Iteración 3: Sistemas Transaccionales

Juan Esteban Coronel Yela - 202111207

Juan Pablo Martínez Pineda - 202012623

Contenido

[1 Análisis 2](#_Toc134129608)

[1.1 Ajustes modelo conceptual (UML) 2](#_Toc134129609)

[1.2 Ajustes al modelo relacional (RM) 4](#_Toc134129610)

[2 Diseño de la aplicación 5](#_Toc134129611)

[2.1 Diferencia entre la entrega de la iteración 2 y la entrega de la iteración 3 5](#_Toc134129612)

[2.2 Modelo BCNF y como se trataron las anomalías 6](#_Toc134129613)

[2.3 Tablas generadas en la base de datos 7](#_Toc134129614)

[2.4 Lógica de los nuevos requisitos y sus mecanismos ACID 10](#_Toc134129615)

[3 Construcción de la aplicación 14](#_Toc134129616)

[3.1 Funcionamiento de la interfaz de AlohAndes para realizar transacciones 14](#_Toc134129617)

[3.2 Implementación y resultado de los requisitos funcionales 16](#_Toc134129618)

[**3.2.1** **RF1: Registrar los operadores de alojamiento para AlohAndes** 16](#_Toc134129619)

[**3.2.2** **RF2: Registrar propuestas de Alojamientos para AlohAndes** 17](#_Toc134129620)

[**3.2.3** **RF3: Registrar las personas habilitadas para utilizar los servicios** 18](#_Toc134129621)

[**3.2.4** **RF4: Registrar una reserva** 20](#_Toc134129622)

[**3.2.5** **RF5: Cancelar una reserva** 23](#_Toc134129623)

[**3.2.6** **RF6: Retirar una oferta de alojamiento** 27](#_Toc134129624)

[**3.2.7** **RF7: Registrar una reserva colectiva** 29](#_Toc134129625)

[**3.2.8** **RF8: Cancelar reserva colectiva** 33](#_Toc134129626)

[**3.2.9** **RF9: Deshabilitar una oferta de alojamiento** 35](#_Toc134129627)

[**3.2.10** **RF10: Rehabilitar un alojamiento deshabilitado** 42](#_Toc134129628)

[3.3 Implementación y resultado de los requisitos funcionales de consulta 44](#_Toc134129629)

[3.4 Pruebas de transacción Exitosas (Demo – pruebas de unicidad) 45](#_Toc134129630)

[3.4.1 Pruebas exitosas para el Requisito Funcional 7 45](#_Toc134129631)

[3.4.2 Pruebas exitosas para el Requisito Funcional 8 47](#_Toc134129632)

[3.4.3 Pruebas exitosas para el Requisito Funcional 9 50](#_Toc134129633)

[3.4.4 Pruebas exitosas para el Requisito Funcional 10 52](#_Toc134129634)

[3.5 Pruebas de transacción No-Exitosas (Demo – pruebas de unicidad) 53](#_Toc134129635)

[3.5.1 Pruebas No-Exitosas para el Requisito Funcional 7 53](#_Toc134129636)

[3.5.2 Pruebas No-Exitosas para el Requisito Funcional 8 54](#_Toc134129637)

[3.5.3 Pruebas No-Exitosas para el Requisito Funcional 9 55](#_Toc134129638)

[3.5.4 Pruebas No-Exitosas para el Requisito Funcional 10 56](#_Toc134129639)

# Análisis

Para el caso de estudio AlohAndes, se define un problema relacionado al mundo de negocio de las reservas, la hotelería y los servicios de estadía. De esta manera, es sumamente importante que clasifiquemos estas características que definen el negocio, antes de pasar al diseño de la arquitectura del negocio. Para esta iteración, fue necesario realizar unos cambios en el modelo conceptual y relacional para ajustar el proyecto al mundo del problema nuevo.

## Ajustes modelo conceptual (UML)

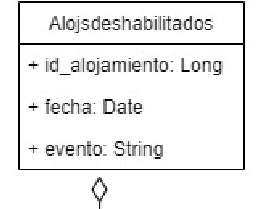
Como se mencionó anteriormente, el modelo conceptual tuvo que sufrir unas modificaciones para adecuarse a los nuevos requerimientos del negocio. En particular, para el nuevo requerimiento funcional de deshabilitar un alojamiento, fue necesario crear una nueva relación o tabla, como se quiera llamar, la cual se encargaría de almacenar los alojamientos deshabilitados temporalmente. Este cambio fue necesario ya que, de esta manera, es más fácil saber que alojamientos fueron deshabilitados, darle un id a ese proceso y también pedir una razón/evento externo por el cual se tuvo que inhabilitar temporalmente el alojamiento. Esta nueva tabla se referencia como Alojsdeshabilitados la cual se evidencia en la siguiente imagen (imagen 1-1). Finalmente, la imagen del modelo conceptual completa se encuentra posterior a la siguiente imagen, y representa la arquitectura manejada para desarrollar los requisitos de AlohAndes.

Imagen 1‑1

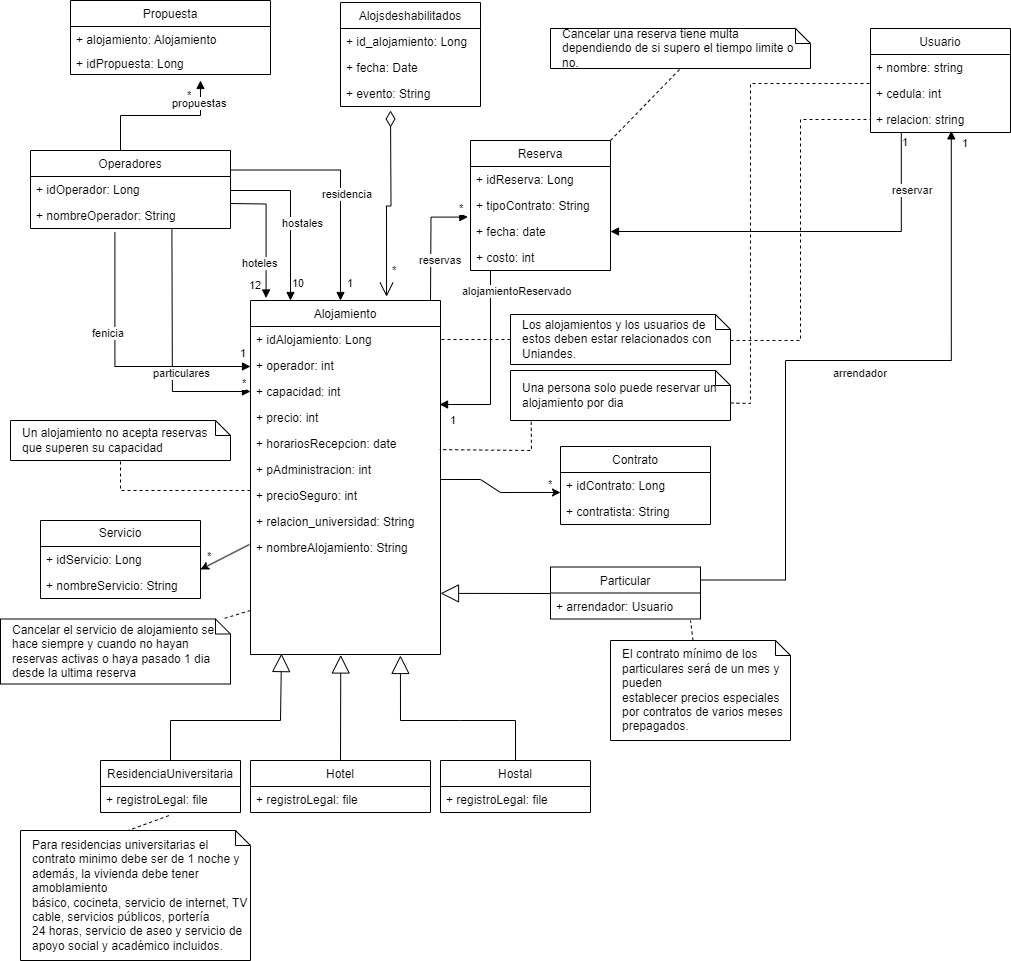


Imagen 1‑2

## Ajustes al modelo relacional (RM)

De igual manera que el modelo conceptual, el modelo relacional también tiene que sufrir cambios para adecuarse a los nuevos requerimientos. Así pues, la nueva tabla relación creada tiene que también ser representada en el modelo relacional actual. Antes de presentar el modelo relacional, el cliente puede estarse preguntando, ¿porque no hacer también una tabla para el nuevo requisito funcional de registrar una reserva colectiva? La respuesta es simple, y es que es posible en las instancias de reservas, en el atributo de tipo de contrato, añadir el id de la reserva masiva para así saber que fueron creadas colectivamente y poder moverlas de un lado a otro simplemente observando si en el atributo existe dicha característica y a que transacción pertenece. Sin mas que añadir, en la siguiente imagen (imagen 1-3) se presenta el modelo relacional con sus tablas nuevas y en la imagen posterior a esta, se encuentra específicamente la nueva relación creada (imagen 1-4).

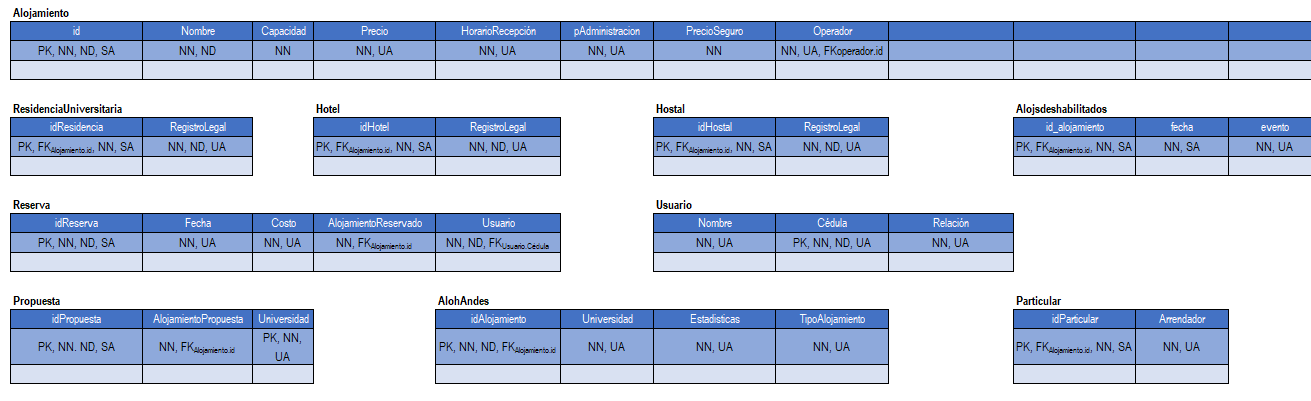


Imagen 1‑3

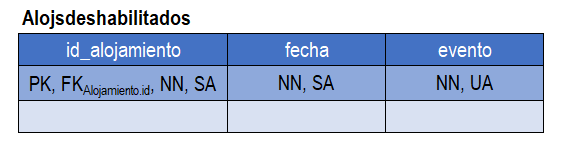


Imagen 1‑4

Finalmente, no esta de mas recordarle al cliente que esta imagen y estas tablas están con mayor resolución en el repositorio o en el proyecto mismo en la carpeta data.

# Diseño de la aplicación

En esta sección se desarrollará el análisis a groso modo de las decisiones que tienen que ser tomadas en el diseño para satisfacer los nuevos requisitos, las reglas de negocio y seguir asegurando la calidad del mismo. Sin embargo, es importante comenzar por definir las principales diferencias entre el diseño entregado para la iteración 2 y el diseño entregado para esta iteración 3. Lo anterior nos da un panorama mas claro para saber de donde partir y corregir los errores que nos dejaron los diseños pasados.

