Operaciones avanzadas, típicamente búsquedas que devuelven una colección de instancias (e.g. las películas cuyos títulos contienen un conjunto de palabras clave)
- Ejemplo: caso de uso “comprar una película”
  - Entrada: **`movieId`**, **`userId`**, **`creditCardNumber`**
  - Salida: **`Sale`**

```
Movie movie = << Leer de BD los datos de la película cuyo
    identificador es "movieId" >>;
LocalDateTime expirationDate = LocalDateTime.now().plusDays(2);
Sale sale = new Sale(movieId, userId, expirationDate,
    creditCardNumber, movie.getPrice(), <<getMovieUrl(movieId)>>,
    LocalDateTime.now());
<< Insertar "sale" en BD >>;
```

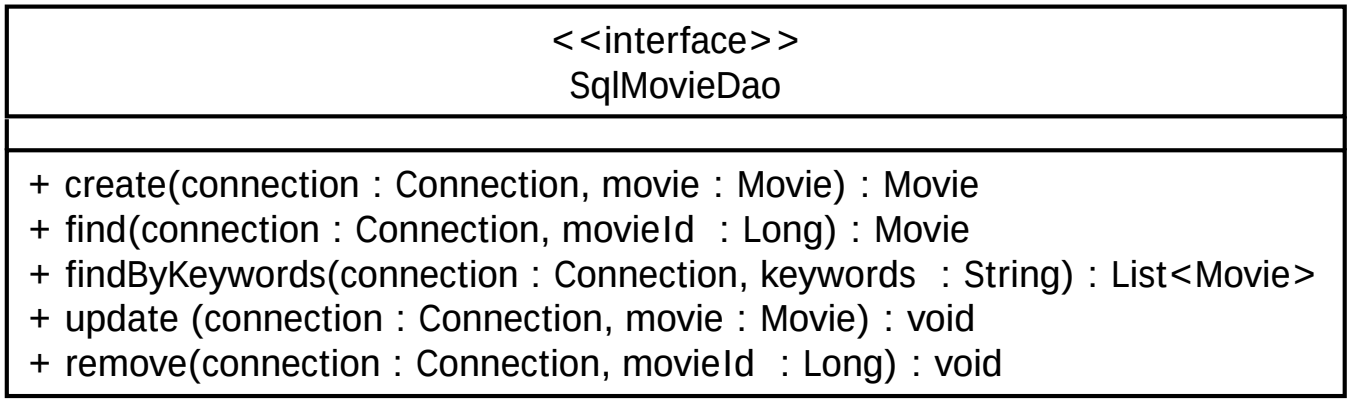
[Paso 2] Gestión de la persistencia (3)

- Otros casos de uso también pueden necesitar estas mismas operaciones de persistencia (leer, insertar, etc.)
- **Por tanto**, para facilitar la implementación de los casos de uso, se necesita una abstracción que permita gestionar la persistencia de cada entidad
- **Además**, sería **ideal** que la abstracción ocultase la BD concreta (e.g. MySQL, Redis, MongoDB, etc.), tipo de BD (e.g. relacional, clave-valor, orientadas a documentos, etc.) y tecnología usada para acceder a la BD (e.g. uso directo de JDBC, API nativa, framework de alto nivel, etc.)
  - Si se cambia de BD o tecnología de acceso, no es necesario cambiar la implementación de los casos de uso

[Paso 2] Gestión de la persistencia (4)

- Esta abstracción corresponde al patrón de diseño **DAO** (Data Access Object)
- Típicamente, un DAO dispone de una interfaz y (al menos) una clase de implementación
- En Movies hay dos DAOs
  - **`es.udc.ws.movies.model.movie.SqlMovieDao`**
  - **`es.udc.ws.movies.model.sale.SqlSaleDao`**
  - Convención de nombrado: **`SqlXxxDao`** (en el mismo paquete que la entidad)
- Conceptualmente, los DAOs corresponden a la “capa Acceso a Datos” estudiada en el tema 1

[Paso 2] Gestión de la persistencia (5)

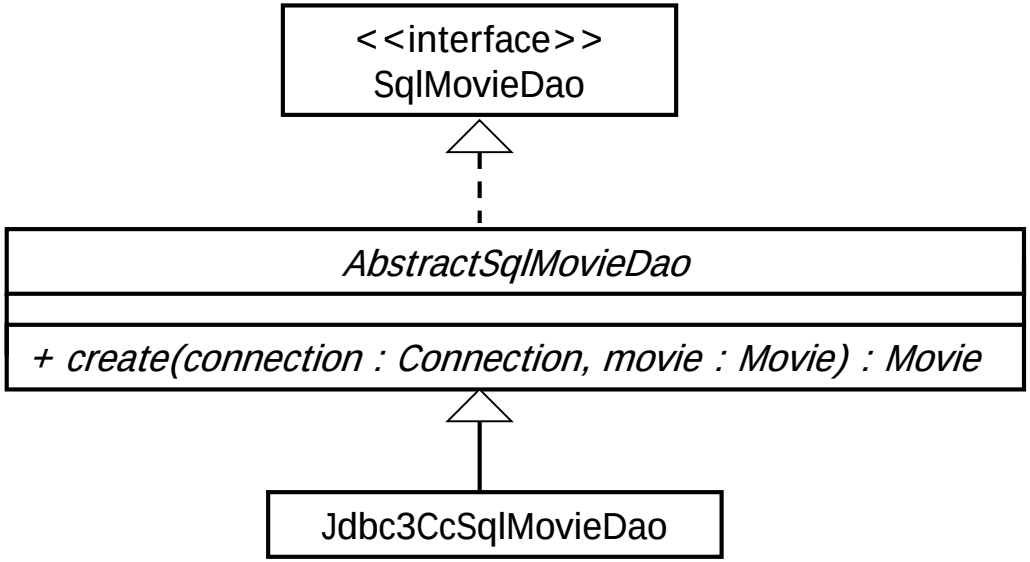


- Los DAOs que definiremos reciben la conexión (**`java.sql.Connection`**), para poder agrupar de manera sencilla varias operaciones de un mismo o distintos DAOs en una misma transacción
- En consecuencia, nuestros DAOs ocultan la BD concreta, pero no su tipo (relacional) ni la tecnología de acceso (JDBC)
  - Y por eso, los nombramos al estilo **`SqlXxxDao`**

