UNIVERSIDAD DE LA EMPRESA

Bases de Datos III
Práctico 4
Licenciatura en Informática
Ingeniería en Informática

Ejercicio 1

Para cada una de las siguientes afirmaciones sobre aplicaciones en ***arquitecturas de* 2 *capas***, indique si es correcta o incorrecta de acuerdo a los conceptos vistos en el teórico. Fundamente todas sus respuestas.

a) Si una aplicación utiliza el patrón **MVC**, pero ***no*** utiliza los patrones **Facade** y **Value Object**, entonces sigue siendo una aplicación en 2 capas.

b) Si una aplicación utiliza los patrones **Facade** y **Value Object** pero ***no*** utiliza el patrón **MVC**, entonces sigue siendo una aplicación en 2 capas.

c) Si una aplicación utiliza el patrón **Facade** pero ***no*** utiliza los patrones **MVC** y **Value Object**, entonces sigue siendo una aplicación en 2 capas.

d) Toda aplicación en 2 capas necesariamente debe hacer uso de ***stored procedures*** para que se la considere una aplicación en 2 capas.

e) Una aplicación que hace uso de un **Pool de Conexiones**, es necesariamente una aplicación en 2 capas y no una aplicación en 1 capa.

f) Toda aplicación en 2 capas necesariamente posee una arquitectura física ***distribuida***.

Ejercicio 2

Se va a desarrollar una ***aplicación en* 2 *capas*** a utilizarse en el reality show ***RuPaul's Drag Race***. La aplicación correrá en una LAN y tendrá una ***arquitectura física* 3-*Tier*** como se muestra:

Habrá un servidor central en el que residirá una fachada que resolverá los requerimientos y accederá a la base de datos, que residirá en un equipo separado. Los usuarios interactuarán con la aplicación mediante una ***interfaz gráfica de ventanas***, la cual será accedida mediante **RMI** desde los equipos clientes.

El esquema de la base de datos **MySQL** para la aplicación será el siguiente:

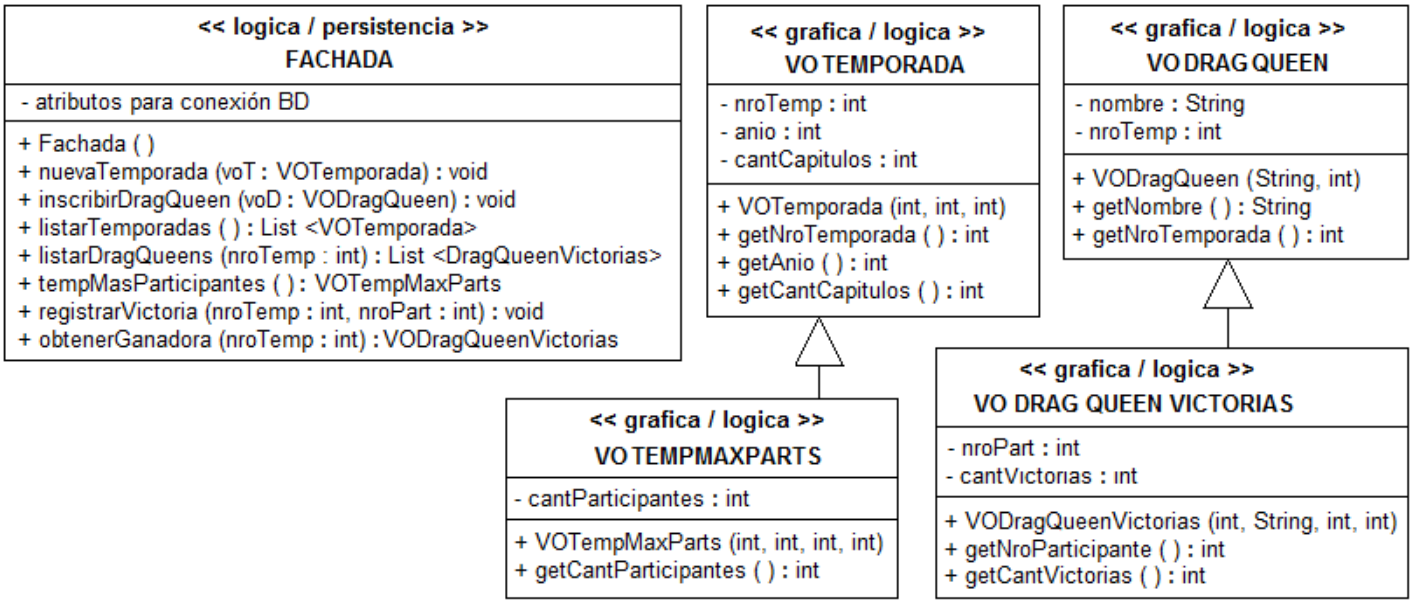
```
Temporadas (nroTemp INT, anio INT, cantCapitulos INT)
DragQueens (nroPart INT, nombre VARCHAR(60), cantVictorias INT, nroTemp INT)
```