## Diferencia entre la entrega de la iteración 2 y la entrega de la iteración 3

Para mencionar los cambios mas relevantes entre el diseño entregado y el anterior, es necesario que consideremos, en principio, que se aspectos se concretaron del proyecto y cuales no, en la iteración pasada. Ahora bien, partir de este punto es generar una introspección respecto a lo entregado puesto lo realizado respecto a la iteración 2 la podemos considerar como un fracaso. Para empezar, el proyecto de software de la base de datos se completo en su 50% en dicha entrega. Lo anterior quiere decir que la aplicación no funcionaba, las clases en java estaban incompletas y aun había mucho código por realizar. Además de eso, no se realizaron los requisitos funcionales o de consulta. Todo esto dejo atrás una gran de carga para el diseño de esta iteración y de las modificaciones restantes para alcanzar los requisitos del proyecto.

No obstante, esta entrega es totalmente diferente. Debido a que, no solamente se completo al 100% el software de la base de datos, si no también, se completo al 100% los requisitos funcionales, no funcionales y de consulta. Claramente, esto implicó una ardua inversión de tiempo y de trabajo para realizar estos cambios. En principio, se completaron todas las clases necesarias para el funcionamiento del software y empezar a trabajar sobre los requisitos funcionales. Es decir, se completaron todas las clases VO y SQL de las clases que representan las tablas del modelo relacional. Lo anterior permite poder desarrollar la clase InterfazAlohandesApp que realiza las consultas, modificaciones, operaciones CRUD y requisitos funcionales, todo a través de la interfaz, que, en la anterior iteración, no fue realizada.

Con la interfaz diseñada, se pudo realizar los requisitos funcionales, la anotación de los cambios en el log de datanucleus y la verificación de las reglas de negocio. Las reglas de negocio, no se tuvieron en cuenta en la iteración 2, por ende, nunca fueron respetadas. Sin embargo, en esta iteración fueron tenidas en cuenta no solamente para las operaciones CRUD si no también para las operaciones funcionales del mundo de negocio. Ahora, cuando una regla de negocio es violada, se informa al usuario por medio del panel de la interfaz cual fue la razón y si la transacción pudo ser realizada o no, lo cual depende de la situación en particular. Finalmente, cabe recalcar que todas las implementaciones en el código que tengan que ver con los requisitos funcionales fueron anotados con la marca @RF + #requisito.

## Modelo BCNF y como se trataron las anomalías

De igual manera que en la iteración 1, el modelo relacional presentado se encuentra en la forma normal de Boyce-Codd debido a que cumple con las condiciones necesarias para evitar la redundancia de datos y la pérdida de integridad referencial. La forma normal de Boyce-Codd establece que, en una relación, cualquier dependencia funcional debe estar determinada por la clave de la relación en lugar de por cualquier otro atributo no clave. Esto significa que no debe haber atributos que estén funcionalmente determinados por otros atributos no clave en la misma relación. Si una relación cumple con esta condición, se considera que está en la forma normal de Boyce-Codd. Esto asegura que en el modelo relacional presentado sus datos almacenados en la relación sean coherentes y estén libres de anomalías en la actualización, inserción o eliminación de registros.

Ahora bien, es necesario explicar que si hubo sí anomalías y fallos cuando se realizaron diversas operaciones y pruebas con los datos de la aplicación. En concreto, se tuvo que lidiar con una anomalía muy especifica y presente. Dicha anomalía se generaba cuando se intentaba ejecutar un Query en las clases SQL donde se esperaba que el valor de retorno del Query sea un int. Esto, ocurría en general en cualquier otra clase, donde si se casteaba un int o un long para ser usado por una operación crud que necesitaba específicamente un int, el sistema arrojaba el error que no existe forma de convertir un objeto Java.math.BigDecimal en int. No había forma de detectar el error puesto java castea los long’s a int de manera semiautomática.

¿Entonces, cual fue la solución a la anomalía? La solución fue un poco costosa pero valiosa para la problemática presentada. En particular, todas las tablas/relaciones que tenían atributos con tipos de datos int, tuvieron que ser cambiados a number. Dicha alternativa, implicaba que se cambie todo el SQL para la creación de tablas, pero, aseguraba que independientemente de si se generaba un valor en java como int o long para hacer manipular los valores de la base de datos, esto no afectaría en absoluto al resultado. Efectivamente, esta solución fue exitosa y permitió el avance del proyecto. Así pues, en la imagen presentada a continuación, (imagen 2-1), se evidencia como se realizó este cambio en las sentencias SQL.

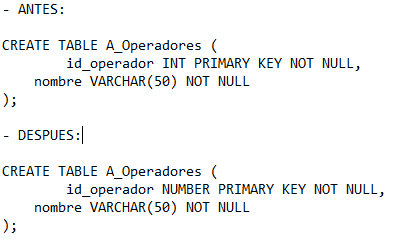


Imagen 2‑1

## Tablas generadas en la base de datos

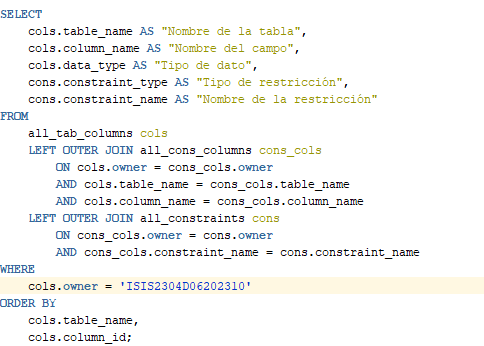
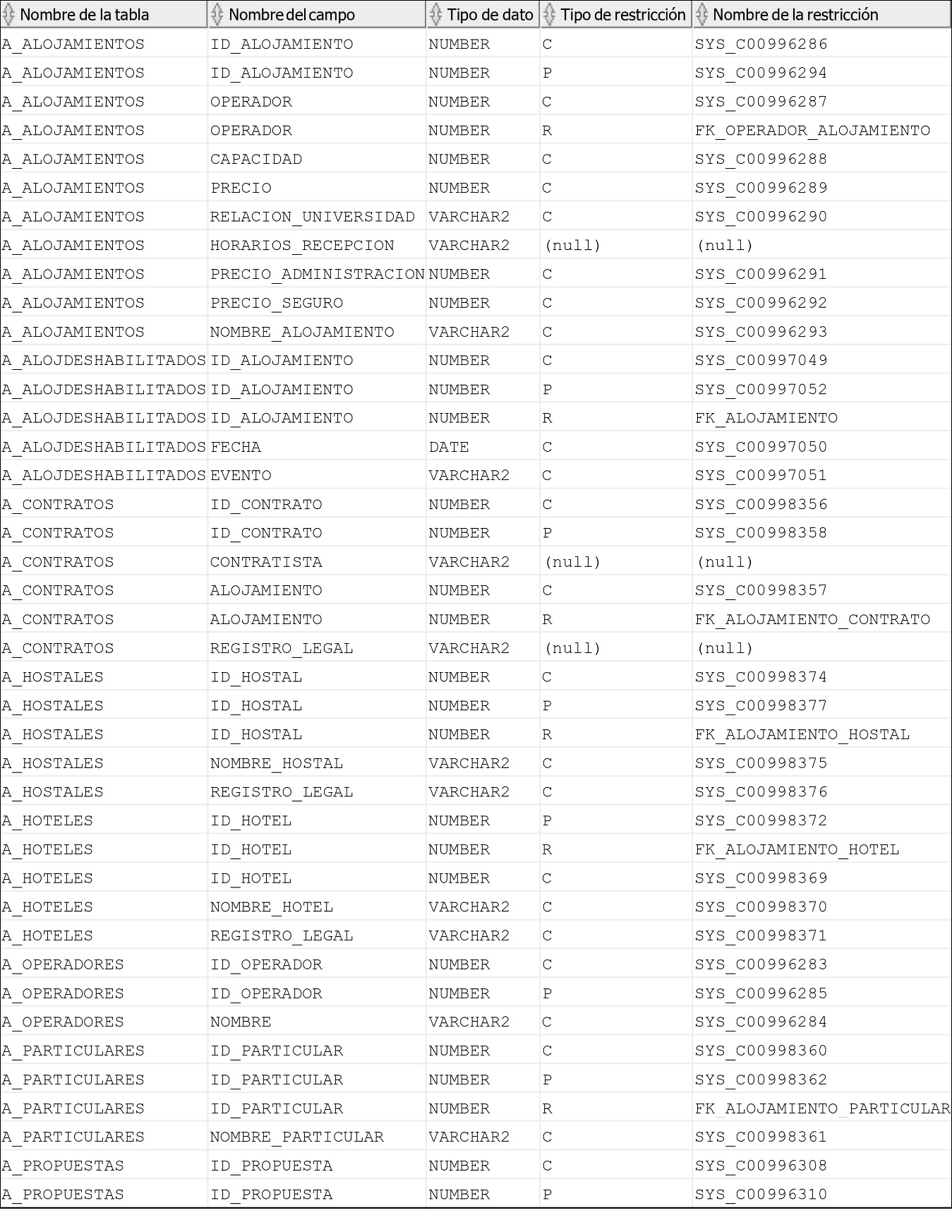
A continuación, se presenta las tablas generadas en la base de datos, las cuales fueron generadas haciendo uso de los comandos SQL que se encuentran en el apartado de ‘*AlohandesCreacionTablas*’. De esta manera, esta muestra se hace con fines ilustrativos y para que el cliente evidencie que todas las restricciones son incluidas en el sistema, preservando la calidad del mismo. En la imagen siguiente (imagen 2-2) se encuentra cual fuel el comando SQL utilizado para la generación de las tablas con su información/restricciones como lo pide el enunciado. Finalmente, en la ultima imagen de esta sección 2.3, se encuentran las tablas con sus resultados generados por el comando SQL (imagen 2-3), cabe recalcar, que esta imagen se encuentra con mayor calidad en el archivo dentro del proyecto llamado *‘TablesAndConstraints.pdf’*.