[Paso 2] Gestión de la persistencia (6)

```
public interface SqlMovieDao {

    public Movie create(Connection connection, Movie movie);

    public Movie find(Connection connection, Long movieId)
        throws InstanceNotFoundException;

    public List<Movie> findByKeywords(Connection connection,
        String keywords);

    public void update(Connection connection, Movie movie)
        throws InstanceNotFoundException;

    public void remove(Connection connection, Long movieId)
        throws InstanceNotFoundException;

}
```

[Paso 2] Gestión de la persistencia (7)



[Paso 2] Gestión de la persistencia (8)

- La implementación de los métodos de **`SqlMovieDao`** en **`AbstractSqlMovieDao`** es válida para cualquier BD relacional
- **`AbstractSqlMovieDao`** deja sin implementar el método **`create`**
  - Este método tiene que crear dinámicamente la clave de la película
  - Existen múltiples estrategias para generar dinámicamente claves numéricas y no numéricas
  - **`Jdbc3CcSqlMovieDao`** inserta una película en BD usando una estrategia particular para generar dinámicamente la clave numérica de la película

[Paso 2] Gestión de la persistencia (9)

```
public abstract class AbstractSqlMovieDao implements SqlMovieDao {

    protected AbstractSqlMovieDao() {}

    @Override
    public Movie find(Connection connection, Long movieId)
        throws InstanceNotFoundException {

        /* Create "queryString". */
        String queryString = "SELECT title, runtime, description, " +
            "price, creationDate FROM Movie WHERE movieId = ?";

        try (PreparedStatement preparedStatement =
            connection.prepareStatement(queryString)) {

            /* Fill "preparedStatement". */
            int i = 1;
            preparedStatement.setLong(i++, movieId.longValue());

            /* Execute query. */
            ResultSet resultSet = preparedStatement.executeQuery();
```

[Paso 2] Gestión de la persistencia (10)

```
            if (!resultSet.next()) {
                throw new InstanceNotFoundException(movieId,
                        Movie.class.getName());
            }

            /* Get results. */
            i = 1;
            String title = resultSet.getString(i++);
            short runtime = resultSet.getShort(i++);
            String description = resultSet.getString(i++);
            float price = resultSet.getFloat(i++);
            Timestamp creationDateAsTimestamp =
                resultSet.getTimestamp(i++);
            LocalDateTime creationDate =
                creationDateAsTimestamp.toLocalDateTime();
            /* Return movie. */
            return new Movie(movieId, title, runtime, description,
                price, creationDate);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
```

[Paso 2] Gestión de la persistencia (11)

- Observaciones
  - Los objetos **`PreparedStatement`** son recursos
    - Implementan **`AutoCloseable`**
    - Cada método del DAO (e.g. **`find`**) usa un bloque **`try-with-resources`** que cierra el objeto **`PreparedStatement`** inmediatamente cuando se termina de procesar la consulta
      - El **`ResultSet`** asociado también se libera en ese momento por la semántica de **`close`** en **`PreparedStatement`**
  - Las excepciones de tipo **`SQLException`** se relanzan como **`RuntimeException`** porque representan errores de infraestructura

[Paso 2] Gestión de la persistencia (12)

```
@Override
public List<Movie> findByKeywords(Connection connection,
    String keywords) {

    /* Create "queryString". */
    String[] words = keywords.split(" ");
    String queryString = "SELECT movieId, title, runtime, "
            + " description, price, creationDate FROM Movie";
    if (words.length > 0) {
        queryString += " WHERE";
        for (int i = 0; i < words.length; i++) {
            if (i > 0) {
                queryString += " AND";
            }
            queryString += " LOWER(title) LIKE LOWER(?)";
        }
    }
    queryString += " ORDER BY title";

    try (PreparedStatement preparedStatement =
            connection.prepareStatement(queryString)) {
```

[Paso 2] Gestión de la persistencia (13)

```
            /* Fill "preparedStatement". */
            for (int i = 0; i < words.length; i++) {
                preparedStatement.setString(i + 1, "%" + words[i] + "%");
            }

            /* Execute query. */
            ResultSet resultSet = preparedStatement.executeQuery();

            /* Read movies. */
            List<Movie> movies = new ArrayList<Movie>();
            while (resultSet.next()) {
                int i = 1;
                Long movieId = new Long(resultSet.getLong(i++));
                // ...
                movies.add(new Movie(movieId, title, runtime,
                    description, price, creationDate));
            }
            /* Return movies. */
            return movies;
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
```

[Paso 2] Gestión de la persistencia (14)

```
@Override
public void update(Connection connection, Movie movie)
        throws InstanceNotFoundException {

    /* Create "queryString". */
    String queryString = "UPDATE Movie"
            + " SET title = ?, runtime = ?, description = ?, "
            + "price = ? WHERE movieId = ?";

    try (PreparedStatement preparedStatement =
        connection.prepareStatement(queryString)) {

        /* Fill "preparedStatement". */
        int i = 1;
        preparedStatement.setString(i++, movie.getTitle());
        preparedStatement.setShort(i++, movie.getRuntime());
        preparedStatement.setString(i++, movie.getDescription());
        preparedStatement.setFloat(i++, movie.getPrice());
        preparedStatement.setLong(i++, movie.getMovieId());
```

[Paso 2] Gestión de la persistencia (15)

```
            /* Execute query. */
            int updatedRows = preparedStatement.executeUpdate();

            if (updatedRows == 0) {
                throw new InstanceNotFoundException(movie.getMovieId(),
                        Movie.class.getName());
            }

        } catch (SQLException e) {
            throw new RuntimeException(e);
        }

    }
```

[Paso 2] Gestión de la persistencia (16)