- La columna `nroTemp` en `DragQueens` referencia a la columna `nroTemp` en `Temporadas`.
- Los números de las temporadas **no** necesariamente han de ser consecutivos.
- Pueden existir drag queens diferentes con el mismo nombre, incluso dentro de una misma temporada. Lo que las distingue es el número de participante que tienen en dicha temporada.
- Cuando una drag queen es inscripta en una temporada, su cantidad inicial de victorias siempre es cero. A medida que la temporada avanza, cada vez que una drag queen gana un desafío, se incrementa en 1 su cantidad de victorias ganadas.

Escriba un programa `Main` en **Java** que cree la base de datos según el esquema descrito.

Ejercicio 3

En este ejercicio desarrollaremos la aplicación descrita en el ejercicio 2. Por el momento supondremos que existirá un único usuario y **no** nos preocuparemos de realizar manejo de transacciones (eso se hará en el ejercicio 4). Las siguientes clases en UML corresponden a la fachada de acceso a la capa lógica y de persistencia y a los value objects a utilizar para transferir información entre las dos capas:



- El método `nuevaTemporada` registra una nueva temporada, chequeando que no existiera.
- El método `inscribirDragQueen` inscribe una drag queen en una temporada, chequeando que la temporada le corresponde esté registrada. El programa asignará automáticamente a la nueva drag queen el número siguiente al de la última drag queen inscripta en dicha temporada. Por ejemplo, si la temporada tenía 5 drag queens, asignará el nº 6 a la nueva drag queen. De este modo, los números de participantes de la temporada irán quedando consecutivos a medida que se inscriben.
- El método `listarTemporadas` devuelve un listado de todas las temporadas registradas, ordenadas por número de temporada.
- El método `listarDragQueens` devuelve un listado de todas las drag queens de una temporada (chequeando que dicha temporada esté registrada), incluyendo además la cantidad de victorias de cada drag queen en la temporada, ordenadas por número de participante.
- El método `tempMasParticipantes` devuelve los datos de la temporada con la mayor cantidad de participantes, junto con la cantidad de participantes correspondiente (chequeando que exista al menos una temporada). En caso de haber dos o más, devuelve la que tiene número más alto.
- El método `registrarVictoria` agrega una victoria a la drag queen correspondiente al número de temporada indicado, chequeando que ambos estén registrados.
- El método `obtenerGanadora` devuelve la drag queen ganadora de una temporada (chequeando que la temporada exista y tenga al menos una drag queen inscripta), incluyendo además la cantidad de victorias de la ganadora. Se considerará ganadora a la drag queen con la mayor cantidad de victorias en la temporada. En caso de haber dos o más, devuelve la que tiene número más alto.

Observación: En caso de que cualquiera de los chequeos mencionados falle, la fachada lanzará una ***excepción personalizada*** para notificar el correspondiente error a la capa gráfica.

a) Cree un proyecto para la aplicación con la estructura de packages dada en la figura.

b) Implemente en el package `excepciones` todas las clases correspondientes a las posibles excepciones que puedan ocurrir.

c) Implemente en el package `valueObjects` las clases `VOTemporada`, `VODragQueen`, `VOTempMaxParts` y `VODragQueenVictorias` correspondientes a los value objects especificados en UML.

d) Implemente en el package `accesoBD` la clase `Consultas` que define los textos de todas las sentencias SQL que la aplicación necesite ejecutar sobre la base de datos.