Imagen 2‑2



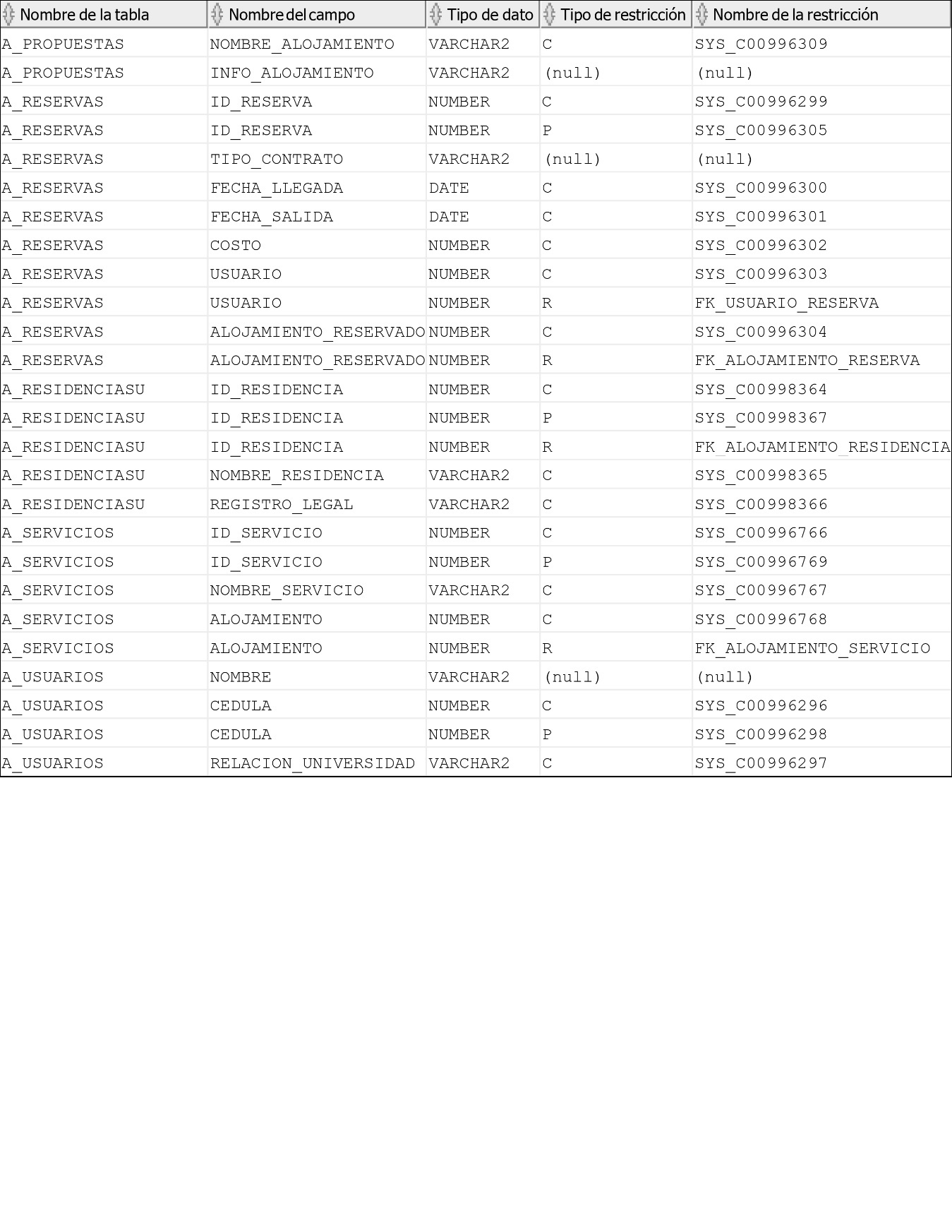


Imagen 2‑3

## Lógica de los nuevos requisitos y sus mecanismos ACID

La lógica de los nuevos requisitos recibe siempre como entrada el id de la reserva o el alojamiento a modificar y devuelve la secuencia de transacciones que hacen posible el desarrollo del requisito. Para el requisito funcional de registrar una reserva colectiva se recibe como parámetro la id del usuario, para satisfacer la privacidad, la id de la reserva masiva, las fechas de llegada y salida, el tipo de alojamiento deseado, los servicios que le gustaría tener y la cantidad de reservas deseadas. Ahora bien, esta lógica propone 3 pasos para desarrollar la transacción de manera exitosa. Primero, se filtran los alojamientos según el tipo de alojamiento deseado, esto retorna una lista filtrada de alojamientos 1. Segundo, se filtra entre los alojamientos del primer filtro, los que alojamientos que proveen los servicios requeridos por el usuario, esto nos devuelve el segundo filtrado. Tercero, se reparte la cantidad de reservas a generar entre los alojamientos del segundo filtrado y se consideran ciertas particularidades. Dichas particularidades, son tales que, por ejemplo, si en el alojamiento caben todas las reservas, usamos este alojamiento para realizar todas las reservas, de lo contrario, llenamos lo que podamos en ese alojamiento y el resto de reservas se las entregamos al siguiente alojamiento en el ciclo del tercer filtrado. Finalmente se devuelven todas las transacciones realizadas y las reservas confirmadas.

Para el requisito funcional de eliminar una reserva colectiva, se siguen los pasos a continuación. Como se genero un id de la reserva colectiva y este se almacena en un atributo de las tuplas de reserva, simplemente se busca todas las reservas colectivas con el id dado y se eliminan una a una con el requisito funcional para ello. Para el requisito funcional de deshabilitar un alojamiento se necesita la creación de una nueva tabla llamada Alojsdeshabilitados. La lógica que se sigue para completar este requisito con éxito, se basa en añadir la reserva que se quiere deshabilitar, con un motivo dado por parámetro, a la tabla de alojamientos deshabilitados. Posterior a ello, se extraen las reservas del alojamiento y se migran a otros alojamientos teniendo como prioridad a las reservas vigentes. Para ingresar las reservas a los alojamientos correspondientes, se usa el requisito funcional correspondiente a la creación de reservas. Por último, para el requisito funcional de rehabilitar un alojamiento deshabilitado, simplemente se sigue la lógica de eliminar la tupla de la tabla de alojamientos deshabilitados que corresponda al id del alojamiento ingresado por parámetro.

¿Sin embargo, como se asegura que esta lógica a implementar respeta las reglas ACID? Pues bien, en principio, la regla de ATOMICIDAD, se ve evidenciada en los requisitos funcionales cuando se evalúan las reglas de integridad y las subtransacciones de manera muy granular. Es decir, cada requisito va detallando paso a paso los mecanismos para llegar a la transacción global y en caso de haber alguna inmediatez, se informa al usuario que sucedió, si se logró la transacción o no y en que parte quedo. Lo anterior es evidenciable en la imagen 2.4 la cual representa un fragmento de la implementación del requisito funcional 9.



Imagen 2‑4

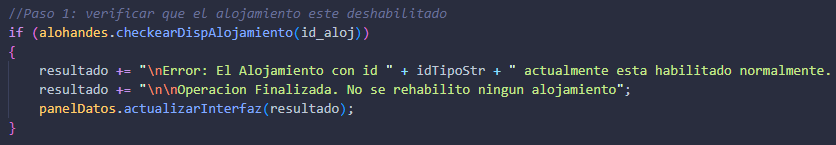
Ahora bien, la CONSISTENCIA se ve reflejada mediante el mecanismo de llevar la traza de las transacciones pasadas y las que se quieren realizar. Por ejemplo, si antes de rehabilitar un alojamiento se evidencia que este no esta deshabilitado, no se procede con la transacción puesto una se tiene que ser consistente con un valor que no ha sido modificada en transacciones pasadas. Dicho ejemplo se puede ver en la imagen 2.5, la cual corresponde a un fragmento de la implementación del requisito funcional 10.

Imagen 2‑5

Para garantizar el AISLAMIENTO, se maneja cada transacción como una transacción aislada iniciando la tx en el software de la aplicación la cual se realizará o no dependiendo de si hubo un error o de si se cumplen las reglas de negocio. Por ejemplo, para las reservas colectivas se trata cada subtransaccion como si fuera una única transacción aislada mediante el uso de requisitos funcionales ya diseñados para esos casos específicos. La imagen 2.6 es un fragmento del requisito funcional 8 donde se evidencia el aislamiento de cada subtransaccion independiente al resultado.

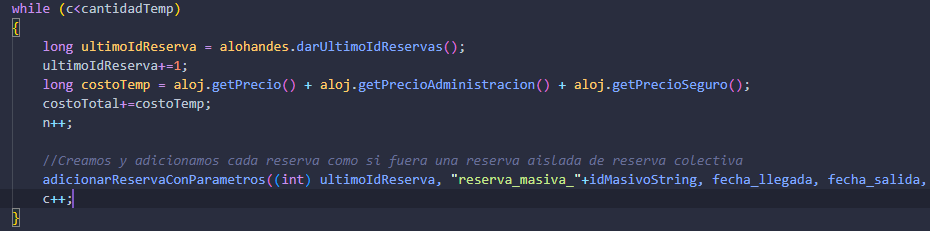


Imagen 2‑6

Para poder garantizar la DURABILIDAD se manejó la instancia del PersistenceManagerpara que las transacciones perduren en la base de datos si fueron completadas con éxito. Además, esto es comprobable mediante la aplicación de SQL Developer, puesto que cada transacción que se genera en la interfaz de AlohAndes diseñada se ve reflejada instantáneamente en la conexión del SQL Developer dentro de la base de datos y viceversa.

¿Finalmente, como se garantiza el nuevo requisito no funcional de **TRANSACCIONALIDAD**? Esta característica se respeta debido a que, como se mencionó anteriormente, cada transacción o subtransaccion inicia una instancia en java de una transacción tx, la cual puede hacer commit o rollback de la información que se este generando. De esta manera, las transacciones se inicializan, ejecutan sus sentencias SQL, evalúan sus requerimientos y las reglas de negocio, ya que así, si todo esta en orden, la transacción puede ser comiteada hacia la base de datos o de lo contrario puede ser deshecha, generando en todo momento, los logs en la bitácora de todo lo que se hace en cada paso de la transacción. La imagen 2.7 representa estas inicializaciones y traza de la bitácora en la aplicación. Además, para cada transacción antes de ejecutar su comando SQL, seteamos el **NIVEL DE AISLAMIENTO** de la transacción a **READ COMMITED**, para así leer datos que han sido confirmados de otras transacciones. Lo anterior nos evita hacer lecturas no definitivas de otras transacciones y permitir concurrencia. Incluso, no nos importa que haya fantasmas o lecturas inconsistentes puesto las transacciones no hacen consultas intermedias, es decir, inician con un valor y nunca toman otro valor de otra transacción. Es decir, nos permitimos un poco de inconsistencia siempre y cuando las transacciones inicialicen de manera prudente. Dicha implementación se puede ver en la imagen 2.8.



Imagen 2‑7

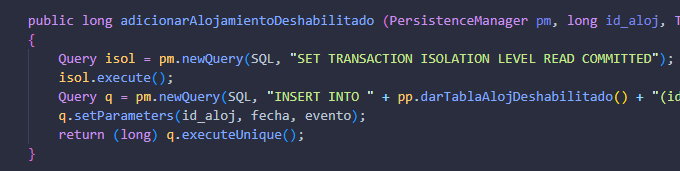


Imagen 2‑8

# Construcción de la aplicación

## Funcionamiento de la interfaz de AlohAndes para realizar transacciones

La interfaz diseñada, basada en la aplicación de referencia Parranderos, se compone de 5 botones para desarrollar las transacciones deseadas y de 3 ventanas para el mantenimiento de la aplicación. Así pues, los 5 botones hacen referencia a 5 relaciones/clases del modelo presentado. Sin embargo, hay muchas mas relaciones, pero se tomó esta decisión para no sobrecargar la interfaz con funcionalidades para todas las clases, si no para las mas vitales/fundamentales para el desarrollo de los requisitos funcionales y no funcionales. Como se puede ver en la imagen 3.1, las relaciones plasmadas en la interfaz son las de: Operador, Propuestas, Usuarios, Reservas y Alojamientos. De manera contigua, nos encontramos el resto de opciones de la interfaz: Mantenimiento, Documentación e información acerca de la aplicación y las licencias. Cuando se presiona un botón, se despliega las funcionalidades en su mayoría crud, mas las funcionalidades particulares que tengan que ver con algún requisito funcional. Por ejemplo, para la clase Operador, como se evidencia en la imagen 3.2, solo tenemos operaciones crud en su interfaz. Sin embargo, para la clase Alojamiento, como se puede observar en la imagen 3.3, tenemos muchas más opciones para el uso de requisitos funcionales que se ven implicados en la lógica del negocio.

Cabe recalcar, que en el apartado mantenimiento se encuentran funcionalidades para limpiar las bitácoras, mostrar el log del datanucleus o de la aplicación, y así evidenciar la traza de las transacciones según se van realizado. Claro está, cada apartado de las clases de la interfaz, desarrolla operaciones crud simples que usan la lógica desarrollada en el paquete src del proyecto, donde se velan por todas las reglas de negocios y particularidades del mundo del problema. No se explicará a profundidad cada operación crud, no obstante, el cliente puede revisar la clase InterfazAlohandesApp donde se hacen las invocaciones a todas las clases para desarrollar la transacción con sus características ACID. Además, la interfaz incluye un panel donde se imprime la información del resultado de la transacción al cliente y de sus estados intermedios (ver Imagen 3.3). Finalmente, las operaciones en la interfaz para realizar los requisitos funcionales de la aplicación serán explicados mas adelante en este documento.