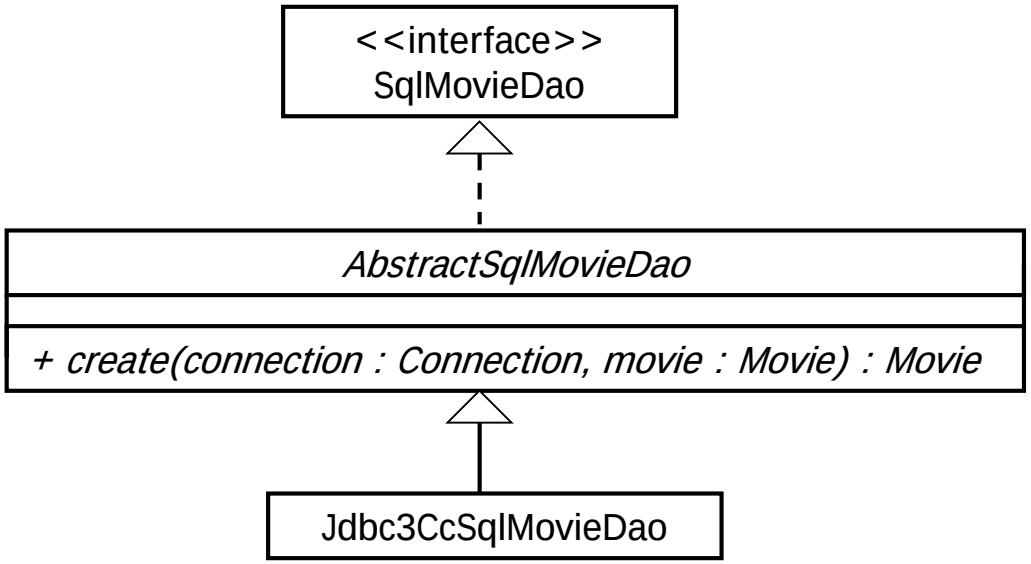
```
@Override
public void remove(Connection connection, Long movieId)
        throws InstanceNotFoundException {

    /* Create "queryString". */
    String queryString = "DELETE FROM Movie WHERE" +
        " movieId = ?";

    try (PreparedStatement preparedStatement =
        connection.prepareStatement(queryString)) {

        /* Fill "preparedStatement". */
        int i = 1;
        preparedStatement.setLong(i++, movieId);
```

[Paso 2] Gestión de la persistencia (17)

```
            /* Execute query. */
            int removedRows = preparedStatement.executeUpdate();

            if (removedRows == 0) {
                throw new InstanceNotFoundException(movieId,
                        Movie.class.getName());
            }

        } catch (SQLException e) {
            throw new RuntimeException(e);
        }

    }

}
```

[Paso 2] Gestión de la persistencia (18)

- Generación dinámica de claves
  - En los ejemplos de la asignatura, el método **`create`** de **`SqlMovieDao`** se implementó en **`Jdbc3CcSqlMovieDao`**, que inserta una película en BD usando una estrategia particular para generar dinámicamente la clave numérica de la película



[Paso 2] Gestión de la persistencia (19)

- Generación dinámica de claves (cont)
  - Existen múltiples estrategias para la generación dinámica de claves numéricas
    - Algunas se apoyan directamente en mecanismos ofrecidos por la BD
    - Otras usan algoritmos para generar claves numéricas en memoria
  - También existen estrategias para generar dinámicamente claves de tipo **`String`**
    - Suelen construir cadenas de caracteres únicas usando una concatenación de varios campos: fecha y hora (en milisegundos), un número aleatorio, etc.

[Paso 2] Gestión de la persistencia (20)

- Generación dinámica de claves numéricas soportadas por las BDs
  - Las BDs suelen diferir en cuanto al soporte que tienen para generar claves numéricas
  - Existen BDs que disponen de “generadores de identificadores numéricos”
    - Contadores (secuencias) que se pueden leer e incrementar atómicamente
    - Ejemplos: Oracle, PostgreSQL, etc.
    - En estas BDs es posible: (1) generar el identificador (mediante una consulta especial, dependiente de la BD) y (2) a continuación insertar la fila con el identificador generado

[Paso 2] Gestión de la persistencia (21)

- Generación dinámica de claves numéricas soportadas por las BDs (cont)
  - Otras BDs disponen de “columnas contador”
    - Cada vez que se inserta una fila, se le asigna automáticamente un valor a esa columna contador (el siguiente)
    - No se puede consultar el valor antes de insertar la fila
    - Ejemplos: MySQL, SQL Server, DB2, etc.
    - En estas BDs es necesario: (1) insertar la fila (sin especificar un valor para la columna contador) y (2) a continuación leer el valor que se le ha asignado a la columna-contador lanzando una consulta especial
    - El paso 2 se puede implementar lanzando una consulta especial, dependiente de la BD, o usando la API de JDBC si se dispone de un driver JDBC 3.0 o superior
    - En los ejemplos de la asignatura, la clase **`Jdbc3CcSqlMovieDao`** utiliza la API de JDBC para el paso 2, por lo que es válida para cualquier BD que proporcione columnas contador y un driver JDBC 3.0 o superior

[Paso 2] Gestión de la persistencia (22)

```
public class Jdbc3CcSqlMovieDao extends AbstractSqlMovieDao {

    @Override
    public Movie create(Connection connection, Movie movie) {

        /* Create "queryString". */
        String queryString = "INSERT INTO Movie"
                + " (title, runtime, description, price, creationDate)"
                + " VALUES (?, ?, ?, ?, ?)";

        try (PreparedStatement preparedStatement =
            connection.prepareStatement(
                queryString, Statement.RETURN_GENERATED_KEYS)) {

            /* Fill "preparedStatement". */
            int i = 1;
            preparedStatement.setString(i++, movie.getTitle());
            // ...
            preparedStatement.setTimestamp(i++,
                Timestamp.valueOf(movie.getCreationDate()));
```

[Paso 2] Gestión de la persistencia (23)