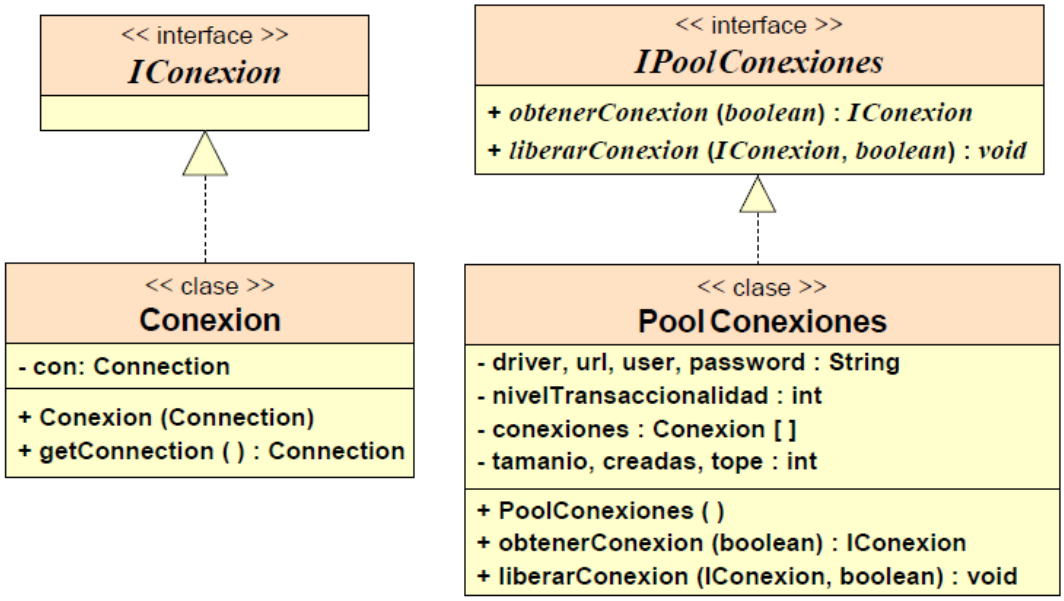
e) Implemente en el package `accesoBD` la clase `AccesoBD` que encapsula el acceso a la BD. Debe poseer los métodos que ejecuten las sentencias SQL definidas en la clase `Consultas`.

f) Implemente en el package `logicaPersistencia` la clase `Fachada` de modo que sus métodos se comporten según la descripción dada antes. Hará uso de la clase `AccesoBD` y será accedida desde los equipos clientes mediante **RMI** (Remote Method Invocation).

g) Implemente en los packages `ventanas` y `controladores` las clases correspondientes a la capa gráfica de la aplicación. Las mismas deberán aplicar el patrón **Model View Controller** y brindar las ventanas necesarias para dar solución a los requerimientos de la aplicación. Los controladores accederán a la Fachada mediante **RMI** (Remote Method Invocation).

Ejercicio 4

En este ejercicio desarrollaremos en **Java** un **Pool de Conexiones** como el visto en el teórico. La implementación a realizar en este práctico utilizará internamente una estructura de arreglo con tope para albergar las conexiones. Luego haremos uso del pool para permitir el acceso de usuarios en forma ***concurrente*** y realizar ***manejo de transacciones*** en la aplicación del ejercicio anterior.



- La descripción de los atributos y métodos de cada una de las clases e interfaces es la que está dada en el capítulo 4 del material teórico.
- Los métodos `obtenerConexion` y `liberarConexion` internamente harán uso de las primitivas `wait` y `notify` de **Java** para manejo de concurrencia.

- Las conexiones irán siendo instanciadas **a demanda**. Es decir, sólo se creará una nueva conexión cuando todas estén actualmente en uso y aún haya espacio para crear nuevas.
- La carga de `driver`, `url`, `user` y `password` de la base de datos será realizada en el método constructor del Pool desde un ***archivo de configuración***.
- Los métodos de la clase del Pool lanzarán una `PersistenciaException` como **única** excepción en caso de ocurrir cualquier error de comunicación con la BD.

a) En la estructura de packages del ejercicio anterior, cree un package `poolConexiones` dentro de la capa lógica/persistencia. Implemente en dicho package las clases e interfaces anteriores con **todos** sus métodos. Luego Haga un programa `main` de prueba que permita testear el correcto funcionamiento del Pool de Conexiones.

b) Incorpore el uso del Pool de Conexiones a la aplicación del ejercicio anterior. Para ello, incorpore a la Fachada el siguiente atributo: `private IPoolConexiones pool;` Luego modifique la implementación de cada requerimiento de modo que:

- Lo primero que haga el requerimiento sea solicitar una *`IConexion`* al pool.
- Mantenga exactamente el mismo comportamiento que ya tenía, sólo que la conexión usada ahora será la obtenida del pool en vez de la conexión aislada usada antes. Dentro de la clase `AccesoBD` deberá ahora castear la *`IConexion`* hacia una `Conexion` concreta.
- Devuelva la *`IConexion`* al pool, finalizando internamente la transacción mediante `commit` o `rollback`, según corresponda.