Imagen 3‑1



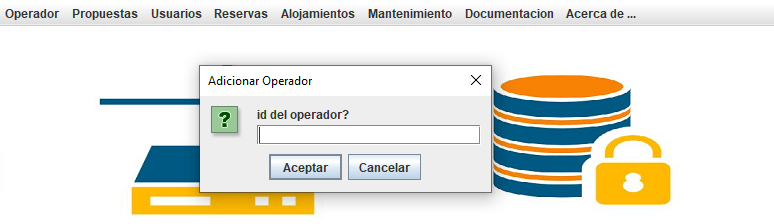


Imagen 3‑2

Imagen 3‑3

## Implementación y resultado de los requisitos funcionales

### **RF1: Registrar los operadores de alojamiento para AlohAndes**

Esta operación, siendo una operación de crud simple, necesito de la clase Operadores.java y sus respectivas clases para operaciones SQL: VOOperadores y SQLOperadores. Asi pues, el código usado en la interfaz de la aplicación para invocar e inicializar la transacción es el siguiente:

/\*\*

*\* @RF1*

*\* Adiciona un Operador con la información dada por el usuario*

*\* Se crea una nueva tupla de Operador en la base de datos, si un Operador con ese nombre no existía*

\*/

    public void adicionarOperador( )

    {

        try

        {

            String idString = JOptionPane.showInputDialog (this, "id del operador?", "Adicionar Operador", JOptionPane.QUESTION\_MESSAGE);

            int id = Integer.parseInt(idString);

            String nombreTipo = JOptionPane.showInputDialog (this, "Nombre del operador?", "Adicionar Operador", JOptionPane.QUESTION\_MESSAGE);

            if (nombreTipo != null)

            {

                VOOperador tb = alohandes.adicionarOperador(id, nombreTipo);

                if (tb == null)

                {

                    throw new Exception ("No se pudo crear un Operador con nombre: " + nombreTipo);

                }

                String resultado = "En adicionarOperador\n\n";

                resultado += "Operador adicionado exitosamente: " + tb;

                resultado += "\n Operación terminada";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

                panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

            }

        }

        catch (Exception e)

        {

// *e.printStackTrace();*

            String resultado = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultado);

        }

    }

En el código anterior, se puede evidenciar que se pide por parámetro al usuario la información del operador: id y nombre del operador. De esta manera, se inicializa la transacción y se completa con la operación SQL a continuación:

Query q = pm.newQuery(SQL, "INSERT INTO " + pp.darTablaOperador () + "(id\_operador, nombre) values (?, ?)");

        q.setParameters(id\_Operador, nombre\_Operador);

### **RF2: Registrar propuestas de Alojamientos para AlohAndes**

De igual manera, esta es un operación CRUD simple, la cual necesita de la clase Propuesta.java y sus respectivas clases para operaciones SQL: SQLPropuesta y VOPropuesta. En el código a continuación, se puede ver la implementación en la interfaz de la aplicación que hace las invocaciones a las respectivas clases para inicializar la transacción debida:

/\*\*

*\* @RF2*

*\* Adiciona una Propuesta con la información dada por el usuario*

*\* Se crea una nueva tupla Propuesta en la base de datos, si una Propuesta con ese nombre no existía*

\*/

    public void adicionarPropuesta( )

    {

        try

        {

            String idString = JOptionPane.showInputDialog (this, "id de la propuesta?", "Adicionar Propuesta", JOptionPane.QUESTION\_MESSAGE);

            int id = Integer.parseInt(idString);

            String nombreTipo = JOptionPane.showInputDialog (this, "Nombre del alojamiento a proponer?", "Adicionar Propuesta", JOptionPane.QUESTION\_MESSAGE);

            String info = JOptionPane.showInputDialog (this, "Informacion del alojamiento?", "Adicionar Propuesta", JOptionPane.QUESTION\_MESSAGE);

            if (nombreTipo != null)

            {

                VOPropuesta tb = alohandes.adicionarPropuesta(id, nombreTipo, info);

                if (tb == null)

                {

                    throw new Exception ("No se pudo crear una Propuesta con nombre: " + nombreTipo);

                }

                String resultado = "En adicionarPropuesta\n\n";

                resultado += "Propuesta adicionada exitosamente: " + tb;

                resultado += "\n Operación terminada";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

                panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

            }

        }

        catch (Exception e)

        {

// *e.printStackTrace();*

            String resultado = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultado);

        }

    }

En el código anterior, se puede evidenciar que se pide por parámetro al usuario la información de la propuesta: id de la propuesta, nombre e información del alojamiento a ofrecer. De esta manera, se inicializa la transacción y se completa con la operación SQL a continuación:

Query q = pm.newQuery(SQL, "INSERT INTO " + pp.darTablaPropuesta () + "(id\_propuesta, nombre\_alojamiento, info\_alojamiento) values (?, ?, ?)");

        q.setParameters(id\_Propuesta, nombre\_alojamiento, info\_alojamiento);

### **RF3: Registrar las personas habilitadas para utilizar los servicios**

Nuevamente, esta es un operación CRUD simple, la cual necesita de la clase Usuario.java y sus respectivas clases para operaciones SQL: SQLUsuario y VOUsuario. En el código a continuación, se puede ver la implementación en la interfaz de la aplicación que hace las invocaciones a las respectivas clases para inicializar la transacción debida:

/\*\*

*\* @RF3*

*\* Adiciona una Usuario con la información dada por el usuario*

*\* Se crea una nueva tupla del Usuario en la base de datos*

\*/

    public void adicionarUsuario( )

    {

        try

        {

            String nombre = JOptionPane.showInputDialog (this, "nombre?", "Adicionar Nombre", JOptionPane.QUESTION\_MESSAGE);

            String idString = JOptionPane.showInputDialog (this, "Cedula?", "Adicionar Nombre", JOptionPane.QUESTION\_MESSAGE);

            int id = Integer.parseInt(idString);

            String relacionU = JOptionPane.showInputDialog (this, "Relacion con la universidad?", "Adicionar Nombre", JOptionPane.QUESTION\_MESSAGE);

            if (nombre != null)

            {

                VOUsuario tb = alohandes.adicionarUsuario(nombre, id, relacionU);

                if (tb == null)

                {

                    throw new Exception ("No se pudo crear un usuario con nombre: " + nombre);

                }

                String resultado = "En adicionarUsuario\n\n";

                resultado += "Usuario adicionada exitosamente: " + tb;

                resultado += "\n Operación terminada";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

                panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

            }

        }

        catch (Exception e)

        {

            e.printStackTrace();

            String resultado = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultado);

        }

    }

En el código anterior, se puede evidenciar que se pide por parámetro al usuario la información de la propuesta: nombre, cedula, la cual sirve como identificador inequívoco del usuario, y su relación con la universidad. De no tener una relación con la universidad, se viola la regla de negocio y la transacción no se realiza. Puesto solo las personas que están relacionadas de alguna manera con la universidad pueden hacer uso de estos servicios. De esta manera, se inicializa la transacción y se completa con la operación SQL a continuación:

Query q = pm.newQuery(SQL, "INSERT INTO " + pp.darTablaUsuario () + "(nombre, cedula, relacion\_universidad) values (?, ?, ?)");

        q.setParameters(nombre, cedula, relacion\_universidad);

### **RF4: Registrar una reserva**

Esta es una operación CRUD más compleja debido a que se deben considerar varios elementos de tablas exteriores para encontrar un alojamiento que supla las necesidades del cliente. Siendo así, se necesita de la clase Reserva.java, Alojamiento.java y sus respectivas clases para operaciones SQL: SQLUsuario, VOUsuario, SQLAlojamiento y VOAlojamiento. Antes de realizar la transacción se deben verificar varias reglas de negocio. Entre estas se debe verificar si el Alojamiento a reservas está actualmente habilitado o no, puesto ahora contamos con alojamientos que pueden estar deshabilitados. Otra regla que se debe respetar, es que el usuario no puede hacer más de una reserva en un mismo día. Una última regla verifica que la reserva no sobrepase el aforo total permitido por el Alojamiento. Una vez todo se haya verificado se procede a realizar la transacción. En el código a continuación, se puede ver la implementación en la interfaz de la aplicación que hace las invocaciones a las respectivas clases para inicializar la transacción debida:

/\*\*

*\* @RF4*

*\* Adiciona una Usuario con la información dada por el usuario*

*\* Se crea una nueva tupla del Usuario en la base de datos*

\*/

    public void adicionarReserva()

    {

        try

        {

            String resultado = "En adicionarReserva\n\n";

            String idString = JOptionPane.showInputDialog (this, "id de la reserva?", "Adicionar Reserva", JOptionPane.QUESTION\_MESSAGE);

            int id = Integer.parseInt(idString);

            String tipo\_contrato = JOptionPane.showInputDialog (this, "tipo de contrato?", "Adicionar Reserva", JOptionPane.QUESTION\_MESSAGE);

            String fecha\_llegada = JOptionPane.showInputDialog (this, "fecha de llegada 'ej: 2023-05-01 12:30:45'?", "Adicionar Reserva", JOptionPane.QUESTION\_MESSAGE);

            String fecha\_salida = JOptionPane.showInputDialog (this, "fecha de salida 'ej: 2023-05-01 12:30:45'?", "Adicionar Reserva", JOptionPane.QUESTION\_MESSAGE);

            String usuario = JOptionPane.showInputDialog (this, "cedula del usuario?", "Adicionar Reserva", JOptionPane.QUESTION\_MESSAGE);

            int cedula\_int = Integer.parseInt(usuario);

            String alojamiento\_reservado = JOptionPane.showInputDialog (this, "id del alojamiento reservado?", "Adicionar Reserva", JOptionPane.QUESTION\_MESSAGE);

            int id\_aloj = Integer.parseInt(alojamiento\_reservado);

            //*[Verificacion regla de negocio]: Ver si el alojamiento esta habilitado o no*

            if (!alohandes.checkearDispAlojamiento(id\_aloj))

            {

                resultado += "Error de integridad: el alojamiento con id '"+id\_aloj+"' no esta habilitado actualmente!";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

            //*[Verificacion regla de negocio]: el usuario no puede reservar mas de 1 vez en un dia*

            Usuario user = alohandes.darUsuarioPorId(cedula\_int);

            boolean cent = false;

            List<Reserva> reservas = alohandes.darReservas();

            int i = 0;

            while (i<reservas.size() && cent==false)

            {

                if (reservas.get(i).getUsuario()==cedula\_int)

                {

                    cent = true;

                    resultado += ("\nError: El usuario '"+ user.getNombre()+"' ya ha hecho una reserva en el dia actual.");

                    resultado += "\n Operación terminada\n";

                    panelDatos.actualizarInterfaz(resultado);

                }

                i++;

            }

            //*[Verificacion regla de negocio]: la reserva no puede superar la capacidad del alojamiento*

            Alojamiento aloj = alohandes.darAlojamientoPorId(id\_aloj);

            if ((aloj.getCapacidad()-1)<0)

            {

                cent = true;

                resultado += ("\nError: La reserva actual supera el aforo maximo del alojamiento.");

                resultado += "\n Operación terminada\n";

                panelDatos.actualizarInterfaz(resultado);

            }

            if (idString != null && cent==false)

            {

                long costoTotal = aloj.getPrecio() + aloj.getPrecioAdministracion() + aloj.getPrecioSeguro();

                VOReserva tb = alohandes.adicionarReserva(id, tipo\_contrato, fecha\_llegada, fecha\_salida, (int)costoTotal, cedula\_int, id\_aloj);

                if (tb == null)

                {

                    throw new Exception ("No se pudo crear una reserva con id: " + idString);