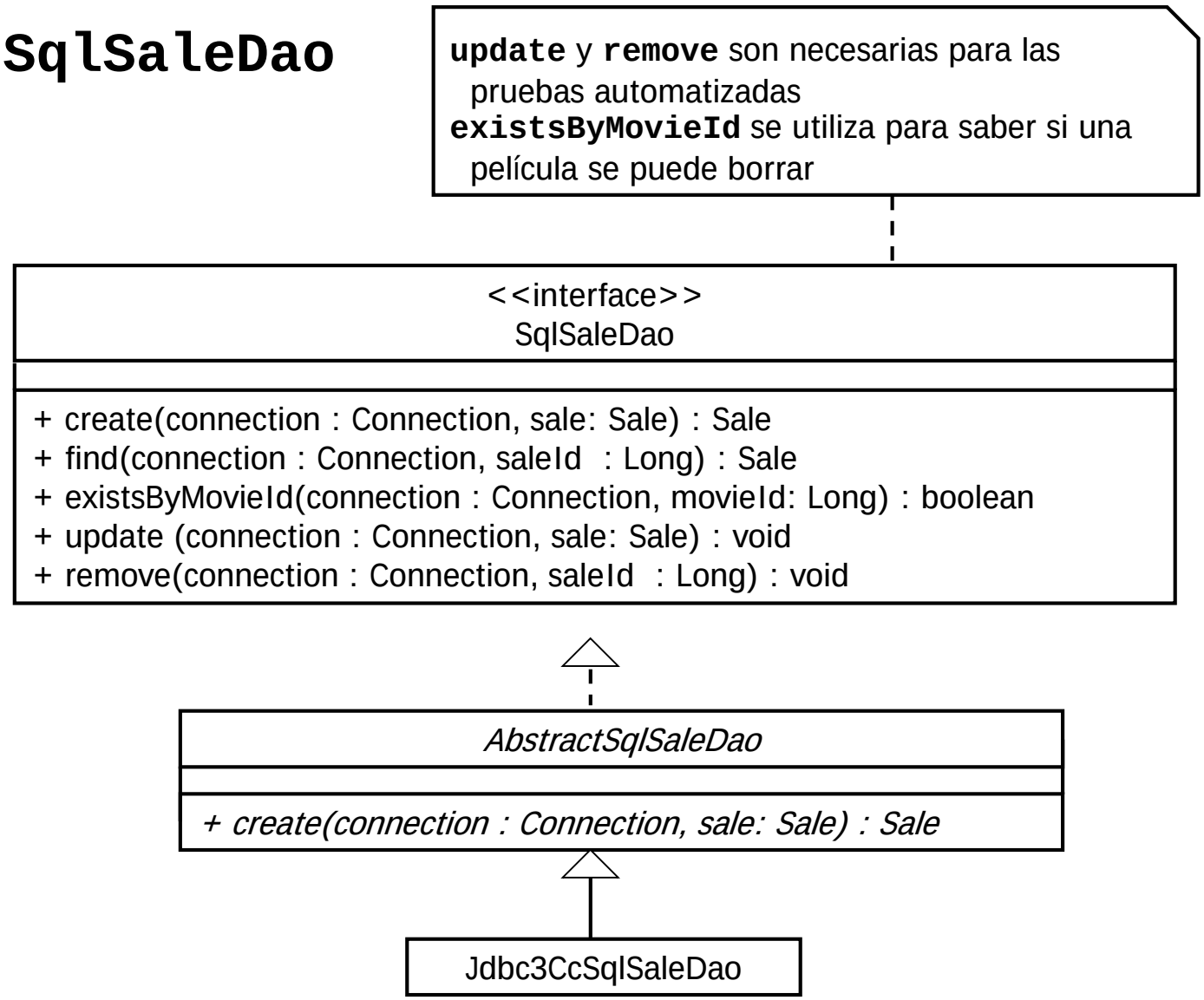
```
            /* Execute query. */
            preparedStatement.executeUpdate();

            /* Get generated identifier. */
            ResultSet resultSet = preparedStatement.getGeneratedKeys();

            if (!resultSet.next()) {
                throw new SQLException(
                        "JDBC driver did not return generated key.");
            }
            Long movieId = resultSet.getLong(1);

            /* Return movie. */
            return new Movie(movieId, movie.getTitle(),
                movie.getRuntime(), movie.getDescription(),
                    movie.getPrice(), movie.getCreationDate());

        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

[Paso 2] Gestión de la persistencia (24)

- **`Jdbc3CcSqlMovieDao`**
  - El objeto **`PreparedStatement`** se crea con una versión de **`prepareStatement`** que recibe adicionalmente el parámetro **`Statement.RETURN_GENERATED_KEYS`**
  - La consulta de inserción que se lanza con el **`PreparedStatement`** no especifica la clave
  - Cuando se ejecuta el método **`getGeneratedKeys`** de **`PreparedStatement`**, el driver de JDBC devuelve el identificador generado durante la inserción (en esa transacción)
    - La consulta devuelve un **`ResultSet`** con un único resultado: el identificador generado

[Paso 2] Gestión de la persistencia (25)

- **`SqlSaleDao`**



[Paso 2] Gestión de la persistencia (y 26)

- Ejemplo: caso de uso “Comprar una película” en términos de DAOs
  - Entrada: **`movieId`**, **`userId`**, **`creditCardNumber`**
  - Salida: **`Sale`**

```
/* Get DAOs. */
SqlMovieDao movieDao = ...
SqlSaleDao saleDao = ...
```

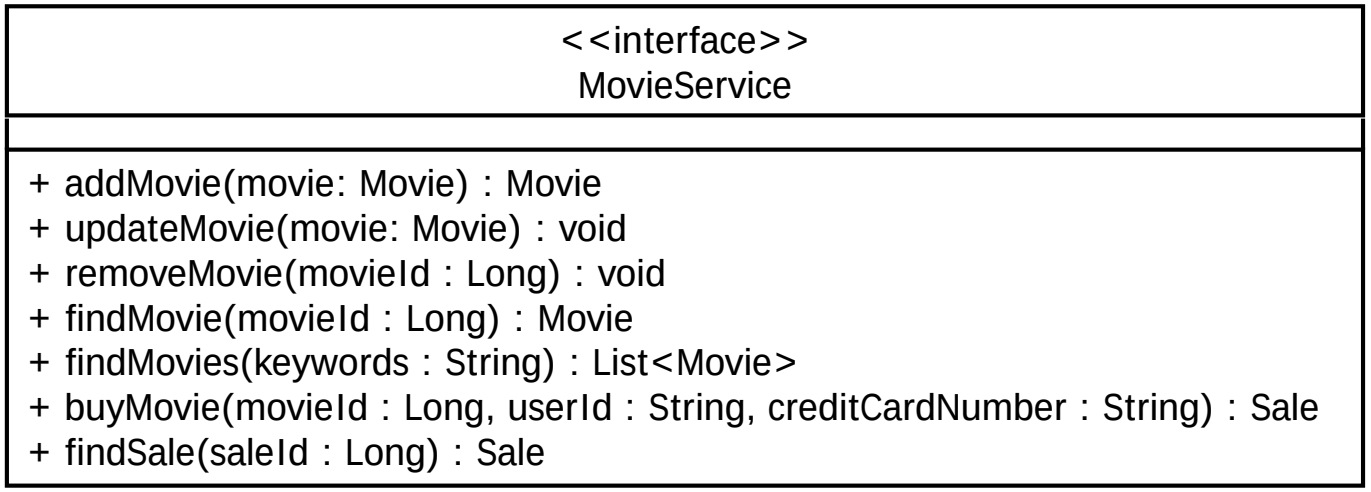
Se explica más adelante cómo resolver este aspecto