                }

                //*Disminuir el numero de la capacidad del alojamiento (-1)*

                long capacidad = aloj.getCapacidad() - 1;

                alohandes.actualizarCapacidadAlojamiento(id\_aloj, capacidad);

                resultado += "\nReserva adicionada exitosamente: " + tb;

                resultado += "\nCosto total de la reserva: " + costoTotal+"000 COP";

                resultado += "\nOperación terminada\n";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

                panelDatos.actualizarInterfaz(resultado);

            }

        }

        }

        catch (Exception e)

        {

            e.printStackTrace();

            String resultado = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultado);

        }

    }

Esta transacción solicita varia información al usuario para ser completada: tipo de contrato de la reserva, fechas de llegadas y salida, cedula del usuario de AlohAndes e id del alojamiento reservado. Claro está, si alguno de estos parámetros no es el correcto, la transacción no se realiza. De lo contrario, si la transacción finaliza, se retorna el costo total de la reserva. El comando SQL para realizar la reserva se puede ver en el código a continuación:

Query q = pm.newQuery(SQL, "INSERT INTO " + pp.darTablaReserva () + "(id\_reserva, tipo\_contrato, fecha\_llegada, fecha\_salida, costo, usuario, alojamiento\_reservado) values (?, ?, ?, ?, ?, ?, ?)");

        q.setParameters(id\_Reserva,  tipo\_contrato,  fecha\_llegada, fecha\_salida, costo, usuario, alojamiento\_reservado);

### **RF5: Cancelar una reserva**

Este requisito funcional es una operación CRUD compleja, debido a que se tienen que hacer varias verificaciones de integridad con tablas externas y otros elementos del negocio. En la implementación, primero se obtiene de la tabla de Reservas la hora en la cual la reserva fuer creada. Con este valor, en milisegundos, se procede a comparar la fecha de creación de la reserva y la fecha actual en ese momento, con los tiempos de llegada y salida de la misma. Así pues, dependiendo en qué momento fue creada la reserva, el tiempo límite, y cual es la fecha actual vigente, se toma la decisión de cual será la multa a aplicar para el usuario. Una vez esta multa ha sido calculada se procede a inicializar la transacción para eliminar la reserva de la base de datos. La implementación antes explicada, se encuentra a continuación:

/\*\*

*\* @RF5*

*\* Borra de la base de datos la Reserva con el identificador dado por el usuario*

*\* la cancelacion tiene una multa segun el tiempo de cancelacion*

\*/

    public void eliminarReservaPorId( )

    {

        try

        {

            String idTipoStr = JOptionPane.showInputDialog (this, "Id de la reserva?", "Borrar Reserva por Id", JOptionPane.QUESTION\_MESSAGE);

            int id  = Integer.parseInt(idTipoStr);

            if (idTipoStr != null)

            {

                String resultado = "En eliminar Reserva\n\n";

                long tbEliminados = 0;

                //*Reserva a eliminar*

                Reserva res = alohandes.darReservaPorId(id);

                //*HORA DE CREACION DE LA RESERVA*

                Timestamp fecha\_creacion\_timestamp = res.getfechaCreacion();

                SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

                // *Convertir el objeto Timestamp a una cadena de texto con el formato especificado*

                String fechaHoraCreacionString = dateFormat.format(fecha\_creacion\_timestamp);

                resultado += ("\nHora y fecha de la creacion de la reserva: "+fechaHoraCreacionString);

                // *Crear objeto LocalDateTime desde la cadena de texto*

                DateTimeFormatter formatter0 = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

                LocalDateTime fechaHora0 = LocalDateTime.parse(fechaHoraCreacionString, formatter0);

                // *Crear objeto Instant a partir del LocalDateTime*

                Instant instant0 = fechaHora0.atZone(ZoneId.systemDefault()).toInstant();

                // *Obtener el timestamp en milisegundos*

                long timestampMilisegundos0 = instant0.toEpochMilli();

                //*HORA LLEGADA*

                // *Crear objeto SimpleDateFormat con el formato deseado*

                // *Convertir el objeto Timestamp a una cadena de texto con el formato especificado*

                String fechaHoraString = dateFormat.format(res.getFechaLlegada());

                resultado += ("\nHora y fecha de la llegada a la reserva: "+fechaHoraString);

                // *Crear objeto LocalDateTime desde la cadena de texto*

                DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

                LocalDateTime fechaHora = LocalDateTime.parse(fechaHoraString, formatter);

                // *Crear objeto Instant a partir del LocalDateTime*

                Instant instant = fechaHora.atZone(ZoneId.systemDefault()).toInstant();

                // *Obtener el timestamp en milisegundos*

                long timestampMilisegundos = instant.toEpochMilli();

                //*HORA ACTUAL*

                LocalDateTime fechaActual = LocalDateTime.now();

                DateTimeFormatter formato = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

                String fechaHoraStringActual = fechaActual.format(formato);

                resultado += ("\nHora y fecha actual: "+fechaHoraStringActual);

                LocalDateTime fechaHora1 = LocalDateTime.parse(fechaHoraStringActual, formatter);

                // *Crear objeto Instant a partir del LocalDateTime*

                Instant instant1 = fechaHora1.atZone(ZoneId.systemDefault()).toInstant();

                // *Obtener el timestamp en milisegundos*

                long timestampMilisegundosActual = instant1.toEpochMilli();

                //*TIEMPO QUE HA PASADO DESDE LA CREACION DE LA RESERVA*

                long dias\_creacion = timestampMilisegundosActual - timestampMilisegundos0;

                //*VERIFICACION DEL TIEMPO Y TOMA DE DECISION*

                // *Calcular el tiempo total en milisegundos para 3 días*

                long tresDiasEnMilisegundos = 3 \* 24 \* 60 \* 60 \* 1000L;

                // *Calcular el tiempo total en milisegundos para 1 día*

                long unDiaEnMilisegundos = 24 \* 60 \* 60 \* 1000L;

                long fechaLlegadaMenos1dia = timestampMilisegundos-unDiaEnMilisegundos;

                Alojamiento aloj = alohandes.darAlojamientoPorId(res.getAlojamientoReservado());

                if (dias\_creacion<tresDiasEnMilisegundos)

                    {

                        double multa = 0.1\*(aloj.getPrecio() + aloj.getPrecioAdministracion() + aloj.getPrecioSeguro());

                        resultado += ("\nHan pasado menos de 3 dias desde la creacion de la reserva. La multa es el 10% del costo = "+multa+"000 COP\nSe procede a eliminar la reserva con id: "+id+"\n\n");

                        tbEliminados += alohandes.eliminarReservaPorId(id);

                        //*Liberamos un cupo en la capacidad del alojamiento (+1)*

                        long capacidad = aloj.getCapacidad() + 1;

                        alohandes.actualizarCapacidadAlojamiento(aloj.getIdAlojamiento(), capacidad);

                    }

                else if (tresDiasEnMilisegundos<timestampMilisegundosActual && timestampMilisegundosActual<fechaLlegadaMenos1dia)

                    {

                        double multa = 0.3\*(aloj.getPrecio() + aloj.getPrecioAdministracion() + aloj.getPrecioSeguro());

                        resultado += ("\nLa cancelacion esta entre la fecha limite y 1 dia antes de la fecha de llegada, la multa es del 30% = "+multa+"000 COP\nSe procede a eliminar la reserva con id: "+id+"\n\n");

                        tbEliminados += alohandes.eliminarReservaPorId(id);

                        //*Liberamos un cupo en la capacidad del alojamiento (+1)*

                        long capacidad = aloj.getCapacidad() + 1;

                        alohandes.actualizarCapacidadAlojamiento(aloj.getIdAlojamiento(), capacidad);

                    }

                else

                    {

                        double multa = 0.5\*(aloj.getPrecio() + aloj.getPrecioAdministracion() + aloj.getPrecioSeguro());

                        resultado += ("\nHa pasado algun tiempo desde la fecha de llegada, por ende, la multa es del 50% = "+multa+"000 COP\nSe procede a eliminar la reserva con id: "+id+"\n\n");

                        tbEliminados += alohandes.eliminarReservaPorId(id);

                        //*Liberamos un cupo en la capacidad del alojamiento (+1)*

                        long capacidad = aloj.getCapacidad() + 1;

                        alohandes.actualizarCapacidadAlojamiento(aloj.getIdAlojamiento(), capacidad);

                    }

                resultado += "\n"+tbEliminados + " Reservas eliminadas\n";

                resultado += "\n Operación terminada";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

                panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

            }

Finalmente, después de haber hecho todas las verificaciones de negocio e integridad, se utiliza la siguiente secuencia SQL para eliminar la reserva de la base de datos e indicarle al usuario que la transacción se completo y cual es su multa correspondiente. El comando SQL generado se puede ver a continuación:

        Query q = pm.newQuery(SQL, "DELETE FROM " + pp.darTablaReserva () + " WHERE id\_reserva = ?");

        q.setParameters(idReserva);

### **RF6: Retirar una oferta de alojamiento**

Este requisito funcional es una operación CRUD compleja, debido a que se deber ciertas verificaciones de integridad y de negocio antes de proceder con la transacción de eliminación. Primero, se verifica que el id del alojamiento exista. Segundo se verifica que el alojamiento no tenga reservas activas, de lo contrario será deshabilitado de recibir reservas a partir de la fecha actual. Si no tiene reservas activas puede ser eliminado de la base de datos sin problemas. El código de esta implementación se encuentra a continuación:

/\*\*

*\* @RF6*

*\* Borra de la base de datos el Alojamiento con el identificador dado po el usuario*

*\* Se tiene que cumplir todas las condiciones de las reglas de negocio para ser eliminado*

\*/

    public void eliminarAlojamientoPorNombre( )

    {

        try

        {

            String nombre = JOptionPane.showInputDialog (this, "Nombre del alojamiento?", "Borrar Alojamiento por nombre", JOptionPane.QUESTION\_MESSAGE);

            if (nombre != null)

            {

                List<Alojamiento> aloj = alohandes.darAlojamientoPorNombre(nombre);

                //*verificar que el id exista*

                long id = aloj.get(0).getIdAlojamiento();

                List<Reserva> reservs = alohandes.darReservas();

                boolean cent = false;

                for (Reserva rev:reservs){

                    if (rev.getAlojamientoReservado()==id){

                        panelDatos.actualizarInterfaz("Error: Hay reservas activas para el alojamiento!");

                        cent = true;

                    }

                }

                if (cent == false){

                    long tbEliminados = alohandes.eliminarAlojamiento(nombre);

                    String resultado = "En eliminar Alojamiento\n\n";

                    resultado += tbEliminados + " Alojamientos eliminados\n";

                    resultado += "\n Operación terminada";

                    panelDatos.actualizarInterfaz(resultado);

                }

            }

            else

            {

                panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

            }

        }

        catch (Exception e)

        {

// *e.printStackTrace();*

            String resultado = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultado);

        }

    }

Finalmente, si el alojamiento se puede eliminar, se remueve de la base de datos con el siguiente comando SQL:

Query q = pm.newQuery(SQL, "DELETE FROM " + pp.darTablaAlojamiento () + " WHERE nombre\_alojamiento = ?");

            q.setParameters(nombreAlojamiento);

### **RF7: Registrar una reserva colectiva**

Esta es una operación masiva que necesita de los requisitos funcionales antes diseñados para el uso aislado de cada subtransaccion como si fuera una transacción única. Primero se verifica que el usuario sea un usuario registrado en la base de datos para continuar con la transacción. Segundo, se pide la información de cuando las fechas de la reserva, el tipo de alojamiento y de los servicios que le gustaría tener. Lo anterior se hace para proceder con el filtrado de los alojamientos que cumplen los requisitos de la reserva colectiva. En principio, se filtran los alojamientos de la base de datos que encajen en el tipo de alojamiento que el usuario esta buscando. Posterior a esto, se filtra únicamente los alojamientos cuyo conjunto de servicios contenga al subconjunto de los servicios requeridos, esto se hace con la operación booleana de java.util.containsAll. Por último, se reparte la cantidad de reservas entre los alojamientos filtrados del anterior paso y se generan las reservas en cada uno según su capacidad. Las reservas son generadas individualmente una por una mediante el requisito funcional de registrar una reserva. Sin embargo, todas las nuevas reservas tienen en su tipo de contrato un comentario que indica que fueron una reserva colectiva + la id con la cual la reserva colectiva fue creada. De esta manera, la implementación de lo antes dicho se presenta a continuación:

/\*\*

*\* @RF7*

*\* Adiciona una reserva colectiva con la información dada por el usuario*

*\* Se genera todas las reservas con la subtransaccion de insertar reservas, @RF4.*

\*/

    public void registrarReservaColectiva( )

    {

        long costoTotal = 0;

        int n = 0;

        try

        {

            String resultado = "En Generar Reserva Colectiva\n\n";

            String cedulaString = JOptionPane.showInputDialog (this, "Indique su id de usuario (Esta operacion solo la pueden realizar usuarios del sistema): ", "Registrar Reserva Colectiva", JOptionPane.QUESTION\_MESSAGE);

            int cedula = Integer.parseInt(cedulaString);

            String idMasivoString = JOptionPane.showInputDialog (this, "Asigne un id a la reserva masiva (esto servira como identificador para cuando se quiera eliminar la reserva colectiva): ", "Registrar Reserva Colectiva", JOptionPane.QUESTION\_MESSAGE);

            String fecha\_llegada = JOptionPane.showInputDialog (this, "fecha de llegada 'ej: 2023-05-01 12:30:45'?", "Adicionar Reserva", JOptionPane.QUESTION\_MESSAGE);

            String fecha\_salida = JOptionPane.showInputDialog (this, "fecha de salida 'ej: 2023-05-01 12:30:45'?", "Adicionar Reserva", JOptionPane.QUESTION\_MESSAGE);

            String tipo\_alojamiento = JOptionPane.showInputDialog (this, "Indique el tipo de alojamiento deseado (Ej: habitacion sencilla, habitacion multiple, vivienda universitaria, hotel, etc...): ", "Registrar Reserva Colectiva", JOptionPane.QUESTION\_MESSAGE);

            String serviciosString = JOptionPane.showInputDialog (this, "Indique que servicios le gustaria tener, separados por coma (Ej: cocineta,TV,piscina, etc.)", "Registrar Reserva Colectiva", JOptionPane.QUESTION\_MESSAGE);

            String cantidadString = JOptionPane.showInputDialog (this, "Indique la cantidad de reservas deseada:", "Registrar Reserva Colectiva", JOptionPane.QUESTION\_MESSAGE);

            int cantidad = Integer.parseInt(cantidadString);

            if (cantidadString != null)

            {

                //*Paso 1: Filtrar los alojamientos segun el tipo de alojamiento*

                List<Alojamiento> allAlojamientos = alohandes.darAlojamientos();

                List<Alojamiento> alojamientosFiltro1 = new ArrayList<Alojamiento>();

                for (Alojamiento aloj: allAlojamientos)

                {

                    if (aloj.getNombreAlojamiento().equals(tipo\_alojamiento))

                    {

                        alojamientosFiltro1.add(aloj);

                    }

                }

                //*Paso 2: Filtrar los alojamientos encontrados segun los servicios requeridos*

                List<String> serviciosLista = Arrays.asList(serviciosString.split(","));

                List<Alojamiento> alojamientosFiltro2 = new ArrayList<Alojamiento>();

                for (Alojamiento aloj:alojamientosFiltro1)

                {

                    //*Extraemos los servicios del alojamiento actual*

                    List<Servicio> servicios = alohandes.darServiciosDeUnAlojamiento(aloj.getIdAlojamiento());

                    List<String> serviciosTemp = new ArrayList<String>();

                    for (Servicio servicioTemp:servicios)

                    {

                        serviciosTemp.add(servicioTemp.getNombreServicio());

                    }

                    //*Si los servicios del alojamiento contiene todos los servicios requeridos, el alojamiento nos sirve*

                    if (serviciosTemp.containsAll(serviciosLista))

                    {

                        alojamientosFiltro2.add(aloj);

                    }

                }

                //*Paso3: repartir la cantidad de reservas entre los alojamientos que tengan la capacidad o parte de la capacidad necesaria*

                long cantidadTemp = cantidad;

                boolean end = false;

                int a = 0;

                while (a<alojamientosFiltro2.size() && end==false && cantidadTemp != 0)

                {

                    Alojamiento aloj = alojamientosFiltro2.get(a);

                    //*Si en el Alojamiento me caben todas las reservas, uso solamente este*

                    if (aloj.getCapacidad()>=cantidadTemp)

                    {

                        //*use*

                        int c = 0;

                        while (c<cantidadTemp)

                        {

                            long ultimoIdReserva = alohandes.darUltimoIdReservas();

                            ultimoIdReserva+=1;

                            long costoTemp = aloj.getPrecio() + aloj.getPrecioAdministracion() + aloj.getPrecioSeguro();

                            costoTotal+=costoTemp;

                            n++;

                            //*Creamos y adicionamos cada reserva como si fuera una reserva aislada de reserva colectiva*

                            adicionarReservaConParametros((int) ultimoIdReserva, "reserva\_masiva\_"+idMasivoString, fecha\_llegada, fecha\_salida, (int)costoTemp, cedula, aloj.getIdAlojamiento());

                            c++;

                        }

                        end = true;

                    }

                    //*De lo contrario, meto todas las reservas que me alcancen*

                    else

                    {

                        //*creamos una cantidad de reservas igual a = 'cantidadTemp'.*

                        int o = 0;

                        while (o<aloj.getCapacidad())

                        {

                            long ultimoIdReserva = alohandes.darUltimoIdReservas();

                            ultimoIdReserva+=1;

                            long costoTemp = aloj.getPrecio() + aloj.getPrecioAdministracion() + aloj.getPrecioSeguro();

                            costoTotal+=costoTemp;

                            n++;

                            adicionarReservaConParametros((int) ultimoIdReserva, "reserva\_masiva\_"+idMasivoString, fecha\_llegada, fecha\_salida, (int)costoTemp, cedula, aloj.getIdAlojamiento());

                            o++;

                        }

                        //*Restamos las reservas que ya anotamos*

                        cantidadTemp -= aloj.getCapacidad();

                    }

                    a++;

                }

                resultado+="\n\nOperacion Realizada con exito!\n";

                resultado+="\nEl costo total de la reserva colectiva es: "+costoTotal+"000 COP\n";

                resultado+="\nEl numero total de reservas creadas fue: "+n+"\n";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

                panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

            }

        }

        catch (Exception e)

        {

            e.printStackTrace();

            String resultado = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultado);

        }

Finalmente, se muestra en la interfaz la cantidad de reservas generadas y el costo total de la creación de dicha reserva colectiva. Además, es importante mencionar que las operaciones SQL que se ejecutan son las mismas, pero de manera masiva, que se usaron para el requisito funcional de registrar una reserva. Por ende, no es necesario mostrar el código de dicha sentencia puesto que ya ha sido mostrado anteriormente.

### **RF8: Cancelar reserva colectiva**

Este es un requisito funcional con operaciones CRUD simple y sencillo. La implementación de esta operación es muy básica puesto lo único que se hace es traer todas las tuplas de la base de datos de la relación Reservas, y verificar que, en tipo de contrato, las reservas contengan el String “reserva\_masiva” + el id de la reserva colectiva, por eso es importante darle id’s a las reservas masivas. De esta manera, una vez se tiene filtradas las reservas que cumplen con estas condiciones, son eliminadas una a una de la base de datos con el requisito funcional de eliminar una reserva, que ya ha sido diseñada hasta este punto. Por lo cual no es necesario mostrar el código SQL que elimina la reserva puesto ya ha sido mostrado. Sin embargo, si es fundamental mostrar la implementación de lo anteriormente mencionado para concretar de manera efectiva este requisito funcional:

/\*\*

*\* @RF8*

*\* Borra de la base de datos todas las reservas que pertenecen*

*\* a la reserva colectiva con el identificador dado por el usuario*

*\* la cancelacion tiene una multa segun el tiempo de cancelacion*

*\* Usa: @RF5*

\*/

    public void eliminarReservaColectiva( )

    {

        try

        {

            String idTipoStr = JOptionPane.showInputDialog (this, "Id de la reserva colectiva?", "Borrar Reserva por Id", JOptionPane.QUESTION\_MESSAGE);

            if (idTipoStr != null)

            {

                //*Paso 1: Traemos todas las reservas de la reserva colectiva correspondiente*

                List<Reserva> reservas = alohandes.darReservas();

                List<Reserva> reservasFiltro1 = new ArrayList<Reserva>();

                int n=0;

                int multaTotal=0;

                for (Reserva res:reservas)

                {

                    if (res.getTipoContrato().equals("reserva\_masiva\_"+idTipoStr))

                    {

                        reservasFiltro1.add(res);

                    }

                }

                //*Paso 2: usamos @RF5 parametrizado para eliminar todas las reservas encontradas*

                for (Reserva res:reservasFiltro1)

                {

                    n++;

                    int multa = eliminarReservaPorIdConParametros((int)res.getIdReserva());

                    multaTotal+=multa;

                }

                String resultado = "";

                resultado+="En eliminar reserva colectivas: \n\n";

                resultado+="\n Total de reservas eliminadas: "+n;

                resultado+="\n Multa total: "+multaTotal+"000 COP";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

                panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

            }

        }

        catch (Exception e)

        {

// *e.printStackTrace();*

            String resultado = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultado);

        }

    }

### **RF9: Deshabilitar una oferta de alojamiento**

Este es un requisito funcional CRUD y hasta el momento es el requisito mas complejo que se ha diseñado en términos de implementación. No obstante, se mencionará los rasgos más importantes creados para la ejecución de este requisito y así no ser tan metódicos. Este requisito necesita la creación de una nueva relación en la base de datos, la cual almacena los alojamientos deshabilitados y el motivo de porque ya no están disponibles. La tabla se llama Alojsdeshabilitados. La primera verificación que se realiza es ver si el alojamiento ingresado por parámetro ya esta deshabilitado, siendo así, no tiene sentido ejecutar la transacción y se informa esta situación al usuario mediante la interfaz. Posterior a esto, se inserta la oferta de alojamiento en la tabla de alojamientos deshabilitados. Después, se extraen, con el requisito funcional de eliminar una reserva de un alojamiento, todas las reservas del alojamiento y se reparten entre los alojamientos disponibles según sus disponibilidades. Todas estas subtransacciones se realizan una a una con el requisito funcional de adicionar una reserva a un alojamiento. Todo este proceso se le llama con el seudónimo de “Migrar”, así es como se referencia en la implementación del código, la cual se muestra a continuación:

//*@RF9: Deshabilitar temporalmente una oferta de alojamiento*

    // *y reubicar a las reservas actuales, dandole prioridad a las vigentes*

    public void deshabilitarOfertaAlojamiento()

    {

        String resultado = "En deshabilitar Alojamiento\n\n";

        try

        {

            String idAlojString = JOptionPane.showInputDialog (this, "id del alojamiento a deshabilitar?", "Deshabilitar un alojamiento", JOptionPane.QUESTION\_MESSAGE);

            int id\_aloj = Integer.parseInt(idAlojString);

            //*Assert: ver si el alojamiento ingresado ya esta deshabilitado*

            if (!alohandes.checkearDispAlojamiento(id\_aloj))

            {

                resultado += "Error de integridad: El alojamiento ingresado ya esta deshabilitado!";

                panelDatos.actualizarInterfaz(resultado);

            }

            else

            {

                Date date = new Date();

                Timestamp fecha = new Timestamp(date.getTime());