```
/* Find Movie. */
Movie movie = movieDao.find(connection, movieId);

/* Create sale. */
LocalDateTime expirationDate = LocalDateTime.now().plusDays(2);
Sale sale = new Sale(movieId, userId, expirationDate,
    creditCardNumber, movie.getPrice(), <<getMovieUrl(movieId)>>,
    LocalDateTime.now());

/* Save sale. */
return saleDao.create(connection, sale);
```

[Paso 3] Definición API modelo (1)

- 1: Agrupar casos de uso de manera lógica
  - Distintas personas pueden pensar en distintos criterios para realizar la agrupación
- 2: Por cada grupo, definir una interfaz
  - En general, contiene un método por cada caso de uso
  - En cada método, los argumentos corresponden a los datos de entrada del caso de uso y el valor de retorno a los datos de salida
  - Cada una de estas interfaces corresponde a la aplicación del patrón de diseño **Facade**
- **Por sencillez**, en Movies sólo se ha definido una fachada
  - **`es.udc.ws.movies.model.movieservice.MovieService`**
  - Convención de nombrado: **`XxxService`**
  - NOTA: en un caso real, quizás podríamos tener dos fachadas: una con los casos de uso “de administración” (añadir, actualizar, eliminar, etc.) y otra con casos de uso orientados al usuario comprador (e.g. buscar, comprar, etc.)

[Paso 3] Definición API modelo (2)



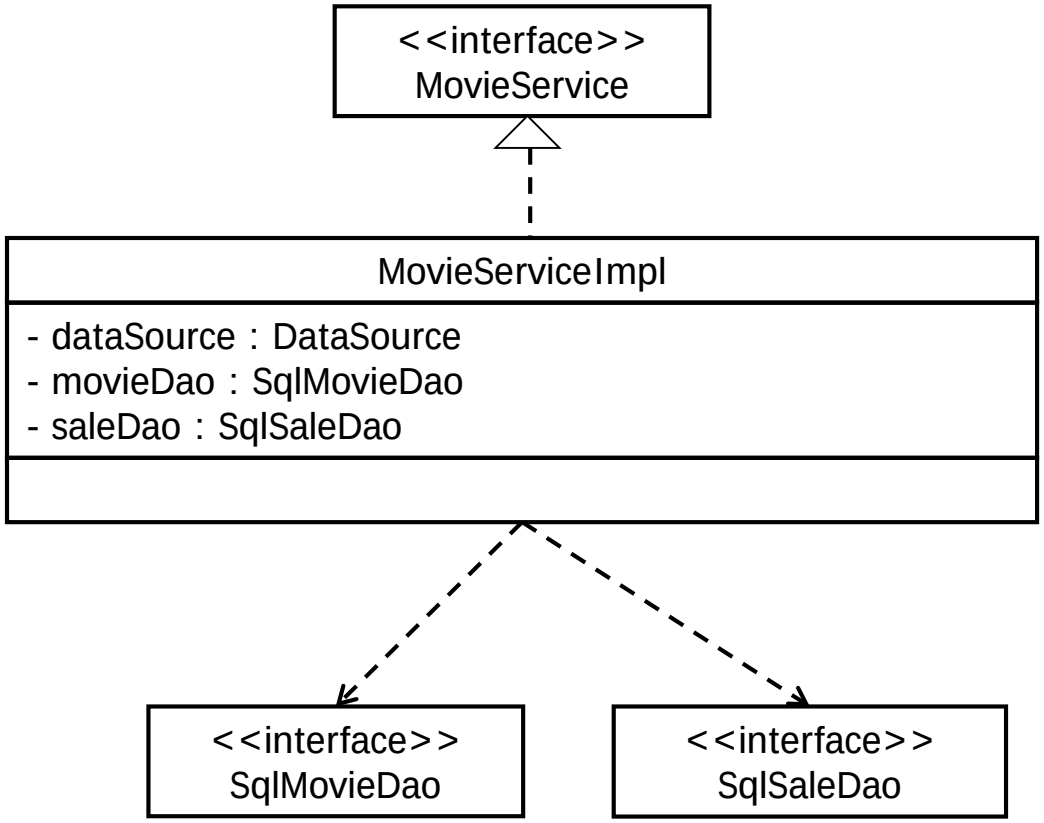
[Paso 3] Definición API modelo (3)

```
public interface MovieService {

    public Movie addMovie(Movie movie) throws InputValidationException;

    public void updateMovie(Movie movie)
        throws InputValidationException, InstanceNotFoundException;

    public void removeMovie(Long movieId)
        throws InstanceNotFoundException, MovieNotRemovableException;

    public Movie findMovie(Long movieId)
        throws InstanceNotFoundException;

    public List<Movie> findMovies(String keywords);

    public Sale buyMovie(Long movieId, String userId,
        String creditCardNumber)
        throws InstanceNotFoundException, InputValidationException;

    public Sale findSale(Long saleId)
        throws InstanceNotFoundException, SaleExpirationException;
}
```

[Paso 3] Definición API modelo (y 4)

- La API del modelo se ha definido como una interfaz (**`MovieService`**)
  - Permite tener implementaciones alternativas
  - Movies proporciona la implementación **`MovieServiceImpl`**
  - En una fase temprana de desarrollo podríamos haber proporcionado de manera rápida una implementación ficticia, e.g. **`MovieServiceMock`**
    - No accede a la base de datos
    - Los métodos no hacen nada o devuelven valores ficticios
  - Si en el equipo de desarrollo hay más de una persona, sería posible desarrollar la capa cliente de la capa Modelo mientras se implementa esta última (los DAOs y **`MovieServiceImpl`**) y en algún momento cambiar de **`MovieServiceMock`** a **`MovieServiceImpl`**
    - La capa cliente podría ser un servicio Web (como en Movies) o la interfaz gráfica (si la capa Modelo fuese local a la interfaz gráfica)
    - La capa cliente sólo trabaja contra la interfaz **`MovieService`**

[Paso 4] Implementación de los casos de uso (1)



[Paso 4] Implementación de los casos de uso (2)

- **`MovieServiceImpl`** implementa los casos de uso haciendo uso de los DAOs
  - Convención de nombrado: **`XxxServiceImpl`** (en el mismo paquete que la interfaz del servicio)
  - Conceptualmente, las clases de implementación de los servicios corresponden a la “capa Lógica de Negocio” estudiada en el tema 1
- Aspectos que hay que tener en cuenta
  - Gestión de transacciones
  - Obtención de referencias a **`DataSource`**
  - Obtención de referencias a DAOs
  - Obtención de referencias al propio servicio **`MovieService`** desde la capa cliente

[Paso 4] Implementación de los casos de uso (3)

- Gestión de transacciones
  - **Seguiremos un enfoque sistemático**
  - Casos de uso que sólo ejecutan una operación de lectura contra la BD
    - Ejemplo: **`findMovie`** y **`findMovies`**
    - Usaremos el modo auto-commit y el nivel de aislamiento por defecto
  - Resto de casos de uso
    - Validar datos de entrada (si es necesario)
    - Empezar transacción
      - Establecer el nivel de aislamiento a **`TRANSACTION_SERIALIZABLE`**
      - Deshabilitar el modo auto-commit de la conexión
    - Comprobar que es posible ejecutarlo en función de otros datos en BD (si es necesario)
    - Si no es posible => **`commit`** (para finalizar la transacción) + lanzar una excepción “checked”
    - Implementar la lógica de negocio usando los DAOs
    - Si se produce una excepción correspondiente a un error grave => **`rollback`** + lanzar excepción
    - En otro caso => **`commit`**

[Paso 4] Implementación de los casos de uso (4)

```
@Override
 public Movie findMovie(Long movieId) throws InstanceNotFoundException {

     try (Connection connection = dataSource.getConnection()) {
         return movieDao.find(connection, movieId);
     } catch (SQLException e) {
         throw new RuntimeException(e);
     }

 }

 @Override
 public List<Movie> findMovies(String keywords) {

     try (Connection connection = dataSource.getConnection()) {
         return movieDao.findByKeywords(connection, keywords);
     } catch (SQLException e) {
         throw new RuntimeException(e);
     }
 }
```

[Paso 4] Implementación de los casos de uso (5)

```
@Override
public Movie addMovie(Movie movie) throws InputValidationException {

    validateMovie(movie);
    movie.setCreationDate(LocalDateTime.now());

    try (Connection connection = dataSource.getConnection()) {

        try {

            /* Prepare connection. */
            connection.setTransactionIsolation(
                Connection.TRANSACTION_SERIALIZABLE);
            connection.setAutoCommit(false);

            /* Do work. */
            Movie createdMovie = sqlMovieDao.create(connection, movie);
```

[Paso 4] Implementación de los casos de uso (6)

```
            /* Commit. */
            connection.commit();

            return createdMovie;

        } catch (SQLException e) {
            connection.rollback();
            throw new RuntimeException(e);
        } catch (RuntimeException|Error e) {
            connection.rollback();
            throw e;
        }

    } catch (SQLException e) {
        throw new RuntimeException(e);
    }

}
```

[Paso 4] Implementación de los casos de uso (7)

```
@Override
public Sale buyMovie(Long movieId, String userId,
    String creditCardNumber)
    throws InstanceNotFoundException, InputValidationException {

    PropertyValidator.validateCreditCard(creditCardNumber);

    try (Connection connection = dataSource.getConnection()) {

        try {

            /* Prepare connection. */
            connection.setTransactionIsolation(
                Connection.TRANSACTION_SERIALIZABLE);
            connection.setAutoCommit(false);
```

[Paso 4] Implementación de los casos de uso (8)

```
/* Do work. */
Movie movie = sqlMovieDao.find(connection, movieId);
LocalDateTime expirationDate =
    LocalDateTime.now().plusDays(SALE_EXPIRATION_DAYS);
Sale sale = sqlSaleDao.create(connection,
    new Sale(movieId, userId, expirationDate,
        creditCardNumber, movie.getPrice(),
        getMovieUrl(movieId), LocalDateTime.now()));

/* Commit. */
connection.commit();

return sale;
```

[Paso 4] Implementación de los casos de uso (9)

```
            } catch (InstanceNotFoundException e) {
                connection.commit();
                throw e;
            } catch (SQLException e) {
                connection.rollback();
                throw new RuntimeException(e);
            } catch (RuntimeException|Error e) {
                connection.rollback();
                throw e;
            }

        } catch (SQLException e) {
            throw new RuntimeException(e);
        }

    }

    private static String getMovieUrl(Long movieId) {
        return BASE_URL + movieId + "/" + UUID.randomUUID().toString();
    }
```

[Paso 4] Implementación de los casos de uso (10)

- No se muestra la implementación de
  - **`updateMovie`**: Misma estructura que **`addMovie`**/**`buyMovie`**
  - **`removeMovie`**: Misma estructura que **`addMovie`**/**`buyMovie`**
    - Comprueba si existe alguna compra para la película que se quiere borrar y en ese caso lanza **`MovieNotRemovableException`**
  - **`findSale`**: Misma estructura que **`findMovie`**/**`findMovies`**
    - Comprueba si la venta ha expirado y en ese caso lanza **`SaleExpirationException`**

[Paso 4] Implementación de los casos de uso (11)

- Validación de datos de entrada
  - La implementación de **addMovie** se apoya en el método privado **validateMovie** para validar el objeto **Movie** que recibe, que a su vez se apoya en la clase **PropertyValidator** (módulo **ws-util**, paquete **es.udc.ws.util.validation**)
  - La implementación de **buyMovie** también hace uso de **PropertyValidator** para validar el formato del número de la tarjeta de crédito
  - **PropertyValidator** es una sencilla clase utilidad que proporciona métodos para validar números enteros, reales, números de tarjetas de crédito (simplificado), etc.
    - Sus métodos lanzan **InputValidationException** si detectan un error de validación