                String evento = JOptionPane.showInputDialog (this, "Razon/evento externo por el cual se va a deshabilitar?", "Deshabilitar un alojamiento", JOptionPane.QUESTION\_MESSAGE);

                if (idAlojString != null)

                {

                    //*Paso 1: Insertar la oferta de Alojamiento en la tabla de alojamientos deshabilitados*

                    alohandes.adicionarAlojamientoDeshabilitado(id\_aloj, fecha, evento);

                    //*Paso 2: extraer las reservas del alojamiento*

                    List<Reserva> reservas = alohandes.darReservas();

                    List<Reserva> reservasAloj = new ArrayList<Reserva>();

                    for (Reserva res:reservas)

                    {

                        if (res.getAlojamientoReservado()==id\_aloj)

                        {

                            reservasAloj.add(res);

                        }

                    }

                    //*Si no tiene reservas por reubicar, finalizamos*

                    if (reservasAloj.isEmpty())

                    {

                        resultado += "\nSe ha deshabilitado el alojamiento y no hace falta reubicar sus reservas porque no tiene ninguna.\n";

                    }

                    //*De lo contrario, continuamos*

                    else

                    {

                        List<Reserva> reservasParaMigrarPrimero = new ArrayList<Reserva>();

                        List<Reserva> reservasParaMigrarSegundo = new ArrayList<Reserva>();

                        //*Paso 3: eliminar las reservas del alojamiento deshabilitado para migrarlas usando @RF5*

                        for (Reserva res: reservasAloj)

                        {

                            //*@RF5*

                            resultado += "\n\nEliminando la siguiente reserva del alojamiento para que sea migrada:\n";

                            resultado += eliminarReservaPorIdConParametrosYTraza(resultado, (int)res.getIdReserva());

                            Timestamp fechaLlegada = res.getFechaLlegada();

                            Timestamp fechaSalida = res.getFechaSalida();

                            Date dateActual = new Date();

                            Timestamp fechaActual = new Timestamp(dateActual.getTime());

                            /\**\**

*\* Seleccionamos las reservas vigentes (Que estan en funcionamiento con la fecha actual)*

*\* Verificar si el timestamp intermedio está entre los otros dos*

\*/

                            if (fechaActual.compareTo(fechaLlegada) > 0 && fechaActual.compareTo(fechaSalida) < 0)

                            {

                                reservasParaMigrarPrimero.add(res);

                            }

                            else

                            {

                                reservasParaMigrarSegundo.add(res);

                            }

                        }

                        List<Alojamiento> allAlojamientos = alohandes.darAlojamientos();

                        List<Alojamiento> alojamientosFiltro2 = new ArrayList<Alojamiento>();

                        //*Filtramos los alojamientos que no esten deshabilitados*

                        for (Alojamiento aloj: allAlojamientos)

                        {

                            if (alohandes.checkearDispAlojamiento(aloj.getIdAlojamiento()))

                            {

                                alojamientosFiltro2.add(aloj);

                            }

                        }

                        //*Paso 4: adicionar las reservas a migrar usando @RF4*

                        //*Primero migramos las vigentes*

                        for (Reserva res: reservasParaMigrarPrimero)

                        {

                                //*Paso 4.1: repartir la cantidad de reservas entre los alojamientos que tengan la capacidad o parte de la capacidad necesaria*

                                long cantidadTemp = reservasParaMigrarPrimero.size();

                                boolean end = false;

                                int a = 0;

                                while (a<alojamientosFiltro2.size() && end==false && cantidadTemp != 0)

                                {

                                    Alojamiento aloj = alojamientosFiltro2.get(a);

                                    //*Si en el Alojamiento me caben todas las reservas, uso solamente este*

                                    if (aloj.getCapacidad()>=cantidadTemp)

                                    {

                                        //*use*

                                        int c = 0;

                                        while (c<cantidadTemp)

                                        {

                                            long ultimoIdReserva = alohandes.darUltimoIdReservas();

                                            ultimoIdReserva+=1;

                                            //*@RF4*

                                            resultado += "\n\nMigrando las reservas (Vigentes: prioridad alta!) del alojamiento deshabilitado a nuevos alojamientos:\n\n";

                                            adicionarReservaConParametros((int)ultimoIdReserva, "migrado\_por\_alojamiento"+id\_aloj, res.getFechaLlegada().toString(), res.getFechaSalida().toString(), (int)res.getCosto(),(int)res.getUsuario(), aloj.getIdAlojamiento());

                                            c++;

                                        }

                                        end = true;

                                    }

                                    //*De lo contrario, meto todas las reservas que me alcancen*

                                    else

                                    {

                                        //*creamos una cantidad de reservas igual a = 'cantidadTemp'.*

                                        int o = 0;

                                        while (o<aloj.getCapacidad())

                                        {

                                            long ultimoIdReserva = alohandes.darUltimoIdReservas();

                                            ultimoIdReserva+=1;

                                            resultado += "\n\nMigrando las reservas (Vigentes: prioridad alta!) del alojamiento deshabilitado a nuevos alojamientos:\n\n";

                                            adicionarReservaConParametros((int)ultimoIdReserva, "migrado", res.getFechaLlegada().toString(), res.getFechaSalida().toString(), (int)res.getCosto(),(int)res.getUsuario(), aloj.getIdAlojamiento());

                                            o++;

                                        }

                                        //*Restamos las reservas que ya anotamos*

                                        cantidadTemp -= aloj.getCapacidad();

                                    }

                                    a++;

                                    //*Restamos las reservas que ya anotamos*

                                    cantidadTemp -= aloj.getCapacidad();

                                }

                        }

                        //*Despues migramos las no vigentes*

                        for (Reserva res: reservasParaMigrarSegundo)

                        {

                            //*Paso 4.2: repartir la cantidad de reservas entre los alojamientos que tengan la capacidad o parte de la capacidad necesaria*

                            long cantidadTemp = reservasParaMigrarSegundo.size();

                            boolean end = false;

                            int a = 0;

                            while (a<alojamientosFiltro2.size() && end==false && cantidadTemp != 0)

                            {

                                Alojamiento aloj = alojamientosFiltro2.get(a);

                                //*Si en el Alojamiento me caben todas las reservas, uso solamente este*

                                if (aloj.getCapacidad()>=cantidadTemp)

                                {

                                    //*use*

                                    int c = 0;

                                    while (c<cantidadTemp)

                                    {

                                        long ultimoIdReserva = alohandes.darUltimoIdReservas();

                                        ultimoIdReserva+=1;

                                        //*@RF4*

                                        resultado += "\n\nMigrando las reservas (No-vigentes: prioridad baja...) del alojamiento deshabilitado a nuevos alojamientos:\n\n";

                                        adicionarReservaConParametros((int)ultimoIdReserva, "migrado", res.getFechaLlegada().toString(), res.getFechaSalida().toString(), (int)res.getCosto(),(int)res.getUsuario(), aloj.getIdAlojamiento());

                                        c++;

                                    }

                                    end = true;

                                }

                                //*De lo contrario, meto todas las reservas que me alcancen*

                                else

                                {

                                    //*creamos una cantidad de reservas igual a = 'cantidadTemp'.*

                                    int o = 0;

                                    while (o<aloj.getCapacidad())

                                    {

                                        long ultimoIdReserva = alohandes.darUltimoIdReservas();

                                        ultimoIdReserva+=1;

                                        //*@RF4*

                                        resultado += "\n\nMigrando las reservas (No-vigentes: prioridad baja...) del alojamiento deshabilitado a nuevos alojamientos:\n\n";

                                        adicionarReservaConParametros((int)ultimoIdReserva, "migrado", res.getFechaLlegada().toString(), res.getFechaSalida().toString(), (int)res.getCosto(),(int)res.getUsuario(), aloj.getIdAlojamiento());

                                        o++;

                                    }

                                    //*Restamos las reservas que ya anotamos*

                                    cantidadTemp -= aloj.getCapacidad();

                                }

                                a++;

                                //*Restamos las reservas que ya anotamos*

                                cantidadTemp -= aloj.getCapacidad();

                            }

                        }

                    }

                    resultado +=  "\n\nAlojamiento deshabilitado: \n"+alohandes.darAlojamientoPorId(id\_aloj)+"\n\n";

                    resultado += "\n Operación terminada";

                    panelDatos.actualizarInterfaz(resultado);

                }

                else

                {

                    panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

                }

          }

        }

        catch (Exception e)

        {

// *e.printStackTrace();*

            String resultadoE = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultadoE);

        }

    }

Finalmente, se retorna los alojamientos migrados y el éxito de la operación en caso de que se haya completado formalmente.

### **RF10: Rehabilitar un alojamiento deshabilitado**

Esta es una operación CRUD sencilla puesto se utilizo una estrategia similar a la del requisito 7. Debido a que se tiene una nueva tabla con los alojamientos deshabilitados y su información, simplemente se elimina la tupla del alojamiento que se quiere rehabilitar de esta tabla y se retorna que el alojamiento ahora esta disponible. Claro está, que si el alojamiento a rehabilitar no aparece en la tabla de alojamientos deshabilitados, no hace falta rehabilitarlo puesto aun sigue disponible. El condigo de dicha implementación se presenta a continuación:

/\*\*

*\* @RF10*

*\* Rehabilita un alojamiento deshabilitado*

\*/

    public void rehabilitarAlojamiento( )

    {

        String resultado = "En Rehabilitar Alojamiento\n\n";

        try

        {

            String idTipoStr = JOptionPane.showInputDialog (this, "Id del alojamiento que sera habilitado?", "Rehabilitar alojamiento por Id", JOptionPane.QUESTION\_MESSAGE);

            int id\_aloj = Integer.parseInt(idTipoStr);

            if (idTipoStr!=null)

            {

                //*Paso 1: verificar que el alojamiento este deshabilitado*

                if (alohandes.checkearDispAlojamiento(id\_aloj))

                {

                    resultado += "\nError: El Alojamiento con id " + idTipoStr + " actualmente esta habilitado normalmente. No hace falta rehabilitarlo.\n";

                    resultado += "\n\nOperacion Finalizada. No se rehabilito ningun alojamiento";

                    panelDatos.actualizarInterfaz(resultado);

                }

                //*Paso 2: eliminar la tupla de los alojamientos deshabilitados*

                else

                {

                    alohandes.rehabilitarAlojamiento(id\_aloj);

                    resultado += "\n\nOperacion Finalizada con exito! Se rehabilito el siguiente alojamiento:\n"+alohandes.darAlojamientoPorId(id\_aloj);

                    panelDatos.actualizarInterfaz(resultado);

                }

            }

            else

            {

                panelDatos.actualizarInterfaz("Operación cancelada por el usuario");

            }

        }

        catch (Exception e)

        {

            e.printStackTrace();

            String resultadoE = generarMensajeError(e);

            panelDatos.actualizarInterfaz(resultadoE);

        }

    }

Finalmente, se retorna al usuario el éxito de la operación si la tupla existe en la tabla de Alojsdeshabilitados. El comando SQL para eliminar la tupla de dicha relación generando la rehabilitación del alojamiento es el siguiente:

        Query q = pm.newQuery(SQL, "DELETE FROM " + pp.darTablaAlojDeshabilitado() + " WHERE id\_alojamiento = ?");

        q.setParameters(id\_aloj);

## Implementación y resultado de los requisitos funcionales de consulta

**3.3.1** RFC 1 – Mostrar el dinero recibido por cada proveedor de alojamiento durante el año actual y el año corrido

-- RFC1 --

SELECT op.nombre as proveedor, sum(r.costo) as ingresos

FROM A\_alojamientos a INNER JOIN

A\_Reservas r on a.id\_alojamiento = r.alojamiento\_reservado

INNER JOIN A\_operadores op on a.operador = op.id\_operador

GROUP BY op.nombre

ORDER BY sum(r.costo) desc;

**Resultado**



**3.3.2** RFC 2 – Mostrar las 20 ofertas más populares

-- RFC2 --

SELECT a.nombre\_alojamiento as Alojamiento, count(r.alojamiento\_reservado) as Popularidad

FROM A\_Reservas r INNER JOIN A\_Alojamientos a on

r.alojamiento\_reservado = a.id\_alojamiento

GROUP BY a.nombre\_alojamiento

ORDER BY count(r.alojamiento\_reservado) desc

FETCH FIRST 20 ROWS ONLY;

**Resultado**

**Tabla

Descripción generada automáticamente**

**3.3.3** RFC 3 – Mostrar el índice de ocupación de cada una de las ofertas de alojamiento registradas

-- RFC 3 --

SELECT ID\_ALOJAMIENTO, NOMBRE\_ALOJAMIENTO, (COUNT(ID\_RESERVA)/CAPACIDAD)\*100 AS "INDICE (%)"

FROM A\_ALOJAMIENTOS INNER JOIN A\_RESERVAS ON id\_alojamiento = alojamiento\_reservado

GROUP BY ID\_ALOJAMIENTO, NOMBRE\_ALOJAMIENTO, CAPACIDAD;

**Resultado**

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

**3.3.4** RFC 4 – Mostrar los alojamientos disponibles en un rango de fechas, que cumplen con un conjunto de requerimientos de dotación o servicios.