```
private void validateMovie(Movie movie)
    throws InputValidationException {
       PropertyValidator.validateMandatoryString(
           "title", movie.getTitle());
       PropertyValidator.validateLong(
           "runtime", movie.getRuntime(), 0, MAX_RUNTIME);
       // ...
}
```

[Paso 4] Implementación de los casos de uso (12)

- Obtención de referencias a **`DataSource`**
  - En el constructor de **`MovieServiceImpl`** se inicializa el atributo **`dataSource`**

```
public MovieServiceImpl() {
    dataSource = // Estudiaremos ahora cómo resolver
                 // este aspecto...
    movieDao = // ...
    saleDao = // ...
 }
```

  - El código anterior tiene que ser capaz de obtener una referencia al **`DataSource`** de la aplicación en dos situaciones distintas
    - [1] Cuando la capa Modelo se ejecuta dentro de una aplicación Web o un servicio Web (temas 6 y 7)
    - [2] Cuando la capa Modelo es invocada por las pruebas de integración de la capa Modelo (tema 4)

[Paso 4] Implementación de los casos de uso (13)

- Obtención de referencias a **`DataSource`**
  - En [1], la aplicación Web o servicio Web se ejecuta dentro de un entorno (el servidor de aplicaciones) que proporciona DataSources (típicamente con pool de conexiones)
  - En [2], las pruebas se ejecutan en un entorno, generalmente, la línea de comandos (e.g. **`mvn test`**) o un IDE, que NO proporciona DataSources. Es responsabilidad del desarrollador de las pruebas proporcionar el **`DataSource`** a la capa Modelo
- Para que la implementación de los servicios de la capa Modelo (**`MovieServiceImpl`** en el ejemplo), pueda acceder a un **`DataSource`** abstrayéndose de las dos situaciones en las que se ejecutará la capa Modelo, el módulo **`ws-util`** proporciona la clase **`es.udc.ws.util.sql.DataSourceLocator`**

[Paso 4] Implementación de los casos de uso (14)

- Obtención de referencias a **`DataSource`** (cont)
  - **`DataSourceLocator`**

```
public MovieServiceImpl() {
    dataSource = DataSourceLocator.getDataSource(
        MOVIE_DATA_SOURCE);
    movieDao = // ...
    saleDao = // ...
}
```

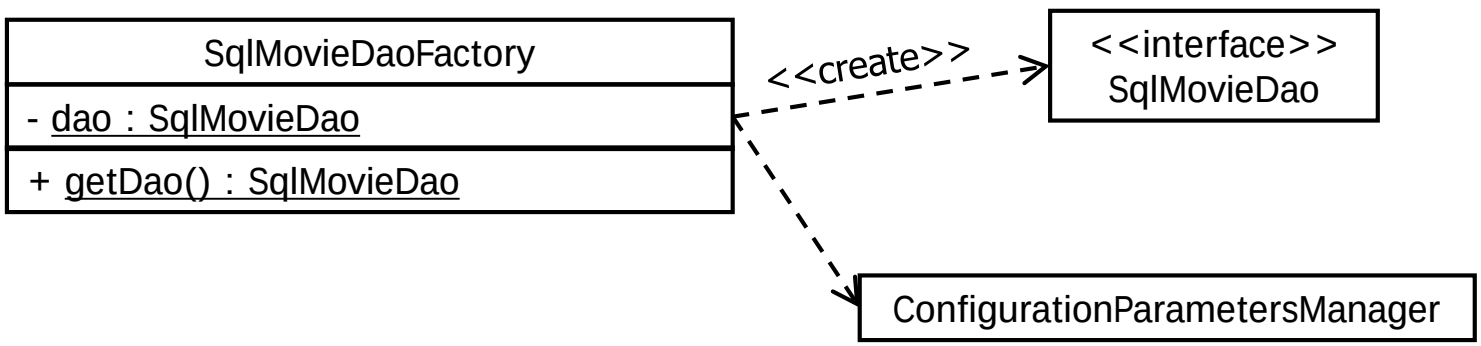
  - El método **`getDataSource`** recibe el nombre del **`DataSource`** que se desea localizar (**`ws-javaexamples-ds`** en el ejemplo)
  - La implementación de **`DataSourceLocator`** se estudiará en el tema 4

[Paso 4] Implementación de los casos de uso (15)

- Obtención de referencias a DAOs
  - En el constructor de **`MovieServiceImpl`** se inicializan dos atributos para los DAOs

```
public MovieServiceImpl() {
    dataSource = DataSourceLocator.getDataSource(
        MOVIE_DATA_SOURCE);
    movieDao = // Estudiaremos ahora cómo resolver
               // este aspecto...
    saleDao = // ...
 }
```

  - Para que sea posible obtener una referencia a un DAO sin conocer su clase de implementación, acudimos al patrón “Factory”



[Paso 4] Implementación de los casos de uso (16)

- Obtención de referencias a DAOs (cont)
  - **`SqlMovieDaoFactory`**
    - Lee el parámetro de configuración **`SqlMovieDaoFactory.className`** y crea una única instancia de esa clase (atributo estático **`dao`**), que es la que siempre devuelve **`getDao`**
    - La factoría trata al DAO como un Singleton, dado que es un objeto que no tiene estado, y en consecuencia puede ser usado trivialmente por múltiples threads
    - Si se desea usar otra estrategia de generación de identificadores numéricos o se cambia a una BD en la que el DAO especificado no sea válido, basta proporcionar una nueva implementación del DAO y modificar el fichero de configuración
  - En **`ws-movies-model/src/main/resources/ConfigurationParameters.properties`**

```
SqlMovieDaoFactory.className=es.udc.ws.movies.model.movie.Jdbc3CcSqlMovieDao
SqlSaleDaoFactory.className=es.udc.ws.movies.model.sale.Jdbc3CcSqlSaleDao
```

[Paso 4] Implementación de los casos de uso (17)

- Obtención de referencias a DAOs (cont)
  - En el constructor de **`MovieServiceImpl`**

```
public MovieServiceImpl() {
    dataSource = DataSourceLocator.getDataSource(
        MOVIE_DATA_SOURCE);
    movieDao = SqlMovieDaoFactory.getDao();
    saleDao = SqlSaleDaoFactory.getDao();
}
```

  - Convención de nombrado: **`SqlXxxDaoFactory`** (en el mismo paquete que la entidad)

[Paso 4] Implementación de los casos de uso (18)

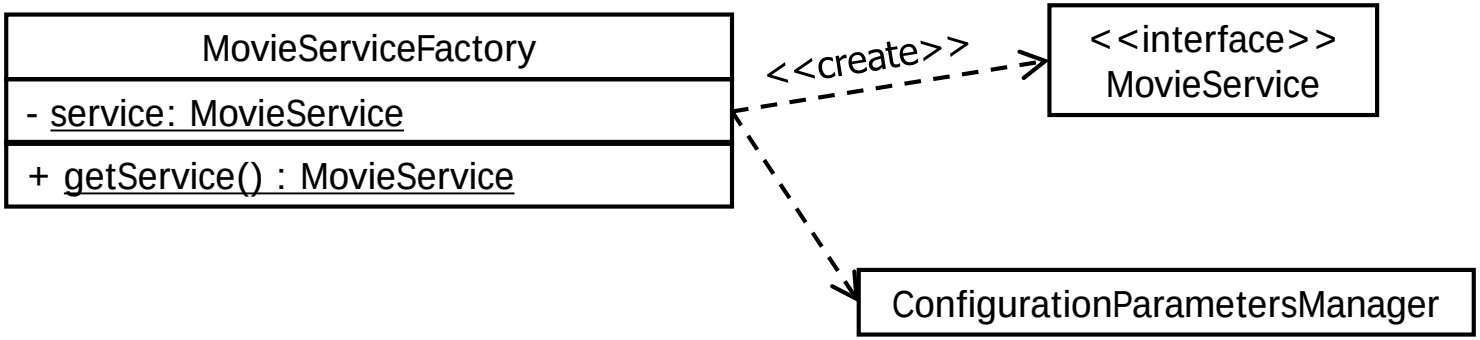
```
public class SqlMovieDaoFactory {

    private final static String CLASS_NAME_PARAMETER =
        "SqlMovieDaoFactory.className";

    private static SqlMovieDao dao = null;

    private SqlMovieDaoFactory() {
    }

    @SuppressWarnings("rawtypes")
    private static SqlMovieDao getInstance() {
        try {
            String daoClassName = ConfigurationParametersManager.
                getParameter(CLASS_NAME_PARAMETER);
            Class daoClass = Class.forName(daoClassName);
            return (SqlMovieDao) daoClass.getDeclaredConstructor().
                newInstance();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

[Paso 4] Implementación de los casos de uso (19)

```
    public synchronized static SqlMovieDao getDao() {

        if (dao == null) {
            dao = getInstance();
        }
        return dao;

    }

}
```

[Paso 4] Implementación de los casos de uso (20)

- Obtención de referencias a **`MovieService`**
  - Para que la capa cliente del modelo no esté acoplada a la clase de implementación de la interfaz **`MovieService`**, también se ha proporcionado una factoría, **`MovieServiceFactory`**, que funciona de manera similar a las factorías de los DAOs
    - Convención de nombrado: **`XxxServiceFactory`** (en el mismo paquete que la interfaz del servicio)
  - Al igual que en el caso de los DAOs, la factoría trata al servicio como un Singleton, dado que no mantiene estado



[Paso 4] Implementación de los casos de uso (y 21)

- Obtención de referencias a **`MovieService`** (cont)
  - En **`ws-movies-model/src/main/resources/ConfigurationParameters.properties`**

```
MovieServiceFactory.className=es.udc.ws.movies.model.movieservice.MovieServiceImpl
```

  - Como se comentaba en el paso 3, en una etapa temprana de desarrollo podríamos especificar **`MovieServiceMock`** en el parámetro de configuración, para ir desarrollando paralelamente la capa cliente y la capa Modelo, y posteriormente especificar la clase de implementación real

[Paso 5] Pruebas de integración de los casos de uso

- Se ha utilizado JUnit (estudiado en una asignatura previa) para implementar las pruebas de integración del servicio **`MovieService`**
- Son pruebas de integración, y no de unidad, porque prueban el correcto funcionamiento de **`MovieService`** (**`MovieServiceImpl`**), e indirectamente lo que ésta utiliza internamente (los DAOs que acceden a la BD)
- Las pruebas de integración del servicio **`MovieService`** residen en **`MovieServiceTest`** y están implementadas al estilo tradicional
  - Convención de nombrado: **`XxxServiceTest`**
  - En general, para cada caso de uso se han diseñado varios casos de prueba, cada uno implementado en un método **`@Test`**
  - A veces resulta conveniente probar más de un caso de prueba en un único método **`@Test`**
- En el tema 4 se explican aspectos específicos a la implementación de pruebas de servicios de la capa Modelo en el contexto de JUnit y JDBC

Temas avanzados (1)

- En este apartado hemos estudiado
  - A nivel de diseño, las técnicas más elementales para construir la capa Modelo de una aplicación y sus pruebas automatizadas
  - A nivel tecnológico, hemos usado la API de JDBC para la implementación de la capa Modelo
- En la asignatura **Programación Avanzada** (Tercer Curso, especialidad “Ingeniería del Software”), se estudian
  - Técnicas de diseño adicionales
  - Las tecnologías Hibernate y Spring Framework para implementar **ágilmente** la capa Modelo y sus pruebas automatizadas
- Hibernate
  - Mapeador objeto-relacional (ORM): permite mapear automáticamente las entidades a una BD relacional
    - Internamente hace uso de JDBC
  - La implementación de DAOs con Hibernate es trivial

Temas avanzados (y 2)

- Spring Framework
  - Inyección de dependencias
    - Ejemplo: es posible construir una instancia de **`MovieServiceImpl`** y que los atributos **`movieDao`** y **`saleDao`** (sus dependencias) se inicialicen automáticamente
    - Evita la necesidad de construir factorías (tanto para DAOs como para servicios)
  - Enfoque declarativo para la gestión de transacciones
    - No es necesario escribir código para empezar una transacción, gestionar excepciones y hacer **`commit`** o **`rollback`**
    - Basta declarar que un método es transaccional (e.g. con **`@Transactional`**)
    - Spring intercepta automáticamente las invocaciones que se hagan a los métodos declarados como transaccionales y se encarga de la gestión de la transacción
  - Ofrece integración con JUnit para agilizar algunos aspectos del desarrollo de pruebas automatizadas
    - Ejemplo: inyectar un servicio a la clase que contiene sus casos de prueba