-- RFC 4 --

SELECT id\_alojamiento, nombre\_alojamiento, nombre\_servicio

FROM A\_ALOJAMIENTOS INNER JOIN A\_RESERVAS ON id\_alojamiento = a\_reservas.alojamiento\_reservado

                    INNER JOIN A\_SERVICIOS ON id\_alojamiento = a\_servicios.alojamiento

WHERE (fecha\_llegada > '10/05/2022' OR fecha\_salida < '01/05/2022') AND (nombre\_servicio = 'cocineta' or nombre\_servicio = 'TV cable')

GROUP BY id\_alojamiento, nombre\_alojamiento, nombre\_servicio;

**Resultado**

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

**3.3.5** RFC 5 – Mostrar el uso de AlohAndes para cada tipo de usuario de la comunidad

**3.3.6** RFC 6 – Mostrar el uso de AlohAndes para un usuario dado

-- RFC 6 --

SELECT nombre, count(\*) AS "RESERVAS REALIZADAS", sum(costo) AS "DINERO TOTAL PAGADO"

FROM A\_RESERVAS INNER JOIN A\_USUARIOS ON usuario = cedula

WHERE nombre = 'Juan Camargo'

GROUP BY nombre;

**Resultado**

****

**3.3.7** RFC 7 – Analizar la operación de AlohAndes

**3.3.8** RFC 8 – Encontrar los clientes frecuentes

-- RFC 8 --

SELECT nombre, COUNT(\*) AS "NUMERO DE RESERVAS"

FROM A\_ALOJAMIENTOS INNER JOIN A\_RESERVAS ON id\_alojamiento = alojamiento\_reservado

                    INNER JOIN A\_USUARIOS ON usuario = cedula

WHERE nombre\_alojamiento = 'Hostal Aguas'

GROUP BY nombre

HAVING COUNT(\*) >= 3;

**Resultado**

****

**3.3.9** RFC 9 – Encontrar las ofertas de alojamiento que no tienen mucha demanda

## Pruebas de transacción Exitosas (Demo – pruebas de unicidad)

### Pruebas exitosas para el Requisito Funcional 7

Para ver la implementación de este requisito, refiérase al método en InterfazAlohandesDemo llamado demoExitoR7. Por lo pronto, aquí esta el resultado de la ejecución de la prueba:

\*\*\*\*\*\*\*\* ESTADO INICIAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 2

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

¡Operación Realizada con éxito!

El costo total de la reserva colectiva es: 21000000 COP

El número total de reservas creadas fue: 50

\*\*\*\*\*\*\*\* ESTADO FINAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 2

---> Reserva con id: 3

---> Reserva con id: 4

---> Reserva con id: 5

---> Reserva con id: 6

---> Reserva con id: 7

---> Reserva con id: 8

---> Reserva con id: 9

---> Reserva con id: 10

---> Reserva con id: 11

---> Reserva con id: 12

---> Reserva con id: 13

---> Reserva con id: 14

---> Reserva con id: 15

---> Reserva con id: 16

---> Reserva con id: 17

---> Reserva con id: 18

---> Reserva con id: 19

---> Reserva con id: 20

---> Reserva con id: 21

---> Reserva con id: 22

---> Reserva con id: 23

---> Reserva con id: 24

---> Reserva con id: 25

---> Reserva con id: 26

---> Reserva con id: 27

---> Reserva con id: 28

---> Reserva con id: 29

---> Reserva con id: 30

---> Reserva con id: 31

---> Reserva con id: 32

---> Reserva con id: 33

---> Reserva con id: 34

---> Reserva con id: 35

---> Reserva con id: 36

---> Reserva con id: 37

---> Reserva con id: 38

---> Reserva con id: 39

---> Reserva con id: 40

---> Reserva con id: 41

---> Reserva con id: 42

---> Reserva con id: 43

---> Reserva con id: 44

---> Reserva con id: 45

---> Reserva con id: 46

---> Reserva con id: 47

---> Reserva con id: 48

---> Reserva con id: 49

---> Reserva con id: 50

---> Reserva con id: 51

---> Reserva con id: 52

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

### Pruebas exitosas para el Requisito Funcional 8

Para ver la implementación de este requisito, refiérase al método en InterfazAlohandesDemo llamado demoExitoR8. Por lo pronto, aquí está el resultado de la ejecución de la prueba:

\*\*\*\*\*\*\*\* ESTADO INICIAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 2

---> Reserva con id: 3

---> Reserva con id: 4

---> Reserva con id: 5

---> Reserva con id: 6

---> Reserva con id: 7

---> Reserva con id: 8

---> Reserva con id: 9

---> Reserva con id: 10

---> Reserva con id: 11

---> Reserva con id: 12

---> Reserva con id: 13

---> Reserva con id: 14

---> Reserva con id: 15

---> Reserva con id: 16

---> Reserva con id: 17

---> Reserva con id: 18

---> Reserva con id: 19

---> Reserva con id: 20

---> Reserva con id: 21

---> Reserva con id: 22

---> Reserva con id: 23

---> Reserva con id: 24

---> Reserva con id: 25

---> Reserva con id: 26

---> Reserva con id: 27

---> Reserva con id: 28

---> Reserva con id: 29

---> Reserva con id: 30

---> Reserva con id: 31

---> Reserva con id: 32

---> Reserva con id: 33

---> Reserva con id: 34

---> Reserva con id: 35

---> Reserva con id: 36

---> Reserva con id: 37

---> Reserva con id: 38

---> Reserva con id: 39

---> Reserva con id: 40

---> Reserva con id: 41

---> Reserva con id: 42

---> Reserva con id: 43

---> Reserva con id: 44

---> Reserva con id: 45

---> Reserva con id: 46

---> Reserva con id: 47

---> Reserva con id: 48

---> Reserva con id: 49

---> Reserva con id: 50

---> Reserva con id: 51

---> Reserva con id: 52

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel En eliminar reserva colectivas:

Total, de reservas eliminadas: 50

Multa total: 2100000 COP

\*\*\*\*\*\*\*\* ESTADO FINAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Finales ==========

---> Reserva con id: 2

========== Alojamientos Finales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

### Pruebas exitosas para el Requisito Funcional 9

Para ver la implementación de este requisito, refiérase al método en InterfazAlohandesDemo llamado demoExitoR9. Por lo pronto, aquí está el resultado de la ejecución de la prueba:

\*\*\*\*\*\*\*\* ESTADO INICIAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 2

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

Eliminando la siguiente reserva del alojamiento para que sea migrada:

Reserva [id\_reserva=2, tipo\_contrato=asd, fecha\_llegada=2023-04-01 12:03:45.0, fecha\_salida=2023-04-01 12:03:45.0, costo=420, usuario=123, alojamiento\_reservado=1]

Migrando las reservas (No-vigentes: prioridad baja...) del alojamiento deshabilitado a nuevos alojamientos:

Alojamiento deshabilitado:

Alojamiento [id\_alojamiento=1, operador=1, operador=100, operador=120, relacion\_universidad=nada, horarios\_recepcion=nada, precio\_administracion=100, precio\_seguro=200, nombre\_alojamiento=hotel]

Operación terminada

\*\*\*\*\*\*\*\* ESTADO FINAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Finales ==========

---> Reserva con id: 1

========== Alojamientos Finales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

### Pruebas exitosas para el Requisito Funcional 10

Para ver la implementación de este requisito, refiérase al método en InterfazAlohandesDemo llamado demoExitoR10. Por lo pronto, aquí está el resultado de la ejecución de la prueba:

\*\*\*\*\*\*\*\* ESTADO INICIAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 1

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

Operacion Finalizada con exito! Se rehabilito el siguiente alojamiento:

Alojamiento [id\_alojamiento=1, operador=1, operador=100, operador=120, relacion\_universidad=nada, horarios\_recepcion=nada, precio\_administracion=100, precio\_seguro=200, nombre\_alojamiento=hotel]

\*\*\*\*\*\*\*\* ESTADO FINAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Finales ==========

---> Reserva con id: 1

========== Alojamientos Finales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

## Pruebas de transacción No-Exitosas (Demo – pruebas de unicidad)

### Pruebas No-Exitosas para el Requisito Funcional 7

Para ver la implementación de este requisito, refiérase al método en InterfazAlohandesDemo llamado demoFracasoR7. Por lo pronto, aquí está el resultado de la ejecución de la prueba:

\*\*\*\*\*\*\*\* ESTADO INICIAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 1

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

Error: el usuario con la cedula '000000' no existe en la base de datos de AlohAndes!

### Pruebas No-Exitosas para el Requisito Funcional 8

Para ver la implementación de este requisito, refiérase al método en InterfazAlohandesDemo llamado demoFracasoR8. Por lo pronto, aquí está el resultado de la ejecución de la prueba:

\*\*\*\*\*\*\*\* ESTADO INICIAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 1

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

Error: la reserva colectiva con el id: '0000' no existe!

\*\*\*\*\*\*\*\* ESTADO FINAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Finales ==========

---> Reserva con id: 1

========== Alojamientos Finales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

### Pruebas No-Exitosas para el Requisito Funcional 9

Para ver la implementación de este requisito, refiérase al método en InterfazAlohandesDemo llamado demoFracasoR9. Por lo pronto, aquí está el resultado de la ejecución de la prueba:

\*\*\*\*\*\*\*\* ESTADO INICIAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 1

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

¡Error de integridad: El alojamiento con id ‘1’ ya esta deshabilitado!

\*\*\*\*\*\*\*\* ESTADO FINAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Finales ==========

---> Reserva con id: 1

========== Alojamientos Finales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

### Pruebas No-Exitosas para el Requisito Funcional 10

Para ver la implementación de este requisito, refiérase al método en InterfazAlohandesDemo llamado demoFracasoR7. Por lo pronto, aquí está el resultado de la ejecución de la prueba:

\*\*\*\*\*\*\*\* ESTADO INICIAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Iniciales ==========

---> Reserva con id: 1

========== Alojamientos Iniciales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel

Error: El Alojamiento con id 2 actualmente esta habilitado normalmente. No hace falta rehabilitarlo.

Operacion Finalizada. No se rehabilito ningun alojamiento

\*\*\*\*\*\*\*\* ESTADO FINAL DE LA BASE DE DATOS \*\*\*\*\*\*\*\*

========== Reservas Finales ==========

---> Reserva con id: 1

========== Alojamientos Finales ==========

---> Alojamiento con nombre: hotel

---> Alojamiento con nombre: hotel